An Embedded Domain Specific Language to Model, Transform and Quality Assure Business Processes in Business-Driven Development

Luana Micallef

Supervisor: Dr. Gordon Pace

Department of Computer Science

Faculty of Information and Communication Technology

University of Malta

June 2008

Submitted in partial fulfilment of the requirements

for the degree of B.Sc. I.T (Hons.)

To all my family

Abstract

Business process models are produced by business analysts to graphically communicate the business requirements to IT specialists. As business processes are updated to meet the new demands in the competitive market, the underlying IT solution is adapted, to reflect precisely the current goals of the organisation. The models should then act as an abstract representation of the solution. It is essential to adapt to Business-Driven Development (BDD), whereby models are refined into the IT solution and implemented in a Service-Oriented Architecture. This means that models must be free from data and control-flow errors, such as deadlocks. If models are not quality assured at the modelling phase, errors would be discovered later and the entire BDD lifecycle would have to be repeated. Combining model transformations with quality assurance would help modellers to preserve the correctness of models and rapidly carry out modifications.

Although various modelling languages have been developed to assist modellers in the production of high quality business process models, none of them adopted a functional approach, based on higher-order logic. As BDD is being adopted by most organisations, the need for such a language is becoming more evident. Since specialized functionality is required, a general-purpose language is not really necessary. Instead, a domain-specific language which provides the right abstraction and captures precisely the semantics of the business process modelling domain, should be developed. The definitions of the models would be easy to comprehend and reason about, by anyone who is not necessarily an IT specialist. However, since languages are made up of domain independent and dependent linguistic components, it is more cost effective and feasible to embed the new language in a general-purpose language.

In this project we present a domain specific language embedded in the functional language, Haskell, to model, transform and quality assure business processes in Business-Driven Development. By adopting a functional approach, we developed a language: 1) with which various models can *rapidly* be produced in a *concise* and *abstract* manner, 2) allows users to focus on the required *behaviour* rather than its implementation, 3) ensures that all the required details, to generate the *executable* code, are specified, 4) the abstract representation can be *interpreted*, *analysed* and *transformed* in various ways, 5) *quality assures* models by carrying out three types of checks; by *Haskell's* type checker, at *construction-time* through our embedded type system, and by specialised functions that *analyse* the components in the model.

By embedding our language in Haskell, the models, quality assurance checks and transformations are essentially functions which can easily be composed and defined. Connection patterns, defined in the language, play an important role to ensure that definitions are concise, readable and easy to comprehend. Different from other previous modelling tools, users are able to define their own parameterized models and transformations. By generating a directed graph for the models, various types of analysis can be carried out with greater ease. Moreover, quality assurance can be combined to model transformations by declaratively defining pre and post conditions for each transformation. These conditions as well as transformations can easily be composed of other previously defined checks or transformations.

With this language, we aim to capture the domain semantics of IBM's WebSphere Business Modeler Advanced v6.0.2.

Acknowledgement

A very special thanks goes to my supervisor, Dr. Gordon Pace, for his support, patience and constant guidance throughout the entire project and the entire course. I would also like to express my gratitude to all my lecturers during these past four years.

I cannot forget to thank my family for their valuable support and understanding not only during this semester but ever since the start of my studies. A very special thank you goes to my mum who managed to put up with my weird behaviour during periods of great stress and tension.

To show my appreciation, I would like to dedicate this project and the completion of this degree to all my family.

Table of Contents

1. Introduction 4

1.1. Overview 4
1.2. Objectives 5
1.3. Document Structure 7

2. Functional Programming languages & Domain specific Embedded Languages 8

2.1. Introduction 8
2.2. Functional Programming Languages 9
2.2.1. Pure Functional Programming Language 10
2.2.2. Referential Transparency 10
2.2.3. Eager and Lazy Evaluation 10
2.2.4. Pattern Matching and Recursion 12
2.2.5. Higher Order Functions 12
2.2.6. Type System 14
2.2.7. Monads 17
2.2.8. The Current Challenge of Effects 20
2.3. Domain-Specific Languages 20
2.3.1. General-Purpose Languages vs Domain-Specific Languages 20
2.3.2. Examples of Domain-Specific Languages 21
2.3.3. Advantages and Disadvantages of Domain Specific Languages 22
2.4. Domain Specific Embedded Languages 23
2.4.1. Choosing an appropriate host language 27
2.4.2. Domain Specific Languages Embedded in Haskell 27
2.4.3. Techniques in Embedded Languages 30
2.5. Conclusion 40

3. Business Process Modelling 41

3.1. Introduction 41
3.2. General Concepts 41
3.2.1. Business Process Modelling 42
3.2.2. Business-Driven Development 43
3.3. Business Process Modelling Notations and Tools 46
3.3.1. Business Process Modelling Notation 47
3.3.2. IBM WebSphere Business Modeler 50
3.4. Model Transformations 60
3.5. Quality Assurance 61
3.5.1. SESE Fragments & Control-Flow Analysis Heuristics 63
3.5.2. Process Patterns & Anti-Patterns 64
3.6. Conclusion 65

4. Embedding IBM's WebSphere Modelling Language in Haskell 66

4.1. Introduction 66
4.2. Basic Modelling Functions & Language Features 67
4.2.1. Tasks 67
4.2.2. Gateways to Handle Control and Data Flow in the Process 73
4.2.3. Events 76
4.2.4. Repositories 77
4.2.5. Constant Input 78
4.2.6. Sub-processes 78
4.2.7. Connectors 79
4.2.8. Adding Additional Input and Output Control Flows 79
4.2.9. Specifying Input Source or Output Target Types for a Task 80
4.3. Managing data types 88
4.4. Embedding the Language Terms and Type System in Haskell 91
4.5. Type Classes in our Language 97
4.6. Tagging Process Fragments into Sub-Processes 101
4.7. Connection patterns 102
4.8. Parameterized Models 108
4.9. Related Work 111
4.10. Conclusion 114

5. Transforming and Quality Assuring Models 115

5.1. Introduction 115
5.2. The Importance of Model Transformations and Quality Assurance 115
5.3. Carrying out Basic Checks on Models 117
5.4. Constructing Basic Model Transformations 118
5.5. Quality Asssurance 120
5.6. Constructing Quality Assured Composite Transformations 126
5.7. Related Work 129
5.8. Conclusion 132

6. Evaluation and Case Studies 133

6.1. Introduction 133
6.2. Case Study 1 133
6.3. Case Study 2 147
6.4. Case Study 3 155
6.5. Case Study 4 160
6.6. Final Remarks on Case Studies 162
6.7. Other Evaluation Techniques 164
6.8. Conclusion 164

7. Conclusions 165

7.1. Introduction 165
7.2. Achievements 165
7.3. Future Work and Enhancements 166
7.4. Final Remarks 167

References 168

A. A Tutorial on our Language 174

A.1. Introduction 174
A.2. Getting Started 174
A.3. Basic and Complex Types 176
A.4. Basic Modelling Functions 178
A.4.1. Task 178
A.4.2. Constants 184
A.4.3. Repository 184
A.4.4. Start, Stop, End Node 185
A.4.5. Decisions, Merges, Forks, Joins 186
A.4.6.Packaging Process Fragments into a Sub-Process 189
A.5. Connection Patterns 190
A.6. Built-In Models 194
A.7. Ways How to Construct a Model 198
A.8. Parameterized Models 199
A.9. Basic Checks 201
A.10. Generating a Directed Graph and Quality Assuring Models 202
A.11. Creating Composite Model Transformations 202
A.12. Conclusion 203

B. Sample Models and Case Studies 204

B.1. Introduction 204
B.2. ABC Project 204
B.2.1. User-Defined Types - Business Items 205
B.2.2. Global Repositories 205
B.2.3. Tasks 206
B.2.4. Business Process 1: *Customer Order Handling* 207
B.2.5. Business Process 2: *Order Verification* 214
B.2.6. Business Process 3: *Payment Handling* 217
B.3. External Claims Assessor Management (ECAM) Project 220
B.3.1. User-Defined Types - Business Items 220
B.3.2. Global Repositories 221
B.3.3. Tasks 222
B.3.4. Business Process 1: *Assessor Determination* 223
B.3.5. Business Process 2: *Auto Claims Handling* 225
B.3.6. Business Process 3: *Auto Claims Process* 230
B.3.7. Business Process 3: *Auto Claims Submission* 231
B.4. Quickstart Finance Project 234
B.3.1. User-Defined Types - Business Items 234
B.3.2. Global Repositories 235
B.3.3. Tasks 235
B.3.4. Business Process 1: *Loan Application (As Is)* 235
B.3.5. Business Process 2: *Loan Application (To Be)* 237
B.5. Other Sample Models 240
B.5.1. Order Handling 240
B.6. Conclusion 247

Chapter 1

Introduction

1.1. Overview

When developing a system for a particular organisation, the business requirements and goals must be communicated to the IT specialists. While the business analyst is an expert in that particular business field, the IT specialist knows nothing about the business and the organisation. Similarly, the business analyst is not an IT specialist. However it is still vital for the business analyst to communicate the business requirements in profound detail for the IT specialist to learn more about the organisation and the problem, and thus try to come up with a feasible solution. For this reason, business process models are usually produced by the business analyst to graphically and textually communicate the current business processes to the IT department.

Since the IT solution is meant to reflect the current needs and goals of the business, the business process models and their respective implementation must not be viewed as two separate entities. Instead, the business process model must serve as an abstract description of the solution and thus used as documentation for the implemented system. For this to be possible, the business process models and their implementation must not evolve independently. However, it is a known fact that while the development process of a system requires a considerable amount of time for it to be fully designed, implemented and tested, the company needs to adapt its goals and processes rapidly to keep up with the competitive market. This situation emphasises the need for Business-Driven Development (BDD) (Mitra, 2005) (Koehler, Hauser, Küster, Ryndina, Vanhatalo, & Wahler, 2006) whereby implementations are directly derived from the business needs. In Business-Driven Development, processes are implemented in a Service-Oriented Architecture.

Adapting to Business-Driven Development and thus producing implementations for specific business processes by carrying out a number of refinements to the original description, brings about new responsibilities. Before deriving the implementation, it is important to ensure the quality of the produced models. If business analysts do not produce models of a high quality from which the final executable code could be derived, then the probability is that errors are discovered later on in the development process, leading to a waste of resources, mainly time and money. Besides quality assurance, the business analyst must also provide all the required details for the executable code to be derived. However business analysts are not IT experts and thus, it might not be so intuitive for the analyst to realize that specific details are actually required. Thus, to assist the modellers to rapidly transform the current *'as-is'* to the future '*to-be*' models, quality assured model transformations (Koehler, et al., 2007) are required. With such transformations, business analysts would be able to carry out a number of different transformations in just a few steps and receive an immediate feedback on the quality of the models.

Although various modelling languages and tools have been developed to assist modellers in the production of high quality models from which the IT solution can be derived, none of the languages adopt a functional approach, based on higher-order logic. As argued in (Koehler, et al., 2007), a declarative approach would be appropriate to define pre and post conditions to assure the quality of the models when in-place transformations are carried out. It would also be possible to allow users to define their own composite transformations. The advantages of such an approach were especially noted in (Koehler, Hauser, Sendall, & Wahler, 2005), where pre and post conditions of out-place transformations were represented in the Object Constraint Language and used successfully to refine the graphical models into the executable BPEL code. As defined by Backus's Turing Award paper (Backus, 1978), the main reason why a declarative and functional approach results to be more effective than other imperative approaches, is that users are able to abstract away from the implementation details and focus on what operations are required rather than how such operations should be implemented. This means that this approach brings about other advantages than simply those identified in (Koehler, et al., 2007) and (Koehler, Hauser, Sendall, & Wahler, 2005). The need for such a functional modelling language is becoming more evident, as more organisations are identifying the advantages of Business-Driven Development and are adopting this methodology.

Since specialized functionality is required within such a functional modelling language, a general-purpose language is not really necessary. Instead, a domain-specific language which provides the right abstraction and captures precisely the semantics of the business process modelling domain should be developed, such that the definitions of the models would be easy to comprehend and to reason about, by anyone who is not necessarily an IT specialist. However, as noted by Peter Landin (Landin, 1966), languages are made up of domain independent and dependent linguistic components. Thus, it is more cost effective and feasible to embed the new language in another well established general-purpose language. The tools, features and limitations of the chosen host language, would be inherited by the embedded language. Thus it is important for the language designer to identify an appropriate host. In this way, the language designers are able to reuse the infrastructure of the host and hence, focus more on the semantics of the new language (Hudak, 1998; Hudak, *Building domain-specific embedded languages*, 1996).

In this project, we present a domain specific language embedded in the functional language, Haskell (Jones S. P., 2003), to model, transform and quality assure business processes in Business-Driven Development.

1.2. Objectives

Our main objective is to develop a domain specific language, which captures precisely the domain of business process modelling. With such a language, users, who are not necessarily IT specialists, should be able to model, transform and quality assure business processes. Users should be able to rapidly produce good quality models in a concise and abstract manner with the least amount of effort and expertise. Besides being concise, models must be readable, easy to comprehend and most importantly type- safe. The language must assist the modeller in the product of such high quality models from which the underlying IT solution can be derived. For this to be possible, the language must also provide a number of basic transformations, to automatically carry out changes to the model, and quality assurance checks, to ensure that the modelled processes are free from data and control-flow errors, such as deadlocks and lack of synchronisation. The user should be able to define composite transformations and ensure their quality by defining pre and post conditions for each one. For this reason, the language must try to identify ill-typed and unsound models as early as possible to prevent errors from being propagated to the succeeding phases in the Business-Driven Development lifecycle. The earlier errors are trapped, the quicker the required IT solution is developed and the lower the costs.

By adopting a functional approach and using higher order logic, modellers should be able to focus on the required behaviour rather than the implementation of operations. Models, quality assurance checks and transformations are essentially functions which can easily be composed and defined. Similarly, users should be able to define their own parameterized models and transformations. Other advanced users might want to define their own quality assurance checks, in which case, either composite checks (based on the provided primitive checks) are defined or else the internal structure of the defined model can be analysed. To facilitate such analysis, the internal structure of the model, can be defined in terms of a directed graph which is generated by the language. Similarly, by adopting a deep embedded approach, the models defined in the language, should be easy to interpret, analyse and transform in various ways. For this reason, the language should be flexible enough to allow additional features to be added by either the modeller or a programmer (depending upon the nature of the extension). In this way, it should be possible in the future, for the language to be connected to other external tools such as model checkers to carry out complete state analysis on the models.

By embedding our language in a strongly statically typed language, such as Haskell (Jones S. P., 2003), it is possible to embed the required domain specific type system in that of the host, such that ill-typed models (i.e. models which contain incompatibly typed elements, where the output types of one are not equivalent to the input types of the element it is connected to), are identified at construction time (that is at compile-time), by Haskell's type checker. Other features of Haskell which result to be useful include its support for higher order functions. By making use of such functions, it is possible to define connection patterns. These patterns are merely functions, which as input take other functions. These are combined in a specific manner such that a new function is returned. Such patterns provide the ideal abstraction and modularity for models to be defined in a concise and abstract manner. By defining appropriate connection patterns, definitions are more readable and easier to comprehend. It should also be possible and easier for users to define their own parameterized models, which would help them to rapidly produce the required processes and reuse structures. These parameterized models are defined as normal functions which take some input. Depending upon the input, the required model is produced. By embedding our domain specific language in a well-established general-purpose language, we are able to inherit the domain independent infrastructure, tools and features, and thus we are able to focus on the domain semantics.

In this way, with our language we aim to assist modellers in the production of good quality models from which the IT solution can be can derived in Business Driven Development (BDD). Thus, in this project we develop a language:

- With which various models can *rapidly* be produced in a *concise* and *abstract* manner
- Allows users to focus on the required *behaviour* rather than the implementation of such behaviour
- The abstract representation can be *interpreted*, *analysed* or *transformed* in various ways
- Ensures that all the required details, for the *executable* code to be generated, are always provided
- *Quality assures* models by carrying out three types of checks:
    - by *Haskell's* type checker
    - at *construction-time* through its embedded type system
    - by specialised functions that *analyse* the model and ensures its *soundness*

With our language, we aim to capture the domain semantics of IBM's WebSphere Business Modeler Advanced v6.0.2[1] (IBM, 2006).

[1] http://www.ibm.com/developerworks/websphere/zones/businessintegration/roadmaps/modeler/roadmap_advanced.html

1.3. Document Structure

Chapter 2 gives a brief overview of the functional programming paradigm and embedded domain specific languages. In the following chapter, the Business Process Modelling domain is introduced with particular reference to IBM's WebSphere Business Modeler Advanced v6.0.2[1].

The development of our language, the approaches adopted and the issues encountered are discussed in Chapter 4. The techniques used to transform and quality assure the models are then investigated in Chapter 5. Both of these chapters contain a section which discusses and compares our approaches to other previous works.

Finally, a number of models defined in our language and defined using IBM's WebSphere Business Modeler Advanced v6.0.2, are analysed in Chapter 6. Investigation is also carried out to identify the different ways how models can be defined in our language.

The concluding chapter (Chapter 7) gives an overview of our achievements and some possible future works and enhancements. Following this, a tutorial on our language is provided as Appendix A and the full definitions of some sample models defined in our language (and used as case studies in Chapter 6), are available as Appendix B.

Chapter 2

Functional Programming Languages & Domain Specific Embedded Languages

2.1. Introduction

This chapter gives an overview of the functional programming paradigm and the techniques used in the development of domain specific languages. In the first section, the main concepts of functional programming languages are discussed with particular reference to Haskell (Jones S. P., 2003), which was chosen as the host for our embedded business process modelling language. In the second section, various domain specific languages and language design techniques are discussed, with particular reference to the concept of embedding domain specific languages in other general purpose languages such as Haskell.

2.2. Functional Programming Languages

Functional, logic and constraint programming languages are known as declarative languages. Different from the imperative programming paradigm, the main objective of such programming languages is to define what needs to be done rather than how it should be done. Functional programming languages fulfil this objective by expressing computation in terms of mathematical functions, avoiding state, side-effects and mutable data. In this way, rather than focusing on the state and variables of the program, values and data objects are considered and once values are bound to identifiers then such bindings cannot change.

Besides primitive functions, which are supplied with the language, user-defined functions can be defined by using function constructors. Similar to lambda calculus, these functions are treated as first-class objects and thus they can be stored in data structures, passed as an input parameter to a function or returned as a result of a function. In fact, since lambda calculus was designed to investigate function definition, recursion and application, functional languages can be considered as an implementation of the calculus augmented with some data types. Since pure functional languages do not have assignment commands, then values must be passed on as parameters to functions, such that the execution of a functional program would simply be the evaluation of an expression. This would avoid side effects and guarantee referential transparency, such that given a specific input then the function would always return the same result. Due to the application of functions, such languages are usually also known as applicative programming languages.

Although most pure functional programming languages have been emphasized and used in academia, other functional languages are successfully being used for commercial applications such as Erlang for concurrent applications, Mathematica for symbolic mathematics, R for statistics, J and K for financial analysis, and Lisp and ML for artificial intelligence. XSLT is an example of a functional domain specific language whereas Haskell (Jones S. P., 2003), is one the most commonly preferred host functional language to embed domain specific languages.

The importance of such as paradigm especially when handling complex systems is the ability to modularize such systems, through the application of concepts such as higher-order functions and lazy evaluation, as illustrated in (Hughes, 1990). Dealing with the meaning of the program and thus what needs to be done, developers are allowed to focus on the main objectives and specifications of the system. Implementation details are hidden away and possibly derived automatically through the analysis and interpretation of this high level mathematical representation of the system. In this way, programs would be written quicker and in a much more concise manner. Programs can be reasoned about formally and later on analysed and interpreted into a more concrete representation. In this manner, the developer can have greater confidence that the derived representation fulfils the specifications of this system, which would have been defined declaratively. Such an abstract and formal representation of the system specifications would also help to ensure the production of a complete documentation of the final system. Another important advantage of such a paradigm is the exclusion of side-effects, which is usually the main source of most bugs. Without side-effects, the order of execution would become irrelevant and the programmer does not need to specify the flow of control. Moreover, it would be easier and possible to execute such program fragments on multiple concurrent parallel architectures.

Various other concepts specific to the functional programming paradigm are discussed in the following sections. In certain situations, code fragments and features specific to Haskell are investigated. It is of utmost importance for the reader to understand such concepts as these shall be referenced in the forthcoming sections, which deal with the design and development of our business process modelling language.

2.2.1. Pure Functional Programming Language

Pure functional programming languages such as Haskell (Jones S. P., 2003), do not allow programmers to define variables. Instead, identifiers which refer to immutable persistent values have to be used, such that programs are solely made up of functions which given an input, always return the same output, independent of the history of execution. This makes such languages inherently referential transparent and thus more adequate to both formal and informal analysis and optimization. Moreover, having no impure functions, call-by-need evaluation is possible. Pure functions are also thread-safe and thus can be executed in parallel, possibly adopting call-by-future evaluation.

Functional languages such as Lisp are not pure as they accept side-effects in their programs.

2.2.2. Referential Transparency

Purely functional programming languages are referentially transparent. This means that given a specific input, a function would always return the same output every time it is invoked, independent of the execution history of the program.

This is certainly not the case with functions such as `DateTime.Now`, `PrintLn` and `ReadFile` (examples of commands used in imperative languages such as Pascal and object oriented languages such as Java and C#). Consider the `ReadFile` command. This function takes a file path as an input argument and returns the contents of the read file. Even though the same function is invoked multiple times for the same file path, the return value might vary depending upon the current contents of the file.

Global variables and *assignments* are other features of languages which are not referentially transparent. Global variables are usually used to help compute the output of specific functions in a program. Even though the same arguments are passed on to a function, due to changes in these global variables, multiple invocations of the same function can lead to different outputs.

Contrary to the above examples, *arithmetic operations* and *mathematical functions* are referentially transparent. For example, the expression `3+5` can be safely replaced by `8`, as both expressions return the same result. Similarly, the function `f(x)=x+3` always yields the same result for a particular value `x`. Even though these expressions are replaced with their values, still the same effects and output given a specific input is retained. For this reason, referential transparency is simply the result of lambda beta reduction, whereby the evaluation of an expression is simply the replacement of its arguments.

One of the most notable advantages of referential transparency is the ability to reason about the behaviour of the program without being concerned about the external environment and how this would affect the program. This would help both the programmer and compiler to: 1) statically analyze the code and thus identify bugs which might not be trapped during testing, 2) modify and optimize the code by means of parallelism and sub-expression elimination, and 3) prove the correctness of the program by checking it for all its possible expected input values. For this reason, programs would be more tractable mathematically than with other paradigms.

However, a program without side-effects is not really useful. All systems must produce some output and allow some input to interact with the user or interface with some external devices. In purely functional language this is only possible through the use of monads (discussed in Section 2.2.7).

2.2.3. Eager and Lazy Evaluation

Different functional languages use different evaluation strategies. Such strategies define when and how arguments of functions are processed when expressions within the function are evaluated. Two main strategies of evaluation are eager and lazy evaluation.

While languages using eager or strict evaluation, evaluate all the arguments and expressions within the function before they are actually needed, non-strict evaluation and lazy evaluation pass arguments to functions unevaluated and then allow the invoked function to determine when an expression is required and thus when it should be evaluated. The difference between these two strategies is illustrated in the following example:

Assuming
```
f x y = x + x
g x = 2 * x
```

then to evaluate `f(g 2)(2+3)` the following two processing traces are possible:

```
f (g 2) (2+3) -> f (2*2) (5)  -> f 4 5 -> 4+4  -> 8
```
Listing 2. 1

or
```
f (g 2) (2+3) -> (g 2) + (g 2) -> (2*2) + (2*2)  -> 4 + 4 -> 8
```
Listing 2. 2

The first processing trace (Listing 2.1) illustrates how the arguments and expressions of the functions are processed when *strict* evaluation is used, whereas the second (Listing 2.2) demonstrates the processing trace when *non-strict* evaluation is used.

With *strict* evaluation, the arguments of the function are evaluated first. These arguments include the function `g 2` and the expression `(2+3)`. The actual function `f` is evaluated later. Contrary to this strategy, with *non-strict* evaluation, `f` is the first function that is evaluated. However to be able to complete the evaluation of this function, the value of function `g 2` is required. Thus only then this function is evaluated.

From Listing 2.1 and Listing 2.2, it can be noted that whereas in strict evaluation the input expression `(2+3)` is evaluated, with non-strict evaluation, this expression is never evaluated and its value is never required in the evaluation of function `f`. Thus, the mechanism of call-by-value employed in strict evaluation might lead to the evaluation of uncalled expressions. On the other hand, although non-strict evaluation avoids such redundant evaluations by employing the call-by-need mechanism, expressions (such as `g 2` in the Listing 2.2) might be evaluated more than once, due to duplicates on substitution. To provide a more efficient implementation technique for non-strict languages, lazy evaluation was developed such that values of previously evaluated expressions are obtained from memory (for more details read Chapter 17 of (Thompson, 1999)). In this way, lazy evaluation contributes to increase the performance of programs. Moreover, programmers do not need to take into account the evaluation order or add additional code to prevent the evaluation of redundant expressions. This would help to preserve modularity and allow the user to simply focus on the behaviour of the required function.

Languages using lazy evaluation are also more expressive than eagerly evaluated languages. A function `f` is said to be strict if it obeys $f(\perp) = \perp$. In eager or call-by-value languages, all functions are strict. In lazy languages, functions do not necessarily need to be strict. Thus, while in a lazy language it is possible to express a function such as `f(1\0)=7`, in a strict language, a division by zero error is generated. In the former case, this is possible as the input argument `(1\0)` is not evaluated, since it is not required to output the value of `f`, that is `7`.

Another important feature is the ability of define and traverse infinite data structures, without the need to generate them; generating such data structures, leads to an infinite loop. Thus with lazy evaluation it is possible to deal with potentially infinite structures such as

`nums1 = [1..]` or `nums2 n = n : nums2(n+1)`

An indefinite but finite part of the structure can be used to perform some computation. For instance, if the first 3 values are required from the infinite data structure `nums1`, the function `take` can be used. For example `take 3 nums1` would return the list `[1,2,3]`, by evaluating just the first three elements of the list. Similarly, if the head of the list is required, then `head(nums1)` would immediately return `1`.However, if a programmer tries to carry out some operation on the entire infinite list such as `show nums` or `length nums` then the program will either fail to terminate or continue evaluating elements in the output list until it runs out of memory. Such infinite data structures can be useful to list all the prime numbers or to check the next possible move in a game. In the latter case, the next move is identified without necessarily having to evaluate the entire tree of all the possible infinite solutions

The elegance and usefulness of such evaluation is emphasized in (Hughes, 1990). Hughes claims that if any part of the program is complicated, then it is important for the programmer to try to modularize it by using lazy evaluation and higher-order functions. In a similar way, in (Jones S. P., 2007), Jones claims that lazy evaluation manages to unify data with control.

Despite the advantages and effectiveness of lazy evaluation, call-by-need is usually less efficient than call-by-value. To ensure that unnecessary expressions are not evaluated and duplicated expressions are not evaluated more than once, expressions are initially stored in memory. A significant cost is usually inquired to read, write and overwrite the values of these terms in memory. This also leads to another issue, that of predicting the amount of space required by lazy programs. Recognizing such costs at the time when Haskell was being developed, strict data types and functions such as `seq` (`seq :: a->b->b`) were defined in the Prelude[1] module. In this way, the programmer can enforce the evaluation of the first argument before returning the second, thus avoid unnecessary laziness and improve the performance of the program. Moreover, with lazy evaluation, it is not always intuitive for a programmer to identify the order of evaluation of a program, making it more difficult to debug and analyze the complexity of the program and handle changes in states, input/output and exceptions.

[1] http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html

Some of the earliest functional programming languages, such as Lisp and ML, employ strict evaluation. Other languages, which are also purely functional languages, such as Haskell (Jones S. P., 2003), Miranda and Clean, use lazy evaluation. Being aware of some of the drawbacks of lazy evaluation, most of the mentioned lazy languages also provide some strict functions and data types. Since lambda calculus provides a stronger theoretical foundation for lazy languages, as defined in (Hudak, 1989), laziness helps to keep the purity of such languages.

2.2.4. Pattern Matching and Recursion

A technique which is commonly used with functional reasoning, is pattern matching. To define the behaviour of a function, a number of instances of the same function with specific inputs, are defined to check for the presence of particular components. When evaluating the function, the input arguments are mapped on these equations and the closest and most specific match is chosen. For example:

```
length [] = 0
length (x:xs) = 1 + length xs
```

This function takes a list of elements and returns an integer indicating the length of this list. Thus, if the input list is an empty list ([]), the first equation would be chosen and 0 is returned. Otherwise, the second equation would be selected and the head (that is the first element) of the list would be bound to the variable `x`, whereas the tail (that the rest of the list) would be bound to the variable `xs`. This equation would then invoke the function `length` with `xs` as the input argument. In fact the function is recursively invoked until every element in the initial list is handled and the base case, `length[]=0`, is invoked.

The use of recursion and pattern matching contribute to make the code more readable and help the programmer to decompose the problem and define the behaviour of the function by reasoning about the operation that should be carried out on every single element in the list, applying then the same function on a similar list with fewer elements. In languages such as Haskell (Jones S. P., 2003), top-to-bottom pattern matching is employed.

Although a stack might have to be maintained to handle recursion, tail recursion is usually recognized by compilers and optimized into the some code which is usually used to implement iteration in imperative programs. Keeping in mind that functional programs have no side-effects, iteration is not possible, as some state or global variable would be required to keep count of the number of iterations. For this reason, the only looping mechanism of functional languages is recursion.

2.2.5. Higher Order Functions

Being based on higher order logic, functional programming languages treat functions are first-class objects such that they can be stored in data structures, passed as arguments or returned as a result of a function. Since given the appropriate input, a value is returned, functions are mathematical values just as much as numbers. In this way, functions can be considered as the main abstraction mechanism of such a paradigm, by being abstract representations of some value or behaviour. They also contribute to modularize a program especially when higher-order functions are used. Such functions assume one or more functions as input and possibly return some other function.

Applying a Function to a Data Structure

An example of such a function which is commonly used in functional programs is the `map` function. The aim of such a function is to carry out a particular operation on every single element in a list and return a list with the computed values. Thus,

```
map  abs  [-6,2,-3]
```

which is equivalent to

```
[abs (-6) , abs 3 , abs (-3)] = [6,2,3]
```

Similarly, binary operators can also be passed on as input to the function `map`, example

```
map (3+) [-6,2,-3] = [-3,5,0]
```

In Haskell, such functions are pre-defined (in the Prelude[1] module) and can be implemented recursively as follows:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Combining Functions

In the previous example, functions are passed on as an input argument and applied to a data structure. Other higher-order functions are used to combine functions and return another new composite operation. The infix functional composition operator (.) is commonly used for such a purpose (refer to Listing 2.3).

```
(f . g) x = f (g x)
```

Listing 2. 3: The higher-order function `(.)`

However such a function is valid only if the output type of the function `g` is equivalent to the input type of function `f`.

Curried Functions

Higher-order functions also enable the technique first invented by Moses Schönfinkel and Gottlob Frege, known as currying (for more details refer to (Hindley & Seldin, 1980)). Thus, if a function takes multiple arguments as input, then this is expressed a function that takes one argument and returns another function. This second function would once again take one input argument and return another function, until all the input is used and the function returns a value. For example, assume a function `f` takes two input arguments `x` and `y` and outputs some value `z`, this can be defined either as

```
f (x,y) = z        or        f x y = z
```

The difference between these two definitions is that, while the first evaluates only if both `x` and `y` are passed on as input pair to the function, the second would return the value `z`, if both `x` and `y` are passed as input argument, or a function which takes `y` as input and output some value `z`, if only `z` is passed as an input argument. It is easier to note the difference by analysing the type signature of these functions (where `a`, `b` and `c` are type variables):

```
f :: (a,b) -> c            f :: a -> b -> c
f (x,y) = z                f x y = z
```

[1] http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html

One of the advantages of such a technique is that functions can be defined as infix operators (as illustrated with (.)). Moreover, partial application of functions would result useful in cases such as

```
map (3+) [-6,2,-3]
```

In this example, the function which is passed on to the `map` function, is not the `(+)` operator but the higher-order function which is returned when the `(+)` is partially applied to `3`.

2.2.6. Type System

One of the most important features of a programming language is its type system. Through such a system compilers are able to classify values and expressions into types and identify how these should be handled and manipulated. Thus one of the major functionalities of such a system is to identify typing errors in programs such as the application of string operations with numerical arguments example `isUpper 2`. The type system is also important for the compiler to optimize the code. Types are usually used to allow programmers to reason about programs at a higher level and hence abstract away from the actual representation of a specific value. For example, a String is actually an array of characters, and thus an array of bytes, but for a programmer it is a String. This also contributes to modularity.

Types can also illustrate the meaning of programs and the programmer's objective when defining such functionality. Thus since functional programmers focus on what behaviour is required, rather than how this should be implemented, compilers of strongly typed languages such as Haskell, try to identify errors and possible optimizations by understanding the meaning of programs through *type analysis* and *type inferencing*.

Statically vs. Dynamically Typed Languages

Type analysis or type checking is usually carried out to ensure that type constraints are enforced. This type of checking can be done either at compile-time, that is statically, or at run-time, that is dynamically.

One of the advantages of *static typing* or early binding is the possibility to identify errors as early as compile-time. Another notable advantage is the ability for compilers to produce optimized code which results in more efficient program executions. Although languages such as C, C++, C# and Java employ some form of static typing, there is a notable difference between such typing in these languages and the same form of typing in Haskell. In such languages, static typing is only possible through the annotation of variable, function and method declarations. These annotations would allow the compiler to determine the types of values and expressions and thus generate the correct code, and identify the amount of memory that needs to be allocated. Thus, since all the typing information about the program is obtained from these annotations, which have to be explicitly defined by the programmer, it is very much prone to error. Moreover, if the programmer ignores typing errors identified by the compiler or type casts some value or expression in the program, the type system of these languages can very easily be broken. This clearly illustrates how easily type mismatch errors can be introduced in the code, causing programs to crash. In contrast to this, types in Haskell can automatically be inferred by its inferencing type system, without the need for any annotations, such that, even though static typing is employed, programs are still as concise as those in dynamically typed languages.

On the other hand, languages such as Perl, Ruby and Python use *dynamic typing*. This is also known as late-binding since values and expressions are dynamically bound to some type at run-time. Thus, at compile-time nearly no type checking is carried out. For this reason, no specific annotations are required. This leads to programs which are much more concise than those implemented in some statically typed languages mentioned above. While programs are compiled faster than those implemented by some statically typed language, the execution of such programs tends to take more time. This is due to the fact that most of the typing errors have to be trapped at run-time, which, if encountered, might generate exceptions or cause the program to crash. If errors are ignored, programs might lead to unexpected behaviour.

In (Meijer & Drayton, 2004), the authors argue that a balance between dynamically and statically typed languages should be attained. According to the authors, this can be achieved by trying to use static typing where possible and dynamic typing when needed. A comprehensible study on static and dynamic type checking has been carried out in (Matthews, 1987).

Why Haskell's Type System is Different

Although Haskell is statically typed, the approach adopted by this pure functional language is far more different than that in other statically typed languages. Due to the numerous extensions which have been designed and applied to the language, in (Wadler, Hudak, Hughes, & Jones, 2007) Haskell is defined as "*a type-system laboratory*". All these extensions have contributed to the development of a language, which is statically and strongly typed, able to infer appropriate types of functions whose type signature is not explicitly defined.

What distinguishes Haskell's type system from that of other languages, is its *type checker*. This is used to better understand the program at compile-time by carrying out type analysis of the program. Moreover, besides strictly prohibiting the use of type casting, this type checker does not permit any of the type errors to be ignored. Although such strongly typed languages are type safer than other forms of typing, they might still not be able to guarantee complete safety. However, if a programmer wants to define his own type system over that of Haskell, then this can be easily done by using various type-system functionalities provided in Haskell.

Besides carrying out type analysis, Haskell is also capable to *infer types*. Types in most of the cases are optional. Thus, besides eliminating the need for programmer to add additional annotations to explicitly define the types of values and expressions, these features try to improve on the safety provided by most statically typed languages and at the same time, allow programmers to produce code that is concise as that of dynamically typed languages. Since the compiler tries to determine the meaning of the program and the programmer's intentions while defining the functions through type analysis and type inferencing, similar to UML diagram, types in Haskell express high-level design, with the added advantage that type signatures are machine-checked.

Types of values, expressions and functions in Haskell are defined through a type signature. Thus the type for the functional composition operator defined in Listing 2.3, is expressed by the following type signature:

```
(.) :: (b->c) -> (a->b) -> a -> c
```

Listing 2. 4: The type signature of function composition `(.)`

From the above, it can be noted that, since it is a curried higher order function, the types `(b->c)` and `(a->b)` respectively refer to the first and the second function which are passed on as input. Moreover, the output type of the innermost function in the composition (passed on as the second input) is equivalent to the input type of the outermost function in the composition (passed on as the first input), such that the output type of `(.)` is essentially another function of type `a -> c`. Note that `a`, `b` and `c` are type variables.

In this way, by defining the type signature of functions and data structures above the actual definition, the compiler is able to check that the programmer's intended types, match that of the actual function. Additionally, such a signature can act as part of the documentation, enabling other readers or possibly other programmers who want to invoke such functions, understand how such functions or structures should be used.

Some other features which strongly typed functional languages, such as Haskell, provide are *Abstract Data Types*, *Type Classes* and *Polymorphism*. These concepts are briefly discussed in the following sections. For more details refer to (Thompson, 1999).

Abstract Data Types

Types provide the right abstraction for programmers to deal with values of a specific type, by ignoring the underlying implementation of values of such types. A programmer can define a new *data type* as illustrated in the following example:

```
data Tree a = Empty | Node a (Tree a, Tree a)
```

This is essential a recursive parametrized type. Thus, the complete type is inferred when the type variable is defined. For instance, `Tree String` states that the expected value is a tree of type `String`, and thus, string values are expected at the nodes of the tree. If the tree is empty, then the constructor `Empty` can be used. Besides a value, a node also defines the sub-tree on the left and the other on the right. Since these sub-trees are essentially other trees, they are recursively defined by the data type `Tree a`.

Polymorphism, Parameterized Types, Type Classes, Overloading

Polymorphic techniques are often used to handle different typed values in a uniform manner. Such techniques are usually divided into two, mainly, *parametric polymorphism* and *ad-hoc polymorphism*.

As illustrated in Listing 2.4, if a function accepts some input or produces some output whose type is not specified, type variables are used to represent such types in the type signature. In this way, the function would be evaluated irrespective of its argument types. Such functions are known as *parametrically polymorphic*.

On the other hand, with *ad-hoc polymorphism* or *overloading*, the behaviour and thus the implementation of functions, varies depending upon the specific argument types. For this reason, such polymorphism only permits the application of specific functions with argument types, for which a specific implementation has been explicitly defined.

The approach chosen by Haskell to allow such ad-hoc polymorphism is the use of *type classes*. Although the approach was introduced early in the design process by Wadler and Blott (Wadler & Blott, 1989), the true potential of such an approach was only recognised later on in the development process until finally, Haskell programmers could not do without type classes. Some of these interesting applications of type classes were in fact published by Jones, eight years later in (Jones, Jones, & Meijer, 1997). Examples of such applications include: computations at the type level (mainly through functional dependency), generic programming (i.e. define generic functions that behaves in a uniform manner independent of the argument data types) and testing. Other generalisations are still being explored and investigated.

Haskell also supports *multiple-parameter type classes*. These generalize the interpretation of classes to relations on types such that specific implementations of operators defined in a class are defined for each parameter set instance. The first application which was suggested by Wadler and Blot (Wadler & Blott, 1989) was the use of a two parameter class `Coerce` defined to describe some subtyping relation. Other applications of such type classes were later defined in (Jones, Jones, & Meijer, 1997). The type class in Listing 2.5 defines the operations that are carried out on collections:

```
class Collection c e where
       elemOf :: c -> e -> Bool
       addElem :: c -> e -> c
```

Listing 2. 5: The type class `Collection`

The type variable `c` represents the type of the entire collection, whereas type variable `e` defines the type of the elements stored in the collection. The class has two operators; `elemOf`, which given a collection and an element, checks whether the element is present in the collection, and `addElem`, which given a collection and an element, adds the element to the collection. Depending on the type used to represent the entire collection and the type of the elements stored in the collection, then the appropriate implementations of these operators are deduced and used. A possible instance of this class is defined in Listing 2.6.

```
instance Collection [a] a where
        elemOf = flip elem
        addElem = flip (:)
```

Listing 2. 6: Instance of the class `Collection` defined in Listing 2.5

In this instance the collection is represented as a list of elements of type `a`. Various implementations of the operators defined in the class are inferred depending upon the values of both type variables `c` and `e`. However, from the above instantiation, it is intuitively notable that if type `c` is known, type `e` can be inferred automatically. Thus if the collection is defined as a list of elements of type `a`, then `e` must represent type `a`. To inform the compiler about this kind of relationship between types, a *functional dependency* should be added. The class declaration would thus be defined as:

```
class Collection c e | c -> e where
        …
```

The extra notation | `c-> e` claims that `c` uniquely identifies `e` such that given `c`, there is only one `e`. Additionally another functional dependency can be defined, example `e -> c`. Thus multiple functional dependencies can be defined in a class. The application and usefulness of such functional dependency constraints are discussed in more detail in (Jones M. P., 2000) .

2.2.7. Monads

Due to referential transparency of purely functional languages, side-effects are not allowed. For this reason, state changes, input/output operations and exceptions, cannot be handled in the same way as other conventional languages. Moreover, due to lazy evaluation, it is not possible to determine the execution order of operations. However, in certain situations, an explicitly defined ordered execution sequence is required. A program without side-effects and without any interaction with the user or some external device, is not really useful. To ensure the purity of the language and include such side-effecting operations, in Haskell, monads are used.

This concept of monads has originated from category theory. This branch of mathematics tries to describe patterns applicable to various mathematical fields. Moggi (Moggi, 1989) uses monads with lambda calculus to semantically describe the features of state, exceptions and continuation in a structured manner. This technique was later extended by Wadler (Wadler P. , 1990) and applied to functional programs. Wadler continued to emphasise the importance of monads in functional programming in (Wadler P. , 1992).

Using monads, it is possible to define the sequence of execution of operations and to produce pure functions which return a computation of a specific type rather than an evaluated value. This computation can then be executed whenever the external value is provided. As defined in Hudak's paper (Hudak & Fasel, 1992), monads are merely containers that are instances of the type class `Monad` (Listing 2.7).

```
class Monad m where
        (>>=) :: m a -> (a -> m b) -> m b
        return :: a -> m a
```

Listing 2. 7: The type class `Monad`

Thus, a type `m` is a monad, if it implements the operations `bind(>>=)` and `return`. Monad `m` can be considered as a higher order type since it acts as a container over some other type such as `m a` or `m b`. The `bind` operator (`>>=`) takes a container (`m a`) and a function (`a -> m b`) and returns a new container `(m b)`, such that by accepting the value from the first container, the function would return the second container. The `return` operator is used to lift a simple value into a container.

Originating from category theory, monadic operators are expected to satisfy the following properties (Wadler P. , 1990):

- `return a >>= f      ≡    f a`
- `m >>= return        ≡    m`
- `(m >>= f) >>= g     ≡    m >>= (\x -> f x >>= g)`

All these operators and properties provide the required modularity and abstraction to hide all the underlying computations and sequencing of operations. Functions with monads remain pure and thus they can be understood and handled by the compiler. A monadic function is distinguished from a non-monadic one from its return type. In this way, Haskell's type system keeps the two kinds rigorously separated.

Some of the commonly used monads are the *List* monad, the *IO* monad, the *State* monad and the *Maybe* monad. The *List* monad is frequently used unknowingly in list comprehension. The other types of monads shall be discussed in the next sections. If more than one type of monad is required at the same time then monadic transformers should be used (example *StateIO* monad, which combines *State* and *IO*). More details about monadic transformers are available in (Liang, Hudak, & Jones, 1995).

IO Monad

This is one of the most important monads in Haskell since it allows interactions with the outside world. This to be possible through the use of functions provided in the I/O libraries. The following are examples of two such functions; the first reads input values, whereas the second displays messages on the screen.

```
getLine :: IO String                -- to read a string
putStr :: String -> IO()            -- to display a string
```

Maybe Monad

As the name suggests, this monad is often used in situations where a computation might fail. Thus to avoid exceptions and runtime errors, the constructor `Nothing` would be returned. This is defined in Listing 2.8:

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
       Nothing >>= f = Nothing
       (Just x) >>=  f   = f x
       return = Just
```

Listing 2. 8: `Maybe` Monad

In Listing 2.8, the binding of the constructor `Nothing` with function `f` returns `Nothing` (since there are no values to which `f` can be applied); binding `Just x` to `f`, simply results in the application of `f` to `x` (of the underlying type); `return` obtains the underlying value and returns it as an enclosed value in the type constructor `Just`.

State Monad

This monad is often used to mimic the concept of memory or global variables as in imperative programming. The state monad `State s a` is able to change state `s` before it returns a value of type `a`. Functions making use of this state and whose return type is `State s a`, would actually be returning a computation, which when given the current value of the state, would return both the new updated state (which is usually abstracted away from the user) as well as the required value of type `a`. This computation of type `State s a` is usually evaluated when some function such as `runState`, which provides the value of the state, is invoked. Listing 2.9 defines such functions and monads (these functions are simplified versions of the functions available in the Control.Monad.State.Lazy[1] module, which is built-in in Haskell).

```
newtype State s a = State (s -> (s,a))

runState :: State s a -> s -> (s,a)
runState (State f) x = f x

instance Monad (State s) where
      m >>= k    = State  ( \s  ->  let  (s’,x)  =  runState m s
                                     in   runState  (k x)  s’  )
      return x   = State (\s -> (s,x))
```

Listing 2. 9: `State` Monad and related functions

The `bind` operator states that after evaluating `m` with the current input state `s`, the new updated state and the returned value `x` are used to evaluate the function `k`, returning then the output of this function. In this way, the binding operator enforces the required execution sequence. The `return` operator simply returns value `x` as a state computation.

2.2.8. The Current Challenge of Effects

In (Jones S. P., 2007) Simon Peyton Jones, points out that their current challenge, is to find the right balance between strongly typed languages, such as Haskell, which do not allow any side-effects, and other languages, such as object oriented and imperative languages, which allow side-effects. Since the former languages do no allow side-effects, then they are safe but not that useful. On other hand, languages that allow side-effects are not considered safe, since any arbitrary uncontrolled effects are allowed.

The approach that language designers are trying to adopt is either to add restrictions to languages which allow arbitrary effects, or selectively permit effects to value-oriented languages which by default do not allow any side-effects.

The latter approach can be achieved by either widening the spectrum of functional languages such as Haskell (Jones S. P., 2003) or by defining domain specific languages such as Google Map/Reduce (Lämmel, 2006-2007). In both cases, types play a major role. In this way, similar to DARC[2] and HABES (Harmful Algal Blooms Expert Systems)[3], other large projects can be developed in value-oriented programming languages. One of the companies which already uses such languages is Gaulois.

A good example where the importance and usefulness of functional programming concepts have been noted and applied for object oriented programs is the development of LINQ (Language INtegrated Query) (Meijer, Beckman, & Bierman, 2006) as a Microsoft .NET framework component, which was released as

[1] http://haskell.org/ghc/docs/latest/html/libraries/mtl/Control-Monad-State-Lazy.html
[2] http://www.darc.com/
[3] http://www.habes.net/

part of .NET framework 3.5, in November 2007. This language makes use of lambda expressions to enable the usage of data as first class objects and thus add native data querying capabilities to .NET languages, irrespective of the original source of the data. Besides lambda expressions, other functional programming concepts were adopted. Some of these include: abstraction, the use of higher-level controllable representations and the elimination of side-effects, through the definition of lambda expressions. LINQ, in fact can be considered as an example of an attempt to move from languages allowing arbitrary effects to safe languages which allow controlled effects.

2.3. Domain-Specific Languages

In particular specific domains, just specific specialized functionality is usually required. Thus the use of Turing complete general-purpose languages might result rather useless and confusing rather than helpful, especially when users are not really familiar with programming concepts.

To facilitate the development of such specific systems, Jon Bently, way back in 1986, introduced the concept of *"little languages*" (Bentley, 1986). According to Bently, such languages should merely consist of a set of specialised statements with which any system particular to that specific domain, can be developed with greater ease than is usually the case with general-purpose languages. To further strengthen this argument, in 1988, Herndon and Berzins, claim

> *"Many tasks can be easily described by agreeing upon an appropriate vocabulary and conceptual framework. These frameworks may allow a description of a few lines long to replace many thousand lines of code in other languages"* (R. M. Herndon & Berzins, 1988)

This is in fact what Paul Hudak emphasizes in his papers (Hudak, *Building domain-specific embedded languages,* 1996) and (Hudak, 1998); the importance of programming languages to provide the right abstraction for that domain where they shall be used, to ensure the development of good software, which are easier to understand, reason about and maintain. For this reason, it is important to identify the different factors which distinguish a domain specific language from a general purpose one.

2.3.1. General-Purpose Languages vs Domain-Specific Languages

The main difference between these two types of languages is that, in contrast to domain-specific languages, general-purpose languages are usually Turing complete. Thus, such languages can be used to create different types of programs that can perform any computational task for any domain. Different languages make this possible through the use of different constructs. For example, in most imperative languages such as C, a sequence of expression statements are expressed through the use of a semicolon (;), this normal sequential execution is controlled through the use of if-then-else statements, whereas iterative execution is possible through use of constructs such as for and while loop. Similarly, in declarative languages such as Haskell and Prolog, although no specific loop constructs are available, completeness is guaranteed through recursion.

Such completeness is missing with most domain-specific languages. Their main objective is to solve problems for a particular domain and thus, they cannot be used to fulfil any type of task for any context. If on the other hand, a general purpose language had to be used for a specific domain, then the domain specific concepts would have to be expressed using some syntax of general-purpose languages, sacrificing expressivity for more functionality and capability. For example the language used for text manipulation is not really available as syntax in a general-purpose language. Similarly, it is quite tedious and hard to make a clear textual representation of a user interface, and try to focus on the requirements of the interface rather than its implementation.

Although domain-specific languages are usually not Turing complete, they provide a more intuitive system to develop programs specific to a particular domain.

2.3.2. Examples of Domain-Specific Languages

One of the first and most popular domain specific programming languages, created for educational use to encourage constructivist teaching, is the Logo programming language. Its first version was created by Wally Feurzeig and Seymour Papert at BBN, a research firm in Cambridge, Massachusetts (Feurzeig & Lukas, 1972). Through interactivity, modularity, extensibility and flexibility of the data types the language provides, they managed to design a teaching tool that is actually a variant of the functional LISP language, with easier to read dialect. Although today it is mainly known for its turtle graphics, it also caters for files, I/O, handling of lists and recursion.

This language is an excellent example to illustrate the importance of such domain-specific languages. Such languages capture so concisely the domain semantics, that they can be used very easily by anyone even by non-programmers, without requiring much training, and at the same time, meet the demands of advanced users. In fact, different variants of Logo were adopted by different schools to help children learn mathematics, advanced mathematics (by allowing them to discover relationships between mathematical concepts which they can then build upon) and to help students further develop problem solving skills. Moreover, MIT StarLogo and NetLogo are also being used in other educational domains such as social studies, economics, biology, physics and other sciences.

The UNIX shell scripting language (Arthur, 1986) is an example of a domain-specific language that is used for data organization. Streams, such as *stdin* and *stdout*, and operations, such as redirection and pipe, provide the right abstraction and domain notation to manipulate user input and data in files and to handle the organization and flow of data. Thus, although Turing complete, a distinguishing feature between such languages and general-purpose languages is still notable.

Another domain-specific language which is also Turing complete is XSLT (Extensible Stylesheet Language Transformations) (Kay, 2005), which is based on XML and used to convert XML data into other XML schemas or into other documents that are human-readable, such as HTML and XHTML. This declarative language provides the right syntax to define XSLT stylesheets to describe template rules which are used as instructions and directives for the XSLT processor to produce the required output document.

A specific language which is frequently used to manage and access relational databases is SQL (Structured Query Language). Way back in 1970, a model for RDBMS (Relational Database Management System) was first developed by Codd and then, basing on this same model, the database system "System R", was developed by an IBM research group in San Jose. Later on, in 1974, the SQL language, which at that time was known as SEQUEL, was designed as a simple language that can be easily used to retrieve and manipulate the data held in "System R", through the use of a consistent set of self-describing keywords and statements applicable to tabular structures (Chamberlin & Boyce, 1974).

A very successful domain language used for programming reactive systems is Lustre (Halbwachs, Caspi, Raymond, & Pilaud, 1991). This declarative and synchronous dataflow programming language is used for the development of some of the most critical control software such as in nuclear power plants, aircrafts and helicopters. One of its central features is the ability to perceive time and space requirements by simply looking at the code and to carry out static verification to detect design errors.

Another area where various functional domain-specific languages were developed, is hardware design. The first functional hardware description language, which was a variant of FP, was developed in 1984 by Sheeran and was called μFP (Sheeran, 1984). This first attempt to develop a functional domain specific language for hardware design brought about a number of advantages which were not possible to achieve using other previous hardware description languages such as Verilog and VHDL. Primarily, descriptions were more comprehensible, concise, easier to debug and modify. Connection patterns played an important role to ensure the appropriate abstraction and modularity and allow re-use of circuit descriptions. These connections are merely functions which given some circuit descriptions as input arguments are capable of combining them in some way, such that a new possibly composite or complex circuit is returned. Later on, Ruby (Sheeran, 1990) was developed as the successor of μFP, handling circuits as relations on streams. Other languages such as HDRE (O'Donnell, 1987), Hydra (O'Donnell, 1993), Hawk (Cook, Launchbury, & Matthews, 1998), Lava (Claessen, 2001) and Wired (Axelsson, Claessen, & Sheeran, 2005) were subsequently developed as embedded domain specific languages (discussed in Section 2.4.2). In (Sheeran, 2005), Sheeran defines how powerful functional languages can be for hardware design. Similarly, Cordina in (Cordina & Pace, 2006) discusses the advantages of a functional approach to design circuits and gives a brief historical overview of the different languages which were developed.

Other successful domain specific languages that are often used, include $\TeX$ and $\LaTeX$ for document preparation, Mathematica for symbolic interpretation, Lex for lexical analysis of programs and Yacc for program parsing.

2.3.3. Advantages and Disadvantages of Domain Specific Languages

According to Paul Hudak (Hudak, *Building domain-specific embedded languages*, 1996), the most important factor to write software of a good quality is abstraction, that is, the use of programming languages that provide the precise abstraction to produce programs for that specific domain, quickly and effectively. The language should "*capture precisely the semantics of the application domain -- no more and no less*" and the "*ultimate abstraction*", according to Hudak, is a domain-specific language (Hudak, *Building domain-specific embedded languages*, 1996). Although various abstraction mechanisms such as classes, objects, modules, higher-order functions, abstract data types and monads can be used to achieve these benefits with high-level languages, yet some programming expertise would still be required. Conversely, domain experts who are programmers should be able to use the domain-specific language and focus solely on the required behaviour and functionality, abstracting away from all the implementations details.

Hudak also emphasizes that programs developed using such languages are "*easy to understand, reason about, and maintain*" (Hudak, 1998). This is in fact possible since programmers knowledgeable of that specific domain, are allowed to use a limited vocabulary which is specific for that domain and which "*precisely captures the semantics*" (Hudak, *Building domain-specific embedded languages*, 1996) of that domain. Thus, as stated also in (R. M. Herndon & Berzins, 1988), a program written using a domain-specific language would be much more concise, more expressive and easier to comprehend than the same program written using a general-purpose language. This also means that it is easier and quicker to carry out changes in these programs as the possibility to introduce bugs is much less than is normally the case with other general-purpose languages which would require far more code to carry out the same change. Another advantage is that domain specific abstractions and constructs are not added indirectly as mapping of functions and objects stored in libraries. Instead as noted in (Sloane, Mernik, & Heering, 2003), these constructs are immediately incorporated from the start.

Despite all these benefits, the cost and effort to create the appropriate infrastructure for the new language are quite significant. Various issues are usually encountered during the evolution of the language and thus, years are usually required to launch the first complete version of a language. Still, additional changes would have to be carried out, once users start using it. The field of language design is rather challenging and time-consuming. Great deal of effort is required to establish the proper semantics of the language. Tools, such as parsers, interpreters, compilers and debuggers, and other components, such as development environments, which are not really domain specific, are not really available for these languages and thus, would have to be developed from scratch. The development of such tools are rather costly (example compiler involve code-generation, optimisation, type-checking, error messages). Moreover, since each language has its own compiler, it is not possible to incorporate features from different languages in the same source file (even though the domains overlap); the compiler of one language would not known how to interpret the construct of the other. After all the effort to develop the language, the worst situation is for users to be unwilling to invest the time and effort to learn the new language.

All these drawbacks make the notion of domain specific languages very impractical, and are often the cause of failure of most domain-specific languages. Thus, a solution to all these problems is the development of domain specific embedded languages.

2.4. Domain Specific Embedded Languages

As noted by Peter Landin (Landin, 1966), way back in 1966, programming languages are made up of a domain independent linguistic framework and a set of domain dependent components. Thus, by defining the terms and the type system of the new language using a general-purpose language, a new embedded domain specific language would be constructed. In this way, the chosen general-purpose language would serve as a host or meta-language of the new language. On the other hand, the new language would inherit the infrastructure, as well as all the features and functionality of the new language, such that programs developed by the new language would be considered as first class objects of the host language, and with some familiarity with the host language, the domain language can easily be extended with new domain constructs and functionality. In this way, generic domain independent components and tools such as compilers do not have to be re-constructed, and thus, reduce drastically the cost and effort required to develop such languages.

Although the actual true concept of domain-specific embedded languages was first plainly established by Paul Hudak in (Hudak, *Building domain-specific embedded languages*, 1996) and (Hudak, 1998), notions of embedded languages were already used in 1958, when John Mc Carthy developed the Lisp programming language. To reason about the source code in the form of data structures, the language used the concept of lists as its main data structure. Later on, Lisp was found to be beneficial to allow programmers to easily create new syntax and embed "*little languages*" (Steele, 1990), for the macro system. Comparing the macro system created in Lisp with that in C, it is notable that macros in C are much more limited in functionality than those in Lisp. In fact, macros in C simply represent some syntax, which is later on substituted. Contrary to this, macros in Lisp can have far more functionality such that they can be used to transform the program's structure by using the full Lisp language to represent the transformation. To improve on the macros system in C, C++ templates were then designed to introduce type-safety and to make then applicable to a wider range of problems. While macros in C are restricted to single-line definitions, C++ templates can be defined over multiple lines, making it possible to apply the same template to more problems.

However, considering the methodology which Paul Hudak explains in (Hudak, *Building domain-specific embedded languages*, 1996) and (Hudak, 1998), one of the most notable advantages of embedded languages is the "*reuse of syntax, semantics, implementation code, software tools, as well as look-and-feel*" (Hudak, 1998). Keeping the same look-and-feel as that of the host language, reduces drastically the cost, time and the effort required for users to learn the new domain language. In (Hudak, *Building domain-specific embedded languages*, 1996), Hudak claims that one of the main differences between the Domain Specific Language (DSL) based methodology and the Domain Specific Embedded Language (DSEL) based methodology is the initial cost to develop the domain language. Once the initial costs are covered, then the costs to develop software using these languages are quite similar. In contrast to this is the cost of developing domain specific programs using a general-purpose high-level language. Although there is no significant initial cost, there is a significant difference in cost in the latter phases of the software development lifecycle.

In (Hudak, *Building domain-specific embedded languages*, 1996), Hudak adds to the advantages acquired by inheriting tools and the infrastructure of the host language, by claiming that, doing so, the designers of the domain-specific language are allowed to focus on the semantics of the language and the issues brought up while designing the semantics. It is important for the designers to establish an appropriate vocabulary of terms, which, besides precisely capturing the semantics of the application domain, it should also be easy for domain experts to use it and to create abstract representations of some structure or program specific to that particular domain. In this way, this vocabulary of terms should act as a library of the some primitive atomic components and combinators with which various models and programs, specific to that domain, can be constructed. Using various abstraction techniques, the language designers can then develop some highly modular, straightforward to evolve and easy to comprehend interpreters, which given a model constructed using these terms, they would be able to carry out various interpretations and analysis. In the case of functional languages, these interpreters are simply functions through pattern matching of terms and recursion would be able to handle complex models and interpret each term accordingly. Using this combinatorial approach, a model or program is merely a group of other less complex models combined in a certain way.

For example, assuming that a domain-specific language to define and interpret recipes such as Listing 2.10 (Figure 2.1 is a graphical representation of Listing 2.10) is required:

Apple Cake

- 200grams margarine
- 500grams flour
- 1 apple

1) Mix margarine with flour
2) Chop the apple
3) Mix the 2 together
4) Place in a dish
5) Cook in the oven for
   30mins at 180°C

Listing 2. 10

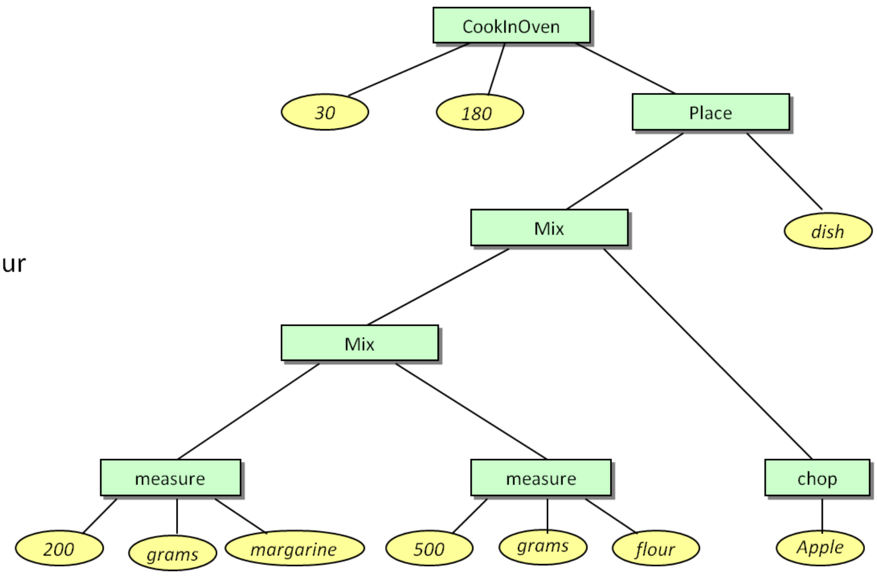


Figure 2. 1: A graphical representation of the recipe in Listing 2.10

A language embedded in the functional language, Haskell, can be developed, such that with the code in Listing 2.11, a domain expert could easily represent the recipe in Listing 2.10 (Figure 2.1).

```
appleCakeRecipe :: Recipe SemiSolid
appleCakeRecipe = let
                      apples = chop (measureEntire 2 apple)
                      rqrdFlour =  measureDry 500 g flour
                      rqrdMargarine = measureSemiSolid 250 g margarine
                      cakeMix = mix rqrdFlour rqrdMargarine
                      cakeWithApples = mix cakeMix apples
                      cakeInDish = place cakeWithApples dish
                  in
                      cookInOven 30 180 cakeInDish
```

Listing 2. 11: Recipe in Listing 2.10 and Figure 2.1 defined using the DSEL presented in (Micallef & Pace, 2007)

From Listing 2.11, it should be noted that the primitive components and combinators defined in the first two lists of Listing 2.12 should be provided in the vocabulary. Moreover, functions which can act as simple interpreters of the constructed descriptions, such as the ones in the third list of Listing 2.12, should be accessible to the user to be able to interpret and analyse the recipe.

Primitive Components:

- Texture Type
- Ingredients
- Utensils
- Measuring Units

Possible Combinators:

- Measure
- Chop
- Beat
- Mix
- Place
- Boil
- CookInOven

Additional Functionality:

- Display
- Time
- Shopping List
- Utensils List

Listing 2. 12: A list of the primitive components and possible combinators that can be provided in the language to define recipes such as Listing 2.10. The third list indicates the type of additional functionality that would be required after the recipe is defined

The full implementation of this simple language was presented as a case study for one of my Assigned Practical Task (APT) (Micallef & Pace, 2007).

This combinatorial approach is also explained in greater depth in (Jones, Eber, & Seward, 2000) where a library was created for the construction and interpretation of financial contracts. Combinators are also used extensively for hardware design.

The above example clearly illustrates, that even though domain experts might not have any particular programming skills, it is still relatively easy for them to use the language and reason about these abstract and modular definitions. This was also notable in the NSWC experiment (Hudak & Jones, 1994); although for some users it was the first time for them to use Haskell, they immediately grasped the concept. Some were also sceptic that the code was truly executable. Additionally, since the language constructs are provided in the form library, it is much easier to create a flexible language and to later extend the language with additional combinators, primitive components and functionality. This flexibility is in fact one of the factors that distinguishes combinator libraries from simple traditional libraries.

Since a general-purpose language is used as the host language, programmers still have access to the less restrictive programming features, as the case with normal domain-specific languages. This means that at any time advanced users who are familiar with the host language, can refer to the functionality of the host language and incorporate these more general features to programs written using the domain-specific language. While comparing domain-specific languages to domain-specific embedded languages, Kamin in (Kamin, 1998) claims that traditional DSLs are usually poorly designed and are not so flexible and easy to extended. In most of the cases, additional features and functionality are added in an ad-hoc manner, thus making the language less maintainable. In contrast, DSELs have access to the general programming features of the host language and thus extension are made to the domain language more elegantly.

Besides designing the domain semantics and developing a library of domain specific terms, language designer can also develop their own domain-specific type system. This would guarantee that models defined through a set of components and primitives combinators are type-safe and thus of the required quality. Users are thus enforced to the use terms in the vocabulary appropriate according to the domain semantics. In languages such as Haskell, the new type system would also be embedded in that of the host.

Hudak also identifies the possibility of using features from different DSELs to achieve the required behaviour. If the DSELs are embedded in the same host language and thus use the same compiler (that of the host language), then constructs from the different languages can be used in the source code, especially when domains overlap. Following all the above mentioned advantages, "*the DSEL approach is highly amenable to formal methods*" (Hudak, *Building domain-specific embedded languages*, 1996). Since most of the reasoning is carried out within the domain semantics and not within the semantics of the programming language, several properties of the DSEL can easily be proven. This advantage was mostly notable in the NSWC experiment (which shall be discussed later on) (Hudak & Jones, 1994). While it was relatively straightforward to prove properties of the DSEL, it was rather difficult if not impossible to prove the same properties in other competing designs.

In (Kamin, 1998), Kamin gives an extensive overview of all the advantages and disadvantages of DSELs in contrast to DSLs, and among the drawbacks, he mentions the non-optimal syntax of DSELs. He argues that for the DSEL to keep abreast with the syntax of the host language, then the language has to be more verbose than is usually the case with DSLs which are developed from scratch. Moreover, since the compiler of the host language is used for the DSEL, error messages are not really helpful to the DSEL users. They are more directed to the host language users. For this reason, a specialized user-friendly environment should ideally be created for the DSELs, to interpret the error messages generated by the host language compiler, in a friendly manner which would be more helpful for the DSEL users. Another disadvantage is the inability to perform domain-specific optimisations and transformations. Thus, considering the use of DSELs for program generators, it is not possible to guarantee the syntactic correctness of generated programs and to type-check the programs before the actual code is generated.

It is a known fact that solutions developed using DSELs hosted in functional rather than imperative languages, tend to carry out a variety of redundant computation which is specific domains such as computer graphics, might result in performance degradation. To avoid such an issue, in (Elliott, Finne, & Moor, 2000) the embedding of an optimized compiler is proposed such that DSEL constructs would be compiled into some optimized imperative code. Due to the embedding, designers are also constrained by the syntax, type system and limitations of the host, and thus, it is also essential for the designers to attentively choose an appropriate host language.

2.4.1. Choosing an appropriate host language

Since the language designers of the DSEL are constrained by the features and limitations of the host language, then it is of utmost important for the designers to scrutinize the features and functionalities offered by various languages and programming paradigms and choose the one which is most appropriate for the required domain.

Having a look at some of the successful DSELs which have been developed during the past years, it is notable that functional languages are usually given preference and chosen as the host language. Some of the features which are usually of interest to DSEL designers include pattern matching, higher-order programming where functions are considered as first class objects in the language and parametric polymorphism. These are actually the key features that make the design elegant and modular, and which provide the appropriate abstraction to allow users to focus on the behaviour of the domain specific system rather than the implementation. Following the publication of Backus' Turing Award lecture (Backus, 1978), it became evident that due to the inherent abstraction of the functional programming paradigm, descriptions of programs using such languages tend to be concise and easy to comprehend, thus concluding that such languages can act as the right platform to describe complex system. In this way, such languages provide the ideal abstraction, which according to Hudak, is one of the most important characteristic of a DSEL to help users write good quality software. After the publication of Backus' paper, a dramatic increase in the amount of funding invested for further research in this area was noted. This led to the application of such a paradigm to various interesting domains, such as hardware design where basic hardware components started to be defined in the form of functions.

Moreover, designers usually aspire for a host language that has a powerful type system, which ideally handled ad-hoc polymorphism effectively. This would help them to design and embed their own type system to type-check in certain cases at compile-time to ensure the appropriate use of the provided constructs in the DSEL and thus, the construction of high quality models or programs. Thus statically and strongly typed language which provides an appropriate type class system, such as Haskell would help to provide this feature. It is important to note the Haskell's type system is both sound and complete, meaning that while the soundness of the system ensures that function are only used with the appropriate typed data, completeness guarantees flexibility such that user are not limited in the ways they can make use of provided functions. To obtain the benefits discussed in the first paragraphs of Section 2.4, laziness is another desired feature which designers usually seek in a language and which are usually available in purely functional language, such as Haskell. For this reason, various DSELs have been embedded in Haskell (Jones S. P., 2003). Some of these languages shall be discussed in the following section.

2.4.2. Domain Specific Languages Embedded in Haskell

Various successful DSLs have been embedded in Haskell for several domains. Some of these languages are discussed in the following sections.

Hardware Description Languages

After the success of functional hardware description languages such as μFP (Sheeran, 1984) and Ruby (Sheeran, 1990), other similar languages embedded in Haskell were developed. One of the first languages was Hydra (O'Donnell, 1996). This language, which was previously embedded in the functional languages Daisy, ML and Scheme, is currently being used to teach computer architecture at undergraduate level. Although simulation is one of its strengths, it uses a simple tagging technique known as explicit naming (a technique discussed in Section 2.4.3.3) to annotate circuit descriptions before generating netlists.

Following Hydra, Hawk (Cook, Launchbury, & Matthews, 1998) was developed to simulate and model microprocessor architectures at both structural and behaviour level.

One of the most complete and successful hardware description languages, which provides various features, is Lava (Claessen, 2001). Two of the basic features include a library of basic components and combinators to model circuits and a variety of connections patterns (i.e. functions which given some circuit descriptions as input arguments are capable of combining them in some way, such that a new possibly composite or complex circuit is returned) for more concise descriptions and to further abstract away the description of certain complex circuits such as the serial composition of circuits or the composition of a row of identical circuits. Once the circuit is modelled, automatic synthesis to VHDL, automatic testing using QuickCheck (Claessen & Hughes, 2000) and automatic verification of properties through the use of observers and model checking tool can be carried out. This hardware description language which is implemented as a library, makes extensive use of most of the features which it inherits from Haskell, example, functions to represent descriptions concisely, abstract data types for the basic constructs of descriptions, recursion and pattern matching to define the behaviour of certain circuits, polymorphism and parametric types for more general descriptions, Haskell type system in which Lava's type system was embedded. Such features make Lava a very elegant, modular and flexible language which can easily be extended with additional functionality and thus allow other possible interpretations of circuits defined using this language.

One of the limitations with languages such as Lava is the inability to incorporate non-functional information about the circuits, such as power consumption and delay, due to the functional description of circuits. To handle this issue, component-based languages such as Wired (Axelsson, Claessen, & Sheeran, 2005) were developed. By treating wires as first-class objects and by modelling circuits as relational blocks, both structural and layout information are included in the circuit description. This contrasts with the approach taken in Lava since in Lava the circuits are modelled as functional rather than relational blocks. A similar language, HeDLa (Pace, 2007) was developed as an attempt to obtain a balance between the component-based approach adopted in Wired and the functional approach employed in Lava. This balance was achieved by explicitly defining connections description as parameters in the actual function which describes the required circuit.

Geometric Region Analysis

In 1994, an interesting experiment conducted by Arpa, ONR and the Naval Surface Warfare Centre (NSWC) was carried out to identify the suitability of certain languages to prototype a real-world application, in this case, a geometric region server, as a component of the AEGIS Weapons System, which at that time was being redesigned by the NSWC. Programs in various conventional programming languages and another with the functional programming language Haskell, were developed and compared. These programs and a number of development metrics were reviewed by a committee chosen by the Navy and the results of the experiment were later published in (Hudak & Jones, 1994).

Besides demonstrating that developing a DSEL in Haskell significantly reduces the development time, results also confirmed that specifications defined using such a DSEL were much more concise and easier to comprehend. Moreover, although additional functionality and enhancements were carried out on the languages and three different versions of the system were developed, still the non-trivial changes were yet incorporated with great ease. Features such as modularity and abstractions played an important role to make such a system maintainable and evolvable with the least amount of effort and cost. The ease to carry out some formal methods on the system was also noted.

Animation

Another interesting domain is animation. A DSEL embedded in Haskell called Functional Reactive Animation (Fran) (Elliott & Hudak, 1997) was intentionally developed to bridge the gap between the specification of the required animation and how such an animation should be presented on a computer. To achieve this goal a declarative model-oriented approach was proposed as an alternative to the conventional presentation-oriented approach. The language provides a collection of data types and functions for users to define highly composable animation descriptions easily. The behaviour of the required animation consisting of interactive multimedia is defined through the use of time-varying, reactive values. In this way, such a language would allow users to focus on the actual animation and allow the system to handle the actual represent of the defined animation on the computer. This language also served as the foundation for Functional Reactive Programming (Nilsson, Courtney, & Peterson, 2002), which introduces the notion of time flow to the host language.

Music Composition

To help users describe abstract musical concepts using some high-level declarative constructs, the DSEL Haskore (Hudak, *Haskore Music Tutorial*, 1996), was developed as a collection of Haskell modules, with the main objective of producing an algebra of music. The basic features of the language include a set of primitive notions to represent musical objects such as notes and rests, a number of operations to transform these objects such as transpose and tempo-scaling and other operations to combine these objects into more complex compositions. Considering Haskell as "*an executable specification language*", Haskore objects represent both abstract musical concepts as well as their implementation, such that if some property is proven for some object, then that property should hold for both the abstract musical notion as well as its concrete implementation.

Financial Contracts

To address issues encountered when handling complex financial contacts, a DSEL combinator library in Haskell is presented in (Jones, Eber, & Seward, 2000). The main objective of such a library is to possible a set of primitive combinators which can be used to construct any new legitimate contract. Moreover, the language also defines a number of functionalities which can be carried out systematically on any type of contract constructed using this languages. Thus, having an abstract representation of the contract, constructed using primitive combinators defined in the provided library, various computations and processes can be defined once and later carried out on any type of contract. The paper also proposes ways how an abstract valuation semantics can be applied to the combinators, to find the value of the contract. After the success of this DSEL, the financial industry such as the global Credit Suisse Groups decided to adopt the language. The authors also claim that although Haskell's laziness is an important feature during the evaluation of such contract, yet the true reason why Haskell was chosen is because it is declarative.

SQL Queries

Since queries are usually communicated to databases in the form of an unstructured string, it is not really possible to check the correctness of such SQL queries before passing it to the database. Thus, to ensure that such queries syntactically correct and correctly typed, a DSEL was proposed in (Leijen & Meijer, 1999). The language also provides additional functionality such as the interpretation of abstract SQL query representations into more concrete target syntax, specific to the invoked database.

Besides describing the proposed language, they clearly define techniques how terms should be embedded and explain the importance of developing a domain-specific type system to type check the constructions produced by the users.

Geometric Constructions

In (Grima & Pace, 2007), a DSEL is presented as a teaching tool to help students define geometric constructions based on compass and straight-edges in an algorithmic and to later reason about such constructions. The language also provides additional functionality to allow users to test and verify properties of constructions such as equivalence of constructions and equality of angles.

Another interesting DSEL was developed to construct and analyse Origami models (Caruana & Pace, 2007).

2.4.3. Techniques in Embedded Languages

As DSEL gained popularity over the years, various techniques to facilitate and enhance the design and implementation of such languages were developed and proposed. Some of these techniques include: the notion of shallow and deep embedded languages, higher-order typing, the concept behind observable sharing and block tagging and the use of connection patterns and parameterized blocks. Phantom types are considered a foundational characteristic of any embedded language (Hinze, 2003) and are used in most of the cases to define the type system of the new embedded language. For polymorphic operations then type classes should be defined (Wadler, Hudak, Hughes, & Jones, 2007). If infinite data structures need to be handled, lazy evaluation (discussed in Section 2.2.3) would be essential. If shared nodes and loops have to be detected, then the language designer must consider some of the proposed solutions such as the use of non-updateable references (Claessen & Sands, 1999) (Claessen, 2001)). These techniques have been used for our language and shall be discussed in the following sections.

2.4.3.1. Shallow vs. Deep Embedding

When designing a domain-specific embedded language, one of the first decisions that a designer must take is whether to implement the language using a shallow or a deep embedded approach. The choice usually depends on the objectives of the language and the type of operations and analysis that shall be carried out on the descriptions defined using the language constructs. To illustrate the differences between these two approaches, the following example shall be considered.

Assume that a simple language to construct expressions is required. The operations that this language should support are the `plus`, the `and` and the `if..then..else`. These shall act as combinators with which expressions should be defined. Thus, these combinators can be implemented as in Listing 2.13:

```
(.+.) :: Int -> Int -> Int            -- Plus Operator
(.+.)   = (+)                         --      e.g.:  2 .+. 3 = 5

(.&&.) :: Bool -> Bool -> Bool        -- And Operator
(.&&.)  = (&&)                        --      e.g.:  True .&&. False = False

(.|>.) :: Bool -> (a,a)               -- If..Then..Else
 c .|>. (vT, vF) = if c then vT       --      e.g.:  False .|>. (2,3) = 2
                        else vF       --      e.g.:  True .|>. (True, False) = True
```

Listing 2. 13: Combinators in a simple shallow embedded language to define expressions

It should be noted (in Listing 2.13) that the operators that are defined within the language (`(.+.)`, `(.&&.)`, `(.|>.)`) refer to the operators defined in the host language (`(+)`, `(&&)`, `if..then..else`). These can be used as illustrated in the examples on the right.

Thus, if the following expression (in Listing 2.14) had to be defined

```
expr c x = c .|>. (x .+. 3, 6)
```

Listing 2. 14: Expression defined using the language in Listing 2.13

and later on invoked with a boolean value, to represent the boolean condition, and a value for the argument `x`, then just a single integer value would be returned. This means that all the information about the intermediate evaluation steps would be lost; for instance, the value of the boolean condition, when a boolean expression rather than a value is passed on as input, and the `plus` operator, when the condition is evaluated to true. Thus, using such an approach, the internal structure of the expression cannot be analysed and it is not possible to investigate how and why such a value was returned.

This approach is known as *shallow embedding*.

The same embedded language can be implemented in the following manner (listings 2.22 and 2.23):

```
data PrimExpr =  ConstI   Int
               | Plus     PrimExpr PrimExpr
               | ConstB   Bool
               | And      PrimExpr PrimExpr
               | IfThenElse PrimExpr (PrimExpr, PrimExpr)
```

Listing 2. 15: `PrimExpr` data type to define the primitive constructs of a simple deeply embedded language to define expressions

```
constI :: Int -> PrimExpr
constI x = ConstI x

(.+.) :: PrimExpr -> PrimExpr -> PrimExpr
 x .+. y = Plus x y

constB :: Bool -> PrimExpr
constB x = ConstB x

(.&&.) :: PrimExpr -> PrimExpr -> PrimExpr
 x .&&. y = And x y

(.|>.) :: PrimExpr -> PrimExpr -> PrimExpr
 c .|>. (x,y) = IfThenElse c (x, y)
```

Listing 2. 16: Combinators accessible to the user and defined using the constructs in Listing 2.15

Several differences can be noted between the two implementations that is Listing 2.13 and listings 2.15, 2.16. Primarily, a new abstract data type is defined (Listing 2.15). The defined constructors of this abstract data type represent the primitive components (`ConstI` (for integer constants) and `ConstB` (for boolean constants)) and combinators (`Plus`, `And`, `IfThenElse`) of the language. Besides this data type, a function is defined for every single constructor inside the data type (Listing 2.16). By defining such functions or rather combinators, users would be restricted to use these functions to refer to the language's primitive components and combinators, rather than the constructors within the data type. In this way, by defining appropriate type signatures for each of these functions, static compile-time checks would be carried out to ensure that expressions are correctly typed. In fact, the combinators which are defined in Listing 2.16 are not type safe since the type of both integer and boolean expressions is set to `PrimExpr`. Appropriate type signatures are defined in Section 2.4.3.2 where type embedding is discussed.

Thus, if an expression similar to that in Listing 2.14, had to be defined, example:

```
expr c x = c .|>. (x .+. (constI 3), constI 6)
```

and later on invoked with the following arguments

```
expr (constB True) (constI 2)
```

then the expression is not evaluated, but instead an abstract representation made up the internal primitive constructors of the abstract data type `PrimExpr` would be defined as follows

```
expr(constB True)(constI 2) = IfThenElse (ConstB True)
                                  (Plus(ConstI 2)(ConstI 3),ConstI 6)
```

With such an abstact representation of the defined expression, further analysis and operations can be carried out. Example: `eval` to evaluate the expression or `noOfOps` to return the number of operations in the expression (Listing 2.17).

```
noOfOps :: PrimExpr -> Int

noOfOps (ConstI _) = 0
noOfOps (Plus x y) = 1 + noOfOps x + noOfOps y
noOfOps (ConstB _) = 0
noOfOps (And x y)  = 1+ noOfOps x + noOfOps y
noOfOps (IfThenElse c (x,y)) = 1 + noOfOps c + noOfOps x + noOfOps y
```

Listing 2. 17: Function `noOfOps` to identify the number of operations carried in an expression defined with the language in Listing 2.15 and 2.16 (note that all constructs except for `ConstI` and `ConstB` are considered as operations)

s.t. `expr(constB True)(constI 2) = 2` *(operations* `IfThenElse` *and* `Plus`*)*

This approach is known as *deep embedding*

The implementation for shallow embedding is much concise than that for deep embedding. However, a significant advantage in deep embedding is the possibility to first define the required structure and then carry out any desired interpretation or analysis. Functions which perform such evaluation, need to be defined once and then can be used for any structure expressed using the language.

The choice between the two approaches depends on the domain and the objectives of the language. For instance, while the hardware description language Hawk (Cook, Launchbury, & Matthews, 1998) was implemented using the shallow embedded approach, the hardware description language Lava (Claessen, 2001) and the DSEL for financial contracts (Jones, Eber, & Seward, 2000) opted for a deep embedding approach. The two approaches are contrasted and discussed in more detail in (Wildmoser & Nipkow, 2004).

2.4.3.2. Type Embedding and Phantom Types

One of the most important aspects of Haskell (Jones S. P., 2003) is its powerful type system which is both sound and complete and its ability to carry out static compile time checks (as discussed in Section 2.2.6). Similarly, a crucial feature of a DSEL is its type system. The embedded type system should be sound such that users are restricted to use functions only in a type safe and correct manner, and complete such that the system is flexible and the users are not limited in the ways these functions can be used.

The importance of such a type system was emphasised in (Rhiger, 2003), where phantom types are proposed as a solution to construct an effective type system. In this paper, phantom types are considered as the foundation of embedded languages. These types are essentially parameterized polymorphic data types, usually consisting of just one constructor, whose instances are independent of the type variables. This means that although some type variables might appear on the left hand side of the data type declaration, this variable does not appear on the right hand side of the declaration. Thus the solely purpose of such type variables is to express a type constraint.

For instance, consider the deeply embedded DSEL which was defined in the previous section in Listing 2.15. The abstract data type `PrimExpr` was defined to ensure the syntactic correctness of expressions and to allow users to define any expression in terms of these constructs. Since it would be more convenient for the user to use combinators in the form of functions rather than using the actual constructors in the data type, then the functions in Listing 2.16 were defined. However, the main problem with these combinators is that incorrectly typed expressions, such as the following, can still be produced:

```
(constI 2) .&&. (constI 3)
```

To prevent this, the following abstract data type which uses phantom types, can be defined

```
data Expr a = Expr PrimExpr
```

To ensure the construction of syntactically and correctly typed expressions, the combinators defined in Listing 2.16 can be expressed in terms of the data type `Expr a`, such that the type of the functions would be defined as follows (Listing 2.18):

```
constI :: Int -> Expr Int
(.+.) :: Expr Int -> Expr Int -> Expr Int
constB :: Bool -> Expr Bool
(.&&.) :: Expr Bool -> Expr Bool -> Expr Bool
(.|>.) :: Expr Bool -> Expr a -> Expr a
```

Listing 2. 18: Type safe combinators, for constructs in Listing 2.15, defined in terms of the data type `Expr a`

where for instance,

```
(.&&.) :: Expr Bool -> Expr Bool -> Expr Bool
(Expr x) .&&. (Expr y) = Expr (And x y)
```

In this way, only the combinators have access to the unsafe primitive constructors of the data type `PrimExpr`. At the level of the primitive constructors, the types are abstracted away such that during the production of interpreters, the programmer would only need to consider the constructors in the main data type. Thus, if the previous example

```
(constI 2) .&&. (constI 3)
```

had to be executed in the new language, Haskell's type checker would generate an error; while the type of `(constI 2)` and (`constI 3`) is `Expr Int`, the type which `.&&.` expects is `Expr Bool`.

A variety of applications of these phantoms types are discussed in (Hinze, 2003). It also explains how type classes can be replaced with phantom types. As noted in (Rhiger, 2003), such types are also important to reason about the program correctness. In fact they are used in most DSELs to impose stricter constraints on the type system; example, the DSEL in (Leijen & Meijer, 1999), defined to ensure the syntactic correctness and type safety of SQL queries, and the hardware description language, Lava (Claessen, 2001).

One limitation of such types, is that, if type-safe reconstructions are required then additional tagging and run-time checks would be necessary. To address this problem, a technique suggested in (Cheney & Hinze, 2003), can be used where type constraints are expressed through the use of type equations.

2.4.3.3. Sharing and Loops

As discussed in Section 2.4.3.1, the main advantage of deep embedding is that once the required structure is defined, internally an abstract representation in terms of the primitive data constructors is composed. This can then be passed on as an input to some other function, such that, the required analysis, interpretation and computation would be carried out on that structure. Although deeply embedded languages are rather flexible and extensible, issues are encountered when structures are shared or when they contain loops.

For instance, considering the simple embedded language which was defined and developed in the previous two sections, then the expression defined in Listing 2.19 is a valid expression in the language.

```
z = let
        y = (constI 2) .+. (constI 3)
    in  y .+. y
```

Listing 2. 19: `z` - shares sub-expression `y`

If `z` had to be evaluated using a shallow embedded approach, due to referential transparency, the compiler has a choice how to evaluate it. Since referential transparency in functional languages is simply the result of lambda beta reduction, all the occurrences of `y` in the expression, which defines the value of `z`, are replaced by the actual expression or the value of `y`. This leads to the evaluation of an expression which has no particular user (the programmer) visible sharing.

Considering now a deeply embedded approach, when `z` is evaluated, rather than computing the value of the expression, an abstract finite representation is constructed in terms of the primitive constructors defined in `PrimExpr` (Listing 2.15). Thus, the tree in Figure 2.2 would be generated for expression `z`.

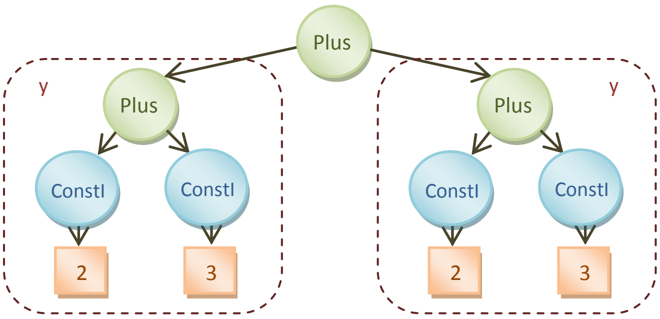


Figure 2. 2: The abstract representation (in terms of the constructors in `PrimExpr`) for expression `z` (Listing 2.19)

Although the abstract tree structure in Figure 2.2 perfectly illustrates the required behaviour, the shared sub-expression `y` is defined twice. This means that, when analysis is carried out on this representation, `y` would be evaluated twice. For instance, if a function was defined to return the number of operations that are essential to evaluate the expression, 3 rather than 2 would be returned. To avoid this, evaluation should ideally be carried out on structures such as that in Figure 2.3.

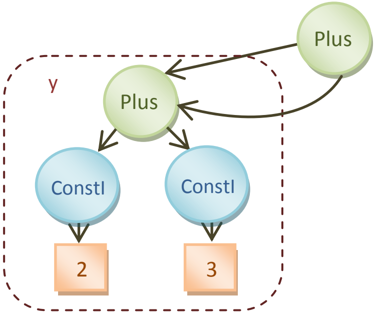


Figure 2. 3: The ideal abstract tree structure to represent expressions with shared sub-expressions

A more problematic type of structure is one which contains loops. Consider expression `x` in Listing 2.20.

```
x = x + 2
```

Listing 2. 20: Expression `x` contains a loop

Assuming that the language used to represent the expression is deeply embedded, then when `x` is evaluated, the Haskell compiler expands `x` until it runs out of stack space. This is caused by the cyclic dependency of the expression which does not have any terminating condition. This means that the internal abstract representation is actually an infinite tree structure.

Thus to avoid such situations, a language designer would either have to adopt a shallow embedded approach or else employ a technique whereby shared structures and loops would easily be detected and handled appropriately during analysis and evaluation of the structure. Due to the advantages of deep embedding, over the years, different techniques were proposed to detect sharing and loops. These are discussed in the following sections.

Explicit Tagging

One of the first and simplest solutions, which was first implemented in the hardware description language Hydra (O'Donnell, 1993), is the explicit tagging of every component and node in the structure with a name explicitly defined by the user. Thus, during evaluation, a list of these nodes is maintained such that before evaluating a node, a check is carried out to verify whether that node has already been evaluated. If so, then the node is not re-evaluated. The name, which is explicitly assigned to every node, acts as an easy way to reference nodes in the structure. This would also be helpful for debugging and for describing the entire structure. Although this solution is rather trivial, it increases the possibility of errors and bugs in the definitions. This is due to the fact that it is the responsibility of the user to provide correct names to the nodes. Unfortunately, these names can easily be misspelt or repeated.

To adopt this approach, every combinator in Listing 2.18 and every constructor in `PrimExpr` (Listing 2.15) must have an additional `String` argument to represent the user-defined *tag*, such that, the `(.&&.)` operator, for instance, would be defined as a function as illustrated in Listing 2.21:

```
and :: Expr Bool -> Expr Bool -> String -> Expr Bool
and (Expr x)(Expr y) t = Expr (And x y t)
```

Listing 2. 21: Cominbator `and` (instead of operator `(.&&.)`) with an explicit tag `t`

Monadic State

To elevate the problems of explicit tagging, state monads (discussed in Section 2.2.7) were used in the first implementation of Lava (Bjesse, Claessen, Sheeran, & Singh, 1998). Since purely functional languages have no side-effects, then the only way to store data, while some computation is carried out, is to use state monads. Thus, the actual process of tagging the nodes is abstracted away from the user and handled automatically. The nodes are automatically tagged by the next unique identifier, usually a positive numerical value, which would be kept in the *State* monad. Although this technique eliminates most of the errors, introduced with explicit tagging, the style of programming that would have to be used, would no longer be that intuitive and straightforward. This would make the code less readable and thus such a technique is not really considered feasible to use.

To employ this approach, every constructor in `PrimExpr` (Listing 2.15) must have an additional `Integer` argument to represent the tag assigned to the structure. Since this tag is the next unique identifier stored in a state monad, then the combinators would have to be defined in a similar way as illustrated in Listing 2.22.

```
(.&&.) :: Expr Bool -> Expr Bool -> State Int (Expr Bool)

(Expr x) .&&. (Expr y) = do
                           modify (+1)
                           nextT <- get
                           return Expr (And be1 be2 nextT)
```

Listing 2. 22: Combinator `(.&&.)` using monadic state

In Listing 2.22, the *do-notation* is simply a syntactic sugar for pure programs that use the `bind` operator (defined in 2.2.7). `State Tag (Expr Bool)` indicates that the state shall store integer values and return an expression of type `(Expr Bool)`. `modify`[1] changes the value of the state; in this case, it increments the value by 1 to obtain the next unique identifier to tag the structure. `get`[1] is used to get the value of the state. This is then passed on to the constructor `And`.

Non-Updateable References

Another approach, which is currently considered the most practical, was implemented in the latest version of Lava (Claessen, 2001) by Claessen and Sands. (This solution was first proposed in (Claessen & Sands, 1999) and then extended and used for the implementation of Lava in (Claessen, 2001)). The technique they developed uses non-updateable references, in a similar way to pointers in C, to detect graph sharing and loops. In this way, every new constructed object is referenced. Reference equality is then carried out to check whether such an object has already been evaluated. All this is done in a manner transparent to the user.

Since new language constructs are introduced, this technique is considered to be non-conservative. However, by importing the data type and functionality provided in a module called `Ref` (Listing 2.23), it is rather simple to create the objects, reference and dereference them and check reference equality.

```
type Ref a = …
ref :: a -> Ref a
deref :: Ref a -> a
(<=>) :: Ref a -> Ref a -> Bool
```

Listing 2. 23: Data type `Ref` and other functions defined in the module `Ref` (Claessen & Sands, 1999)

[1] `modify` and `get` are pre-defined function in Control.Monad.State (http://haskell.org/ghc/docs/latest/html/libraries/mtl/Control-Monad-State-Class.html#v%3Amodify)

The above functions can be used as follows (Claessen & Sands, 1999):

```
(1)  > let x = undefined in let r = ref x in r <=> r        True
(2)  > let x = undefined in ref x <=> ref x                 False
```

Listing 2. 24: Using functions in Listing 2.23

The result in the first case (1) is `True` since only one instance is created and this is compared with itself. Although in the second case (2) the two created instances refer to the same variable, still two instances are created, yielding different references and thus, no detection of sharing.

Thus, to adopt this approach to the DSL defined for expressions, the following data type should be defined

```
newtype RefPrimExpr = RefPrimExpr (Ref PrimExpr)
```

and the data type `Expr a` should be changed to

```
newtype Expr a = Expr RefPrimExpr
```

Moreover, arguments of constructors in `PrimExpr` must not be of type `PrimExpr` but `RefPrimExpr`. In this way, the combinators would have to be defined as illustrated in Listing 2.25 for the `(.&&.)` operator.

```
(.&&.):: Expr Bool -> Expr Bool -> Expr Bool
(Expr x) .&&. (Expr y) =  Expr (ref (And x y))
```

Listing 2. 25: Combinator `(.&&.)` defined using non-updateable references

With the introduction of such new language constructs, referential transparency in no longer supported. Since this technique is based on the unsafe function `unsafePerformIO`, side-effects could easily be introduced. This was also noted in (O'Donnell, 1993). However to limit this impact, references in the latest version of Lava (Claessen, 2001) are enforced to be read-only and thus non-updateable. To avoid the need to carry out changes in the language compiler, the original semantic properties of the language, that is Haskell, are maintained as much as possible, by thus assuming that the compiler in use automatically evaluates shared expressions only once.

2.4.3.4. Block Tagging

One of the limitations of defining domain programs or descriptions, using a functional approach, is the inability of identifying blocks of components, which although easily visible by the user at the description level, in the abstract syntax tree made up of the primitive constructors which is constructed for every new definition, the notion of blocks is lost. Thus the entire internal structure, over which evaluation is carried out, is simply viewed as a collection of connected primitive components.

This issue is encountered frequently in the area of hardware design. As discussed in Section 2.4.2, various HDLs were implemented over the years. While some adopted a functional approach such as Lava (Claessen, 2001), others such as Wired (Axelsson, Claessen, & Sheeran, 2005) employ a component based approach. In the former case the use of appropriate typed functions, automatically induce gate connections and thus end up with the issue mentioned in the first paragraph. In the latter case, in particular Wired, circuits are considered as relational blocks which allow various non-functional aspects to be introduced to the descriptions. Similar to this is HeDLa (Pace, 2007) which tries to bridge the two approaches and allow functions to include explicitly non-functional information.

A simpler approach is to explicitly tag blocks (used in (Caruana & Pace, 2007)). Thus, by allowing the user to explicitly assign a name (or some other properties) to a particular definition, during evaluation the internal structures might or might not be analysed depending upon the interpreter. The main drawback of this technique is that besides being tedious, it is also the source for the introduction of new errors; users might misspell or repeat names.

To adopt this technique then another constructor would have to be added to `PrimExpr`:

```
data PrimExpr =  … | ExprBlock String PrimExpr PrimExpr| …
```

s.t.
```
exprBlock nm e x = Expr (ExprBlock nm (e x) x)
```

`exprBlock` defines an expression `e` with input `x` as a single block named `nm`. To investigate the internal structure during analysis, then the second argument of `ExprBlock` should be analysed. Else, to consider it as one element, then the third argument (i.e. the input of the block) should be investigated.

2.4.3.5. Connection Patterns

In (Sheeran, 2005), Sheeran discusses why she thinks hardware design and functional programming are a '*a perfect match*' and besides arguing that circuits are like functions and can easily be expressed in a concise and readable manner, she also illustrates how powerful connection patterns can be to define circuit generators. In fact, such patterns played an important role in functional hardware description languages to ensure the appropriate abstraction and modularity and allow re-use of circuit descriptions.

Such connection patterns can easily be defined for any language which uses a functional approach. Since functions in such languages are considered first class objects, then higher-order functions can be used to encode these patterns. These are merely functions which given other functions as input, are capable of combining them in some way, such that a new possibly composite or complex function is returned.

Thus considering the language defined in the previous sections to handle simple expressions, for a composite expression to be defined, then the functional composition operator `(.)` defined in Haskell can be used as illustrated below, where `g . f` is a composite function which first executes function `f` and then `g`.

```
g . f        where        f x = x .+. 3        g x = x .+. 4
```

However, if the same function is defined as `f ->- g` it would be more readable as the precise order of execution would be depicted. The operator `(->-)` would simply be defined as

```
expr1 ->- expr2  =  expr2  . expr1
```

If, on the other hand, a list of expressions should be composed in series, then the connection pattern `compose` can be defined as follows by using the previously defined series composition pattern `(->-)`:

```
compose [] = id
compose (expr : exprs) = expr ->- compose exprs
```

Listing 2. 26: Implementing connection pattern `compose`

Connection patterns can abstract away implementation details and thus allow the user to focus more on what behaviour is required rather than how such behaviour should be implemented. Such patterns also make definitions concise and easier to comprehend. For instance, let's assume that our simple language handling expressions provides a connection pattern named `row`. The main purpose of this pattern is to define a linear array of instances of a particular two input and two output expression, whereby one of the inputs of an instance of the expression depends upon one of the outputs of a previous instance, as illustrated in Figure 2.4.

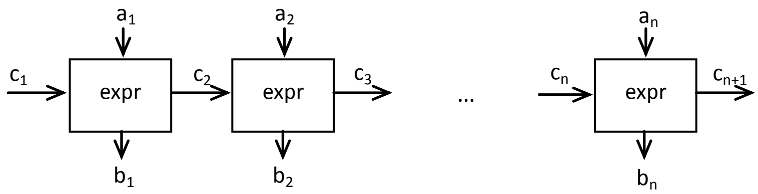


Figure 2. 4: The connection pattern `row`, connecting `n` instances of `expr`

Assuming that the `a` inputs (illustrated in Figure 2.4) are provided as an input list together with the input c_1 which is passed on to the first instance to `expr`, the `row` pattern would be defined as shown in Listing 2.27. This would then return a list of `b` outputs together with the final `c` output produced by the *nth* instance of `expr`.

```
row expr (ci, [])    =  ([], ci)
row expr (ci, a:as) =  (b:bs, co)
                          where (b, c)    = expr (ci,a)
                                (bs, co)  = row expr (c,as)
```

Listing 2. 27: Implementing connection pattern `row`

Considering now the division operation of an n-digit number by a value for instance 2, it should be noted that an operation such as `one_digit_div2` needs to be computed for every single digit (Listing 2.28).

```
one_digit_div2  prev_rem d = let
                                 divend = (prev_rem .x. (constI 10)) .+. d
                                 quot = divend .div. (constI 2)
                                 rem  = divend .mod. (constI 2)
                             in
                                 (quot, rem)
```

Listing 2. 28: Expression `one_digit_div2 prev_rem` to return the quotient and remainder when a single digit is divided by 2

Once the current dividend (`dividend`) is calculated in terms of the previous remainder (`prev_rem`) and the current digit (`v`), the quotient (`quot`) and the remainder (`rem`) are calculated and produced as output. In Listing 2.28, it is assumed that our language defining expressions provides operators `(.x.)` for multiplication, `(.div.)` to calculate the quotient and `(.mod.)` to obtain the remainder. Using `one_digit_div2` and the connection pattern `row`, an n-digit division by 2 operator can be defined as follows (Listing 2.29)

```
ndigit_div2 ds = row one_digit_div2 ((constI 0), ds)
```

Listing 2. 29: An n-digit division by 2 operation, defined in terms of the connection pattern **`row`**

whereby `ds` represents the n-digit value as a list of digits.

As illustrated in these examples, by adopting a connection pattern oriented approach, definitions in the language are more readable, concise and easier to comprehend. Moreover, such patterns provide the right abstraction for the user to focus on the required behaviour rather than the implementation of such operations.

2.4.3.6. Parameterized Blocks

Since definitions in languages embedded in a functional host are essentially functions, it is possible for the user to define a definition with parameters, such that, by providing the value of these arguments, the required definition or structure is produced. In this way, the user is able to reuse the same definition to produce different but similar blocks. These are known as *parameterized blocks* and are used in most functional embedded languages such as hardware description languages, for instance Lava (Claessen, 2001), to provide the right abstraction for the user to focus on the required behaviour rather than the implementation and to produce concise and easier to comprehend definitions.

Considering the language which was previously defined to handle expressions, it should be noted that the operation defined in Listing 2.29 is essentially a parameterized block. Given the n-digit value that should be divided by 2 as a list of digits, the resulting list of quotients and the final remainder, are produced as output. If such an operation needs to be generalized to carry out division operations with other divisors, then the following parameterized block can be defined:

```
ndigit_div one_digit_div ds = row one_digit_div ((constI 0), ds)
```

Thus, besides the digits, an expression to define the operation that should be carried out on each digit is passed on as another input argument.

It might also be convenient for the user to define the following parameterized block such that `n` copies of a given expression are composed together in series. This is defined in terms of the connection pattern `compose` (defined earlier in Listing 2.26) and the function `replicate`, provided by the host language, Haskell.

```
composeN n expr = compose (replicate n expr)
```

Other blocks which are frequently used can easily be defined by the user himself in this same manner. In this way, blocks can be defined once and used to construct specific structures, depending upon the input arguments.

2.5. Conclusion

A brief overview of the functional programming paradigm was discussed in the first part of the chapter. Features of the pure functional programming language, Haskell (which was chosen as the host for our embedded language), were addressed in great depth. Following this, the objectives, advantages and disadvantages of domain-specific and domain-specific embedded languages, as well as examples of some successful languages and language design techniques, were scrutinized. After carrying out such a deep analysis of the domain, it should be evident that Haskell is truly one of the most appropriate languages which can be used as the host of a new domain-specific language.

The next chapter discusses the concepts related to Business Process Modelling.

Chapter 3

Business Process Modelling

3.1. Introduction

This chapter introduces the domain of Business Process Modelling, including construction, transformation and quality assurance techniques applied to business processes in Business-Driven Development. Modelling tools are also investigated with particular reference to IBM's WebSphere Business Modeler Advanced v6.0.2[1]. The embedded language, which we developed, tries to capture concisely and effectively the domain semantics of the modelling language used in this particular tool.

In the first section, the general concepts and terms shall be discussed, followed by an overview of some of the modelling notations and tools available today. Two other sections are dedicated for the analysis of transformations and quality assurance techniques which are of utmost importance to help users create models of a high quality. The transformation framework which was recently proposed in (Koehler, et al., 2007) by IBM for its modelling tool, WebSphere Business Modeler, to specifically fulfil this purpose is also reviewed.

3.2. General Concepts

When developing a system for a particular organisation, the business requirements and goals must be communicated to the IT specialists. While the business analyst is an expert in that particular business field, the IT specialist knows nothing about the business and the organisation. Similarly, the business analyst is not an IT specialist. However it is still vital for the business analyst to communicate the business requirements in profound detail for the IT specialist to learn more about the organisation and the problem, and thus try to come up with a feasible solution. For this reason, business process models are usually produced by the business analyst to graphically and textually communicate the current business processes to the IT department.

Since the IT solution is meant to reflect the current needs and goals of the business, the business process models and their respective implementation must not be viewed as two separate entities. Instead, the business process model must serve as an abstract representation of the solution and thus used as documentation for the implemented system. For this to be possible, contrary to traditional methodologies, the business process models and their implementation must not evolve independently. However, it is a known fact that while the development process of a system requires a considerable amount of time for it to be fully designed, implemented and tested, the company needs to adapt its goals and processes rapidly to keep up with the competitive market. This situation emphasises the need for *Business-Driven Development* (BDD) (Mitra, 2005) (Koehler, Hauser, Küster, Ryndina, Vanhatalo, & Wahler, 2006) whereby implementations

[1] http://www-306.ibm.com/software/integration/wbimodeler/

are directly derived from the business needs. In BDD, processes are implemented in a *Service-Oriented Architecture*.

Adapting to Business-Driven Development (BDD) and thus producing implementations for specific business processes by carrying out a number of refinements to the original description, brings about new responsibilities. Before deriving the implementation, it is important to ensure the quality of the produced model. If business analysts do not produce models of a high quality from which the final executable code could be derived, then the probability is that errors are discovered later on in the development process, leading to a waste of resources, mainly time and money. Besides quality assurance, the business analyst must also provide all the required details for executable code to be derived. However business analysts are not IT experts and thus, it might not be so intuitive for the analyst to realize that specific details are actually required. Thus, to assist the business analysts to rapidly create high quality business process models (to respond to changes in the processes) model transformations (Koehler, et al., 2007) are required. With such pre-implemented model transformations, business analysts would be able to carry out a number of different transformations in just a few steps and receive an immediate feedback on the quality of the models.

The main concept behind business process modelling as well as the Business-Driven Development (BDD) methodology shall be discussed in the following sub-sections. Some modelling notations and tools in particular IBM's WebSphere Business Modeler[1] are discussed in Section 3.3, whereas transformations and quality assurance as investigated in sections 3.4 and 3.5 respectively.

3.2.1. Business Process Modelling

In process modelling, a sequence of business activities, with clearly defined inputs and outputs, is specified. The activities are assigned a particular order and additional information is added to the process, with the main aim of capturing the business' requirements and objectives. In this way, business analysts are able to represent both the current (*'as is'*) and the future (*'to be'*) processes of the organisation. These are later analysed with the main objective of improving the efficiency and the quality of the processes, before they are implemented.

Depending upon the type of analysis and operations that need to be carried out on the process representations, processes are usually modelled at one of these levels:

- ***Process Maps***

  Processes are represented as simple flowcharts. The only details that are usually provided include: the names of the activities and the conditions assigned to decisions

- ***Process Description***

  Additional information is provided to the process defined in the previous level. Such information usually includes the data involved and the people who are expected to carry out the activities. Nevertheless, the processes modelled at this level, are still missing some essential technical details, which are required to execute the process

- ***Process Models***

  Processes at this level are still modelled as flowcharts but in contrast to the previous levels, the models contain all the required details for them to be analysed, simulated and executed, using some external tool

[1] http://www-306.ibm.com/software/integration/wbimodeler/

Since the final process representations, should unambiguously and consistently capture all the required information for both the business analysts and the developers to fully understand the business requirements, then processes modelled at the third level, shall be considered in this project.

For this reason, the business analysts should first discuss the business requirements with other members of the organisation and business requirements owners, and then construct the model to include those requirements, possibly by modifying an existing process model (also known as the *'as is'* model). In this way, the future (*'to be'*) process model, would not have to be constructed from scratch. To ensure the completeness of the models, which would later be passed on the IT developers, the model should represent the business process flow as well as the data, resources and any other commodities that flow through the modelling elements in the process. Modelling elements such as activities, forks, decisions etc. (discussed in Section 3.3.2.1) should be connected together using some control flow connectors. Rather than basic elements, a business process can be seen as a group of sub-processes, which, when connected together, make up a complex model. This would help the modeller to decompose the model and handle it at different abstraction levels. Moreover, pre-existing modelling artefacts such as services or activities can be re-used, thus reducing the amount of time and effort to construct the final model.

Ideally the final complete process model should, besides capturing the business process requirements, model the business items, roles and resources, services, policies and Key Performance Indicators (KPI – as a form of metrics to measure the progress made towards the projected organizational goals) should also be defined. Moreover, the model should serve as a complete documentation for readers to understand the business processes and for collaboration requirements, thus fulfilling regulations, such as Sarbanes-Oxley and Basel II.

To facilitate the production and maintenance of such models, different modelling tools, notations and languages were developed such as Business Process Modelling Notation (BPMN) (OMG, 2008), Unified Modelling Language 2 Activity Diagram (UML2-AD) (OMG, 2005), Event-Driven Process Chain (EPC) and IBM WebSphere Business Modeler[1] (IBM, 2006). Other standards which were established for the execution of these models include: Business Process Execution Language (BPEL), Web Services Description Language (WSDL), IBM WebSphere Business Integration Developer[2].

3.2.2. Business-Driven Development

While business processes constantly change to keep up with the demands of the changing environment, market trends and competition, the traditional software development life-cycle to implement these processes requires more time and effort. To adopt a more agile approach a business-driven rather than IT-centric methodology should be adopted, such that IT solutions that directly satisfy the business requirements and needs would be produced through a process of refinement from the abstract business process model to a more concrete IT implementation. For this reason, using such a methodology, business process models are no longer used exclusively for documentation purposes. Such models in Business-Driven Development (BDD) (Mitra, 2005) (Koehler, Hauser, Küster, Ryndina, Vanhatalo, & Wahler, 2006) have an additional goal, that to serve as input to some tool which would automatically derive the executable code and thus speed-up the implementation of such business processes.

Moreover, basing on the principle of decomposition and thus the idea that a process is intuitively a group of sub-processes which can in turn be made up of other sub-processes, simple software components and services would be defined such that, by combining these components in a variety of ways, different complex software components can easily be composed to satisfy the required purposes. If the current

[1] http://www-306.ibm.com/software/integration/wbimodeler/
[2] http://www-306.ibm.com/software/integration/wid/about/

software services cannot be used to define the requirements of the business process model, then new simple services would be defined and added to the IT service portfolio. This would ensure the production of good quality, highly maintainable software, with the least amount of effort, by primarily re-using the current functionality and avoid redundancy. Thus, considering that nearly 80% of the organisation's IT budget is spent to maintain and enhance their IT systems, such a methodology would reduce such costs, dedicating more time and effort to improve the efficiency of the processes.

The advantages of such a technique led to the development of Service Oriented Architecture (SOA), with the aim of providing the appropriate guidelines and infrastructure for the development of IT solutions as a set of reusable, configurable and composable services. These services are developed with the intention of fulfilling just one simple objective independent of the application and run-time framework, such that they can be invoked, composed into another complex service and executed by any machine.

The main phases of the *Business-Driven Development life-cycle* are illustrated in Figure 3.1 and discussed below (Mitra, 2005).

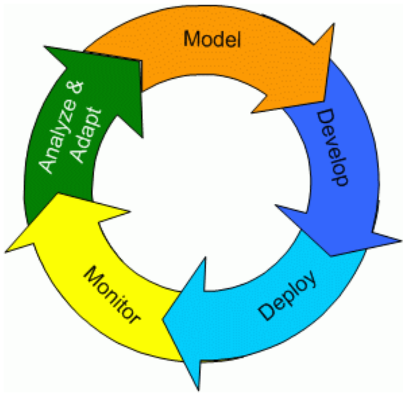


Figure 3. 1 : BDD life-cycle (Mitra, 2005)

For the process to commence, the business requirements must first be represented as business process *model*s. At this stage, it is advisable for the analyst to associate some significant metrics such as Return On Investment (ROI) and KPI (Key Performance Indicators) which would be helpful at later stages of the development process whereby the effectiveness of the resulting IT solution would be analysed. These models are then passed on the IT developers, to implement the processes.

Before the processes are implemented, the requirements of the IT solution are derived from the business process model. Existing services that can be used for the implementation of these processes, as well as the new simple services that need to be developed, are identified. New services are then designed and *develop*ed and subsequently, wired together with other existing services to fulfil the requirements of the business processes.

When a new business process is implemented, a new composite service is produced. Since business processes are decomposable into a number of sub-processes, some of which would already be implemented, then new composite services should be *deploy*ed as discoverable and location-transparent services on some execution runtime such as an application server. In this way, such services can easily be discovered and re-used for the development of other composite services, and thus prevent the unnecessary re-implementation of processes.

It is not enough to simply implement business solutions. It is of utmost importance to *monitor* the business processes. Besides evaluating the performance of the processes, the latency, reports and the data produced should be analysed and compared to the original requirements specified in the business process model. This type of evaluation is usually carried out against business metrics (such as ROI and KPI), that are specified at the modelling phase, and the expected Service Level Agreement (SLA)[1].

The results are *analysed* and possible optimizations and enhancements are noted. While some of these improvements require changes to be carried out in the actual code, others merely involve the *adaptation* of business rules by some external tool which can easily be carried out by the business users themselves. If the actual business processes would have to be modified, then the required changes are passed on to the *modelling* phase and the entire cycle is carried out again, with the intention of producing better improved and optimised code. This would guarantee that the IT solution is strictly derived from the business process

[1] For an outline of the core elements of an SLA refer to http://www.sla-zone.co.uk/

model through a process of refinement, thus enforcing the same requirements as defined in the original model. If the implementation is modified without modifying the business process model, the model would no longer reflect the underlying implementation. Moreover, the process would not really be maintainable as issues would be encountered when transformations are carried out on the current *'as is'* process to create the future *'to be'* process.

To increase the probability of generating the required optimized solution immediately without having to repeat the entire life-cycle over and over again, modelling tool developers are continuously noting the importance for such tools to assist the users in the construction of high quality models from which the required executable code could be derived. For this reason, the incorporation of a number of transformations and quality assurance techniques are being investigated (Koehler, et al., 2007). Moreover, such tools also allow business analysts to simulate the performance of the processes they model, against business metrics such as ROI and KPI, and to carry out static analysis (Fasbinder, 2007).

To help companies manage business processes in Business-Driven Development, IBM has developed a set of tools, which assist users at the different phases of the methodology, as shown in the following diagram (Wahli, Avula, Macleod, Saeed, & Vinther, 2007):

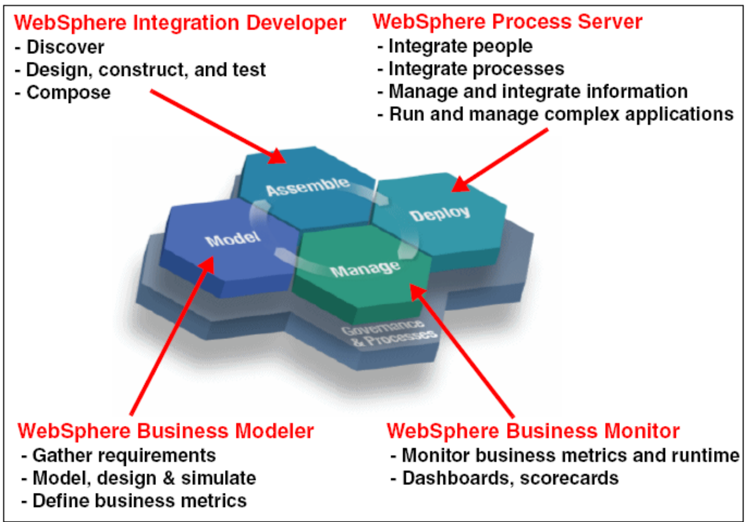


Figure 3. 2:

Using IBM's WebSphere products in Business-Driven Development

(Wahli, Avula, Macleod, Saeed, & Vinther, 2007)

Note that in the Figure 3.2, the '*Assembly*' phase is synonymous to the '*Develop*' phase in Figure 3.1, whereas the *'Manage'* phase includes both the '*Monitor'* and the '*Analyze and Adapt'* phases in Figure 3.1.

The tools provided by IBM include:

- ***WebSphere Business Modeler*[1]**

  This tool is used by business analysts to design the required optimized business process, by specifying the activities involved and the order in which they should be processed. The constructed model can be simulated and thus optimized. With this tool, the analyst can export the business processes in a variety of formats such as Business Process Execution Language (BPEL) for implementation (This tool is discussed in Section 3.3.2). Thus, it is used in the modelling phase of the development life-cycle.

[1] http://www-306.ibm.com/software/integration/wbimodeler/

- ***WebSphere Integration Developer*** [1]

  To implement a modelled business process, produced by some tool such as WebSphere Business Modeler (but not necessarily IBM's tool), the modelled flow is first transformed to an executable IT flow such that the activities in the model are mapped to some reusable service components. These services are then combined so that a composite service which fulfils the goal of the business process is generated. This is in fact an excellent tool to assist developers in the implementation of activities in the development (or assembly) phase of the business driven development life-cycle.

- ***WebSphere Process Server*** [2]

  A system administrator can use this as a production server to deploy, manage and run the process implemented as a composite service. Thus it is used in the deployment phase of the life-cycle.

- ***WebSphere Business Monitor*** [3]

  This tool is ideal for system administrators to monitor the real-time performance of the developed solutions. It is based on the concept of Business Activity Monitoring (BAM), whereby the events produced by the system are monitored. Moreover, if the administrator wants to test the business measures of the system, before deploying them to a Monitor Server, the Monitor Development Toolkit, which is a plug-in in the Integration Developer, can be used. The main objective of this tool is to try to improve the model and its implementation. Thus it is used in the '*manage'* (or *'monitor'* and *'analyse and adapt'*) phase of the life-cycle.

More information about the products is available in (Wahli, Avula, Macleod, Saeed, & Vinther, 2007) or [4].

From the above discussion, it should be evident that besides reducing the cost of software development, through the use of composable and reusable services to implement a variety of business processes, the development time and the quality of the final solution are improved, thus fulfilling the customers' expectations. Tools such as those provided by IBM help companies to adopt this methodology and allow them to benefit from the inherent advantages of Business-Driven Development (Mitra, 2005).

3.3. Business Process Modelling Notations and Tools

Over the years, a variety of tools and methodologies that use different languages and notations, were developed to help business analysts produce business process models and thus, represent the business requirements in an unambiguous manner.

Some of the commonly used notations include Unified Modelling Language Activity Diagrams (UML2-AD) (OMG, 2005), Event-Driven Process Chains, IBM's Business Modeler modelling language (IBM, 2006) and the most recent is Business Process Modeling Notation (BPMN) (OMG, 2008). Some of the modelling tools which use one of these notations to model the processes, include Maestro, which uses BPMN, and IBM WebSphere Business Modeler (IBM, 2006), which uses its own modelling language.

Although different, the main modelling elements of these notations are quite similar. In the following sections, only two notations shall be discussed. These include: BPMN, the most recent notation which was defined with the intention to create a standard modelling notation that captures the features of most of the

[1] http://www-306.ibm.com/software/integration/wid/
[2] http://www-306.ibm.com/software/integration/wps/
[3] http://www-306.ibm.com/software/integration/wbimonitor/
[4] http://www.ibm.com/developerworks/websphere/zones/businessintegration/

previous notations, and IBM WebSphere Business Modeler modelling language, for which we have designed and developed a declarative functional language embedded in Haskell (see Chapter 4). Although our language does not strictly implement the features of BPMN, this notation has been approved as a standard, with which any models can be produced, and thus its features were also analysed, with the aim of identifying the right primitive modelling components for our language. Using these components, any model can be defined, and the user can later export them to IBM WebSphere Business Modeler.

3.3.1. Business Process Modelling Notation

Business Process Modelling Notation (OMG, 2008) is one of the most recent modelling notations, which, in February 2007, became an Object Management Group (OMG)[1] standard. Similar to other previous notations, it is a flowchart based notation, capable of modelling any type of business process at any of the levels defined in Section 3.2.1. Although it is referred to as a notation, it also defines the semantics of the model, since it allows modellers to include all the required details for a process model to be refined to executable code.

Although various tools and notations were previously available to model business processes, there was yet a great necessitate for a standard notation to be established, which would be used by most of the companies. Thus, to ensure that this standard notation contains all the features required by the different organisations and thus encourage most of the companies to adopt this standard, a large number of vendors got together and shared their experience and the features that were of utmost importance for their tools and notations. Their main objective was to agree upon a single notation that would cater for the requirements of different business users, such that users would be trained to use a single notation. Ideally the notation would be adopted by most of the modelling tools, such that, independent of the tool used and the company they worked for, modeller would still be able to understand, transform and produce models. Similarly, it would be easier for the IT developer to understand the models (since they would all be produced using the same notation). The companies on the other hand, would save up on the training costs to retrain their modellers each time a new tool is used. Another advantage that is especially important when a business-driven approach is adopted, is that, using a single notation, it is easier for executable code to be generated from the modelled processes. Thus, by passing on a BPMN process to a BPMN engine, the corresponding executable code defined in the Business Process Execution Language (BPEL – short for WS-BPEL, that is, Web Services BPEL) would be generated. Models constructed using other notations cannot be directly translated to BPEL. Either a tool is required to translate the model into BPMN or else a tool (possibly the modelling tool itself, as is the case with IBM's WebSphere Business Modeler) is required to generate the BPEL code from a model defined using another notation. It is important to note that, as defined in OASIS Standard WS-BPEL 2.0, with BPEL, the behaviour of business processes is defined using web services, and thus executable code would be produced.

However, for such objectives to be achieved, different versions of the notation were produced until a stable standard notation was established. Initially, the Business Process Management Institute (BPMI), which is now part of the Object Management Group (OMG), developed Business Process Modelling Language (BPML), as an XML (eXtensible Markup Language) process execution language. However, this language was very technical and could not be used by the business users. Considering that IT analysts and developers know nothing about the organisation and its requirements, it was essential to develop a language that was easy for business analysts to use. Thus, a notation to graphically represent the business processes was required. This meant that some translation from the business oriented notation, used by the business analyst, to the technical execution language, had to be carried out. For this reason, BPML was replaced with the new target execution language BPEL and later on, in August 2001, the Notation Working

[1] http://www.omg.org/

Group for BPMN was formed to work on the new business-oriented language. As mentioned in the previous paragraph, this group was made up of 35 modelling organisations, companies and individuals who discussed various aspects of modelling. This led to the development of the first version of the notation, BPMN 1.0, which was first released to the public in May 2004. Since then, the main objective behind the development of such a standard started to be achieved, as various companies (around 41), started to make use of this notation. In February 2006, it was approved as an OMG standard and in June 2007, BPMN 1.1 (OMG, 2008) was released.

3.3.1.1. Modelling Elements in BPMN

Since business processes can be very complex and since the users of BPMN are business analysts who are not necessarily IT technical people, then simple shapes similar to normal flowchart components, were chosen as the basic modelling elements. If on the other hand, the user wants to illustrate some specific complex aspects of the process, then they can still do so using variants of the same notation.

Thus, to create a simple notation that is intuitive and easy to reason about, but which still captures the complexity that users might require, three basic shapes were chosen (Figure 3.3). To model specific complexities, then variations of these basic shapes can be used.

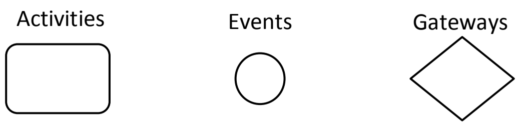


Figure 3. 3: Basic BPMN shapes – Activities, Events, Gateways

By using the shapes in Figure 3.3, the reader can easily identify the behaviour of each element. These were specifically chosen such that they are easily distinguished from a distance. While *activities* are the basic components that illustrate the behaviour of tasks (e.g.. *'Update Customer Order'*), *events* are meant to affect the process (usually start or terminate the process), whereas *gateways* (such as decisions) control the control flow within the process. Since they are the basic modelling elements, they are collectively known as *Flow Objects*. However for a process to be constructed, these flow objects need to be connected in some way. For this reason, three different types of connectors are provided (Figure 3.4).

Figure 3. 4: BPMN Connectors - Sequence Flow, Message Flow, Association

One of the most important connectors, which are frequently used, is the *sequence flow* connector which defines the flow of control within the process and thus the sequence of activities and events. *Message flow* on the other hand, defines the communication between business entities, example between lanes (see swimlanes below), whereas *associations* define some additional information about the flow objects and their relationship with other objects. Thus, using these modelling elements a process such as Figure 3.5 can be constructed:

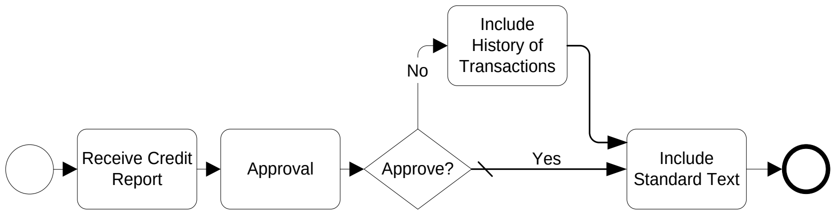


Figure 3. 5: Example of a process defined in BPMN (OMG, 2008)

The model in Figure 3.5 illustrates a start node, followed by activities '*Receive Credit Report'* and '*Approval'* and a decision to check whether it is approved. If approved, activity '*Include Standard Text'* is immediately executed. If not, then the activity '*Include History of Transactions'* is executed before activity '*Include Standard Text'*. The process then terminates. Note that only control flow is depicted (i.e. sequence flow connectors).

Other elements that are provided are categorized as *artifacts* (Figure 3.6) and *swimlanes* (Figure 3.7):

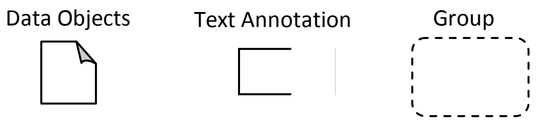


Figure 3. 6: BPMN Artifacts - Data objects, Text Annotation, Group

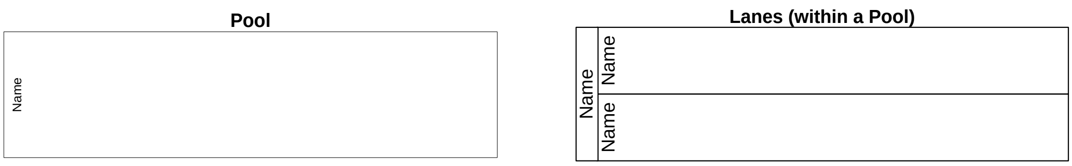


Figure 3. 7: BPMN Swimlanes - Pool, Lanes

The three standard artefacts include: *data objects* to represent any data that might flow in and out of activities in the process, *annotations* for documentation purposes and *groups* as a visual aid for the user to better analyse the process. New artifacts can be added to cater for the specific business requirements. For instance, a cylindrical shape can be added to represent a repository such that Figure 3.8 would still represent a valid process fragment defined with this notation.

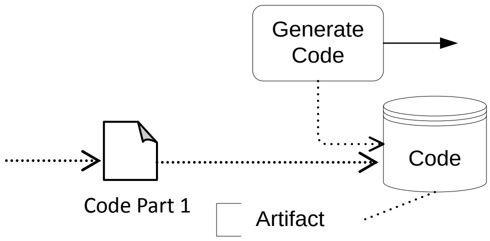


Figure 3. 8: A model containing artifacts

In Figure 3.8, the repository is keeping *code* which is produced as output of some activity labelled as '*Generate Code'* and some data object '*Code Part 1'* which could possibly be generated by some other activity. Some documentation is also included to indicate that the new cylindrical shape is an '*Artifact'*.

It provides two types of *swimlanes*. These include: *pools* to act as containers, to separate collaborating processes, and *lanes*, to partition groups of elements that are specific to a role, within the process. For instance, to depict some activities in a medical client, it could be that two lanes for Patient and Doctor's Office and two pools within the latter lane labelled as Doctor and Receptionist are used.

In this section only the main and most important features of BPMN are discussed. For a detailed description of the notation, the following source should be consulted (OMG, 2008).

3.3.2. IBM WebSphere Business Modeler

As already mentioned in Section 3.2.2, IBM WebSphere Business Modeler[1] is one of the products which IBM has produced to help business users to model their processes and to assist companies to adopt the Business-Driven Development (BDD) methodology (Mitra, 2005) in a Service-Oriented Architecture (SOA). This tool is used in the first phase of BDD and is one of most important steps in the life-cycle since it produces the process models from which the actual implementation would be automatically derived.

Although there are various reasons why and how a process model is used within the organisation, IBM WebSphere Business Modeler results to be a helpful tool for all of these goals. For instance, if the organisation uses process models for tactical purposes such as to accurately document processes for training, legal or regulatory purposes, through the possibility of sharing modelling elements, attaching documents and using other collaborating features, IBM's tool results to be an excellent tool to easily construct these models. If on the other hand, such models are used to improve the model by redesigning the process, features such as simulation, process comparison, and analysis and reporting are all important features that help users of IBM's tool. Another example of some of the most important features to assist users during construction of models is the reuse of components to produce the '*to be'* future process from the *'as is'* current process, and thus, reduce drastically the implementation time. The next and the most important inherent advantage, is the use of such a tool to model the process to commence the Business-Driven Development life-cycle. Moreover, following the CIO study (CIO, 2004), process improvement is considered to be the topmost priority (with an 85% of priority) of the business. Similarly other features which are also given some of the highest priorities include: gaining a better ROI, reducing the implementation time of processes, using IT to document and capture process information. All these priorities are all addressed by IBM's modelling tool.

To complete the implementation of the modelled processes, the tool is able to transform the models into various formats that make it possible for other tools to import the model and continue with the implementation of the process. For instance, if new components and services need to be developed, the tool is able to export the models into UML artefacts. These models can then be imported by other tools such as IBM's Rational Software[2] or Rational XDE[3], which would implement the new services. Similarly, for the process choreography (that is how the services that make up the process collaborate) to be defined, the model can be exported to WS-BPEL for WebSphere Integration Developer[4], BPEL4WS for WebSphere Studio Application Developer Integration Edition[5], or FDL (Flow Definition Language) for WebSphere MQ Workflow Buildtime[6]. Moreover, business criteria that should be monitored after the deployment of the process, are exported in the form of Business Measures Model, which is later imported into WebSphere Business Monitor Toolkit where the system integrator would complete the monitored model (IBM, 2006). This model is later deployed to the WebSphere Business Monitor[7]. Once the model is monitored and analysis results are obtained, these are imported into the WebSphere Business Modeler for the model to be improved (IBM, 2006).

An important fact about WebSphere Business Modeler is that it is easy to use. Most of the operations are possible through a couple of clicks. The final interface was obtained after carrying out extensive user-based testing. Moreover by providing different editing modes that is Basic, Intermediate and Advanced, users are able to view the details of the models at different levels of abstraction depending about the

[1] http://www-306.ibm.com/software/integration/wbimodeler/
[2] http://www-306.ibm.com/software/rational/
[3] http://www.ibm.com/developerworks/rational/products/xde/
[4] http://www-306.ibm.com/software/integration/wid/
[5] http://www-306.ibm.com/software/integration/wsadie/
[6] http://www-306.ibm.com/software/integration/wmqwf/
[7] http://www-306.ibm.com/software/integration/wbimonitor/

requirements and skills of the users. All these features would ensure the key factors of the tool, that include, ease of use, speed and readability, as illustrated in the Figure 3.9 (obtained from (IBM, 2007)).

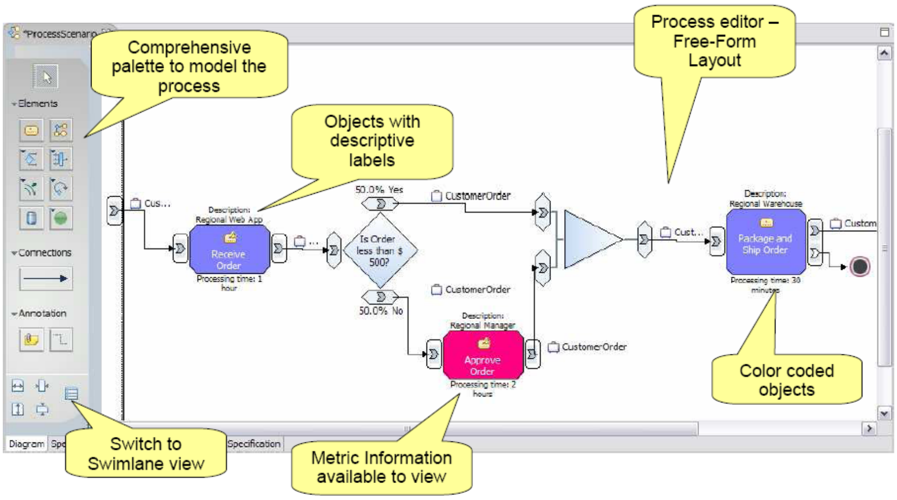


Figure 3. 9:

A screenshot of IBM's WebSphere Business Modeler, illustrating some of its features (obtained from (IBM, 2007)).

IBM’s WebSphere Business Modeler has defined its own simple modelling language to allow business users (who are not IT technical people) to easily manage their models. Although it is similar to BPMN, certain features are notably different. The embedded language which we have created caters for the core features of this modelling tool. Thus it is of utmost importance for the reader to understand the following concepts. These shall be referenced in Chapter 4 where the design of our language is discussed. Although a very recent version (v6.1) of the tool has been issued, this project focuses on the Advanced edition of version 6.0.2[1]. In v6.1, very minor changes were carried out to the tool and not to the modelling language.

The next section gives an overview of the main modelling elements provided by the language. The subsequently section discuss a number of other features of the language. Note that this is not really meant to be a tutorial of how to use the tool (see user manual of tool or (IBM, 2006)). Instead, it points out just the main concepts that a reader must be aware of to understand the design of our embedded language in Chapter 4.

3.3.2.1. Basic Modelling Elements

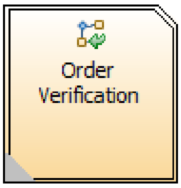
The basic modelling elements which are provided on the side palette and which can be used to construct models include:

Task

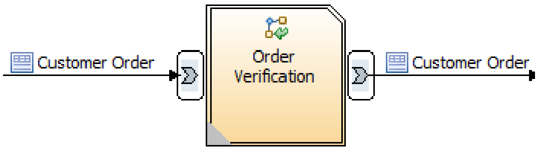
This is one of the basic components of the model, which performs specifically one function within the process. The task is assigned a name which is usually significant to the operation it carries out (‘*Review Loan Application*’ in this example). It also has some inputs and outputs defined, as illustrated in the example on the left. In this case, as input the task has some data item of type String (a basic in-built type), whereas as output it returns a business item of type Application, which is a user-defined type (business items are discussed later in Section 3.3.2.2).

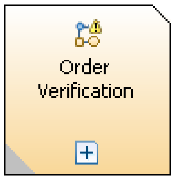
[1] http://www.developers.net/ibmshowcase/product/WebSphere_Business_Modeler_Advanced

Sub-Process



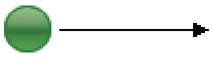
Another element which carries out some activity is a sub-process. The main difference between a sub-process and a task is that while a task is a basic element, a sub-process is strictly another process, whose internal structure is abstracted away by such a shape. Similar to a task, when the sub-process is constructed, it is assigned a name (in this case '*Order Verification'*) and the types of the required inputs and produced outputs are defined, as illustrated in the example on the left. In this case, a business item of type Customer Order is expected as an input and produced as an output. If the sub-process is defined and used locally inside a specific process, then as small plus sign is visible to indicate that the component can be expanded for the internal details to be visible. (Business items and local and global activities are discussed in a later sections 3.3.2.1.3 and 3.3.2.1.4 respectively)





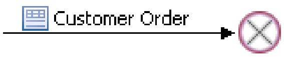
Start Node

This node indicates the beginning of a process flow that is not associated with any data. Thus to indicate the beginning of a task immediately after the process is passed on control and the task does not have any data inputs, then a start node is required. Note that this node has no inputs and always generated one output that is control (see diagram on left)

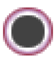


End Node

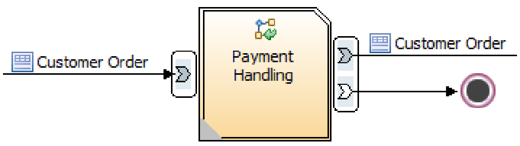
This node indicates the end of a process flow within a process. For instance, if a particular data item is no longer required by any of the remaining tasks, the flow of such data can be ended as illustrated in the example on the left. In this case, the data item concerned is a business item of type Customer Order. In a similar way, a particular control flow can be terminated. In contrast to the stop node, this node does not terminate the entire process but just one particular process branch



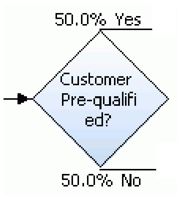
Stop Node



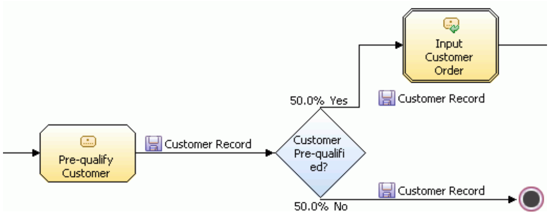
This node indicates the end of a process (and not just a process flow, as the case with end node). When a process terminates with a stop, all the required data or simply the control (if no data output is generated), would be passed on to the parent process. For this reason, to ensure the liveness of the flow, the last node in every process should always have a stop node (as shown in the example on the left). However, if this node is not used appropriately, then process flows within the process might be forced to terminate while other activities are still being executed. This would lead to errors in the model

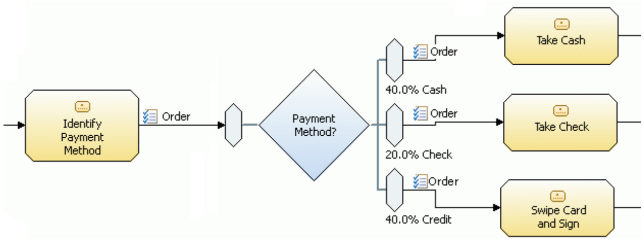


Decision

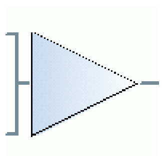


Two-choice (i.e. 1 input & 2 outputs) or multiple-choice (1 input & multiple outputs) decisions are provided. Exclusive and inclusive versions of these decisions are available such that if exclusive, the inputs are routed to only one of the outgoing branches, whereas if inclusive, the inputs are routed to at least one of the outgoing branches. These decisions are synonymous to the exclusive and inclusive gateways in BPMN. However, in contrast to BPMN, the shape used for both types is the same. Thus the only way to identify the different decisions, the attributes of the element have to be read. A condition should be defined for each branch, such that by evaluating this expression, the inputs would be routed accordingly. Moreover the probability of occurrence of each branch is added. This would contribute to the simulation of the modelled process. In the following examples, '*Customer Pre-qualified?*' is an exclusive two-choice decision whereas '*Payment Methods?*' is an inclusive multiple-choice decision

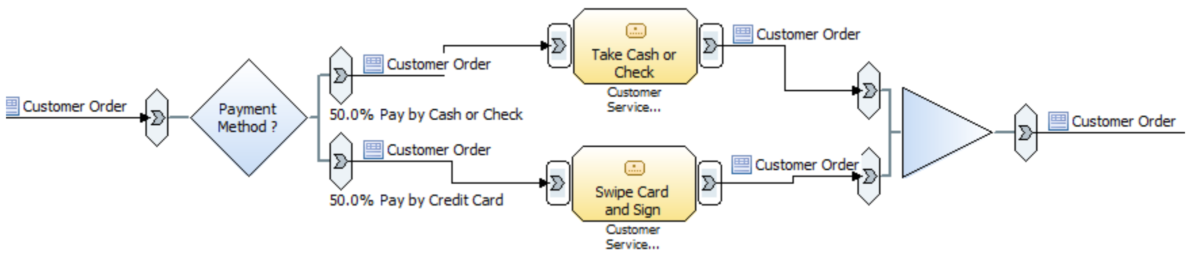


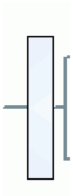


Merge



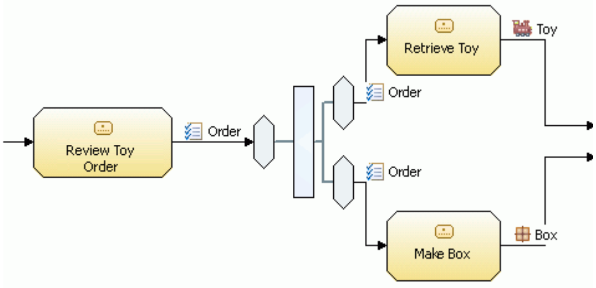
This element recombines processing paths (any number of incoming branches), such that, as soon as some input data or control is received on one of the incoming branches, these inputs are passed on to the next element after this merge. Thus it only waits for inputs to arrive on one of the paths. The only problem with the tool is that if the inputs arrive from both branches, not necessarily at the same time, the sequential elements are evaluated twice. This element usually follows from a decision preferably an exclusive decision (else the scenario mentioned in the previous sentence would occur). The following is an example of a decision-merge fragment:

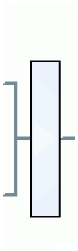




Fork

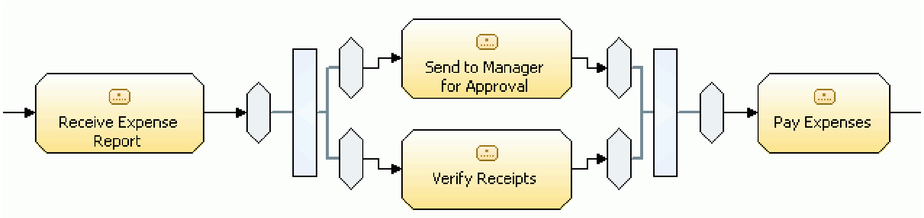
This element makes multiple copies of its inputs and forwards them along the several outgoing paths, such that all the paths would be executed in parallel. This is synonymous to the parallel gateway in BPMN. Example:





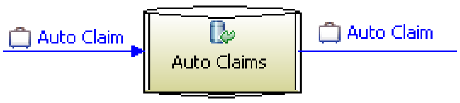
Join

This element recombines and synchronizes all parallel processing paths, such that, it passes data or control to the next element in the process only when data or control is obtained from all of its incoming branches. Join can have more than two incoming parallel paths and it is usually used after a fork, as shown below:

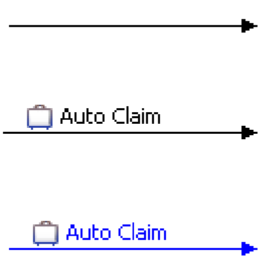


Repository

This element acts as a data store for just one specific type of data. The data in the repository can be accessed as illustrated in the example. If a repository is defined locally within the process then only the data stored during the execution of that process would be available.



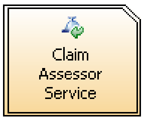
If the repository is defined globally then the process would have access to data produced and stored within the repository by other processes (local and global elements are discussed 3.3.2.3)



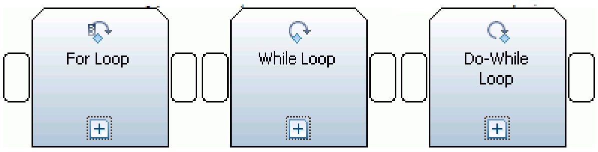
Connector

This simply indicates the flow within a process and thus links elements with a process. To indicate simply control flow then the first connector on the left is used. To indicate data flow then a connector similar to the second (in this case this connector indicates the flow of some business item of type `AutoClaims` – business items are discussed later in Section 3.3.2.2). Finally if the data flowing on the connector is coming from a repository, then the connector would be similar to the third (once again, the expected data is a business item of type `AutoClaim`)

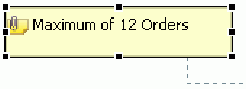
These are the most basic elements with which any business process model can be defined. Other elements which are also provided (but which shall not be dealt with) include (see user manual of tool):



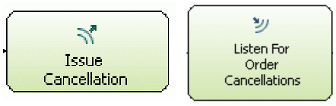
Services



Boxed Loops



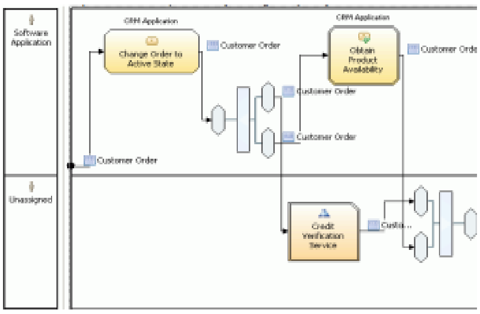
Annotations



Notifications Broadcasters & Receivers

Observers

Timers



Swimlanes & Pools (as in BPMN)

3.3.2.2. Control & Data Flow

Different from the BPMN, in this language, the data and the control flow are both represented on the same diagram. The connectors which are used to indicate data and control flow were discussed in the '*Connector*' section in the previous section. Thus, when data is flowing over a connector then actually that flow is representing both data and control. This is so, since an activity cannot be executed if it does not have control, even though it has the data. In this way, if a particular process flow includes for instance a data of type `a` and a data of type `b` and another indicates the same with the addition of a control input, then the types of the two process flows are still considered equal. This means that, although the data types of all the incoming branches of a gateway (that is a decision, merge, fork or join) must be equivalent to the data types of all the outgoing branches, the gateway in Figure 3.10 is still valid.

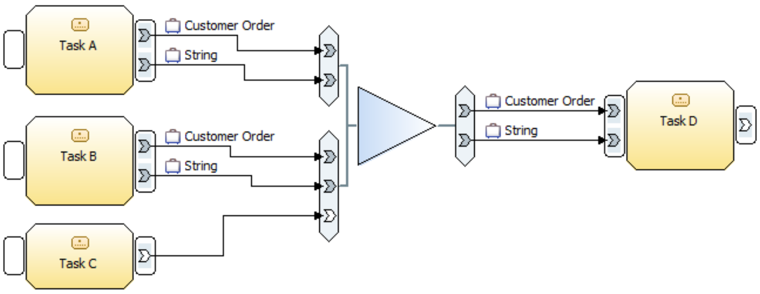


Figure 3. 10: A valid merge with 2 incoming branches. The same branch has an additional control flow input

Note that the second incoming branch of the merge is made up of two inputs of type `Customer Order` and `String` and a control input. This control input is not present in the first incoming branch of the merge. However, since control is always inherently part of data flow, the input types of the two branches are considered equivalent. Thus even though a specific data type is defined for the input or output of a process, still any number of control connections can be added without changing the actual type.

On the connectors the type of the data flowing between components is visible. It is important to note that one connector can only represent one type of data items flowing between elements. Thus, if more than one data item is required, then multiple data connectors would be required. The type of the data can be either one of the built-in *basic types* or a user-defined *complex type*. The provided basic types include: `Boolean, Byte, Date, DateTime, Double, Duration, Float, Integer, Long, Short, String, Time` (for specifications see the help of the tool, '*Reference*' section). Complex types include business items, business services and business service objects. However the most important and frequently used data items are business items.

These *business items* represent products, business documents or some other commodities that are used and produced by the processes. Examples of such items include customer order, invoice, a product. These items, whose name is selected by the modeller, are usually significant in a particular context. Similar to a class in object-oriented programming, the defined business items usually have a set of associated attributes, to refine their description, which are only accessible and visible when the details of the type are explored. However, while the constructing the model, such details are not required.

Another important feature is that the *input source* and the *output target* of specific elements needs to be defined, such that if a task explicitly defines that it expects a data item of a specific type as input from a repository, then the only element that can be connected to the input of that task, is a repository which stores that type of data. Table 3.1 illustrates the different possible input source and output target types of different modelling elements.

Modelling Element	Input Source	Output Target	Note
Activities (i.e. Tasks & Sub-Processes)	Flow, Constant or Repository	Flow or Repository	-
Gateways (i.e. Decisions, Merges, Forks, Joins)	Flow, Constant or Repository	Flow or Repository	*Although the input data types and the output data types must be the same for all the incoming and outgoing branches, the input sources of the incoming branches and the output sources of the outgoing branches can be different.*

Table 3. 1: Different possible input source and output target targets that can be applied to the modelling elements

Note that a constant indicates a constant input whereas flow indicates a usual data or control flow coming from any other element. Figure 3.11 illustrates the pins that are used to indicate different input source types:

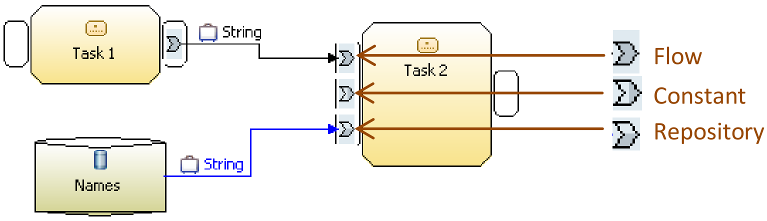


Figure 3. 11: The pins thatare used to illustrate different input source types

The input source and output target can be defined by first clicking on the element and then expanding either the *Inputs* or *Outputs* tabs where a column named *Input Source* or *Ouput Target* would be visible. The user can then select one of the types from the drop down list, as illustrated in Figure 3.12.

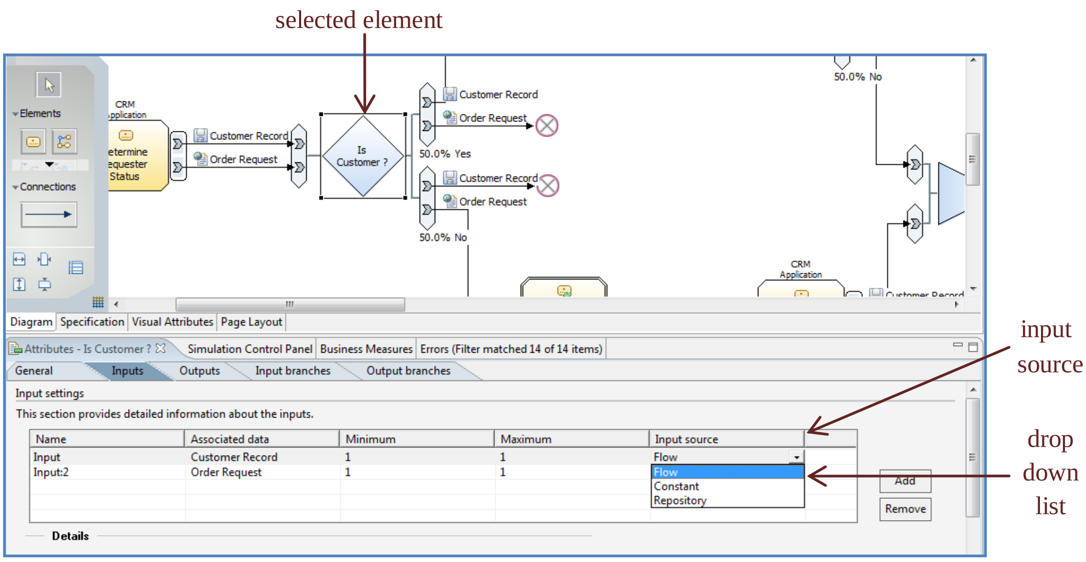


Figure 3. 12: Setting the input source type in IBM WebSphere Business Modeler Advanced

3.3.2.3. Local & Global Tasks, Sub-Processes & Repositories

When tasks, sub-processes and repositories are defined the modeller can either decide to define them as local elements of the parent process or else create them as global re-usable components. The main difference between the two is that while local elements are only accessible through the parent process in which they were defined, global elements are defined as independent components which are then visible in the side *Project Tree View* and which can be referenced and used within any process in the project. Figure 3.13 is a screenshot of the side *Project Tree View* of the tool.

It can be noted that global sub-processes are defined as other processes, and besides, processes, tasks and repositories, global services can be defined as global. Business items are all reusable by any process in the project. Once defined, they are visible in this Project Tree View. Similarly other commodities such as resources, roles, reports and queries are also visible (but are not being tackled in this project).

When a task is defined locally, the user is able to change the properties of the task such as its input and output types, while constructing the parent process. If on the other hand the user drags one of the global tasks which would have been previously defined, the user is only allowed to add control inputs. To change the properties of the task, the actual global task would have to be accessed by clicking on the task in the Project Tree. Similarly, if a local sub-process is defined, a plus sign is visible on the modelling element (as shown in Section 3.3.2.1) and the user is able to access the internal components and carry out the required changes.

If a repository is defined locally within the process, only the data stored during the execution of that process would be available. If, on the other hand, a repository is defined globally, the parent process would have access to data stored within that repository by the other processes in the project.

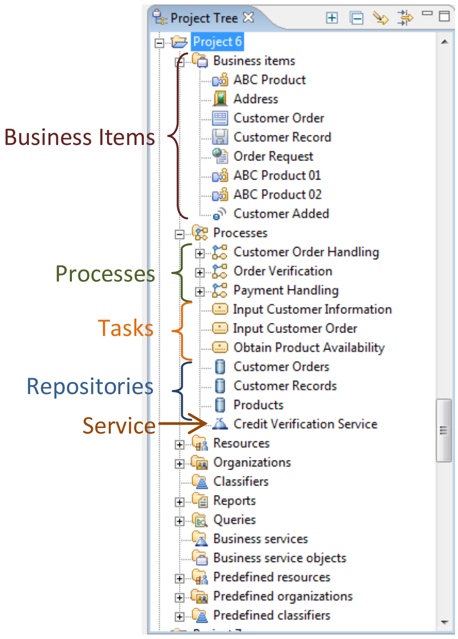


Figure 3. 13: Project Tree View of IBM's tool

3.3.2.4. Gateway vs. Activity Form

Gateways or control nodes (i.e. decisions, merges, forks, joins) are essential to model the flow of control and data. In addition to this approach, WebSphere Business Modeler modelling language and several other modelling languages, allow users to model the flow by adding some criteria to the inputs and outputs of process, such that the models would not have any gateways. Consider Figure 3.14:

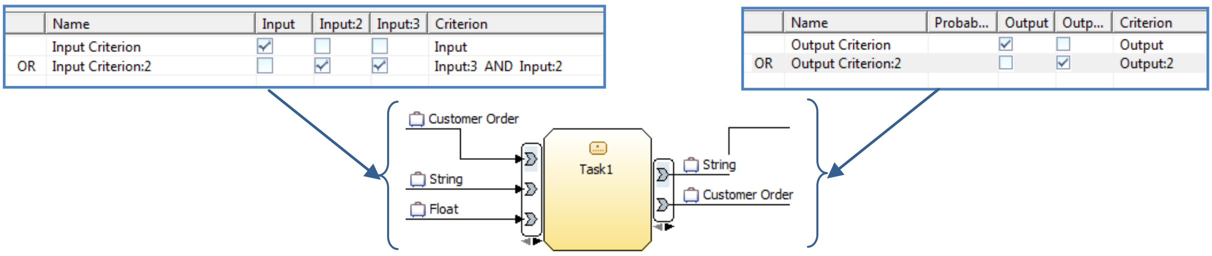


Figure 3. 14: Task 1 with 2 input and output criteria (illustrated in the upper blue boxes)

In Figure 3.14, `Task1` can either start executing when it receives an input of type `Customer Order` or when it receives two inputs of type `String` and `Float`. Similarly, the output of the task is not deterministic such that it can either output a `String` or a business item of type `Customer Order`.

In this way, when control and data flow of a process is modelled using this approach, an activity can only execute if at least one of the input criteria is satisfied. When it terminates then, it produces only the outputs of exactly one criterion. But how would the reader know which and when is a specific output produced? To avoid this ambiguity, the modeller should explicitly associate input and output criteria such that the reader would immediately recognize that a specific output is only produced when particular inputs are passed on to the activity. If these associations are no explicitly specified then the activity would produce one of the output criteria in a non-deterministic manner, thus leading to ambiguities.

Thus, a modeller can either opt to model the process in gateway form or in activity form. In the former approach, only gateways should be used and every task and sub-process in the process must not have more than one input and output criterion (meaning that the user simply defines the inputs and output which would be handled in a deterministic manner). In the latter approach, no gateways should be used and the activities in the process must have multiple input and output criteria defined to model the branching and joining behaviour of gateways. The models in Figure 3.15 and 3.16 define the same behaviour using the two different approach; the first is an activity form model, whereas the second is a gateway form model.

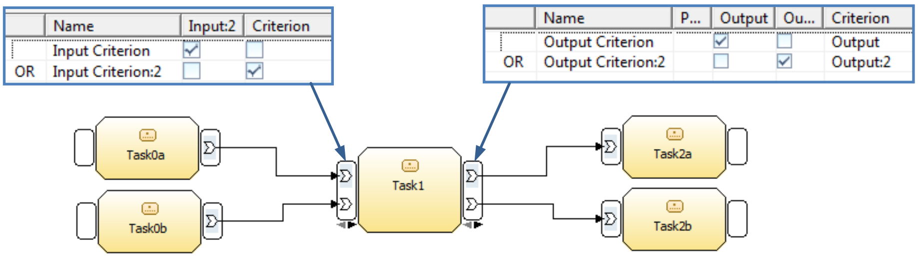


Figure 3. 15: Same model as Figure 3.16 but in activity form

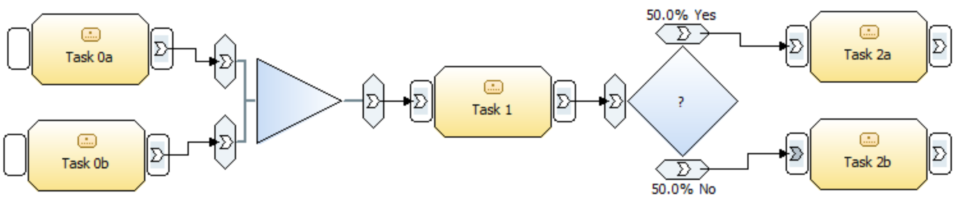


Figure 3. 16: Same model as Figure 3.15 but in gateway form

Both approaches have their own advantages and disadvantages which the user should be aware of (as defined also in (Koehler & Vanhatalo, 2007)). Ideally the modeller should choose one approach and stick to it to ensure homogeneity in the modelling style. *Gateway form* is usually more readable when just the control flow is illustrated in the model. When data is added to the flow, then the reader might prefer to view just the activities in the process and thus the gateways in the model should ideally be abstracted away. For this purpose, an *activity form* model would be more suitable as it is more compact and allows the reader to focus on the activities, which constitute the main functionality of the process. However, this has its own drawbacks since it is usually more difficult for the modeller to capture the flow logic in terms of input and output criteria, and thus increasing the modelling time and the probability of introducing bugs and errors. Moreover, decision conditions in activity form are captured as post-conditions of output criteria. It is more intuitive for a modeller to just edit the output logic of the decision, as is the case in gateway form.

3.4. Model Transformations

Model transformations are essential in Business-Driven Development. Starting off with a simple model which captures concisely all the business requirements of the organisation, the executable code needs to be generated. Such a Model Driven Architecture (MDA) (launched by the Object Management Group (OMG)) is possible through the application of transformations, whereby an abstract model is transformed into a more concrete representation which is closer to the required executable code. This refinement process would ensure that the produced solution fulfils the original business requirements specified in the business process model.

The benefits noted by Model Driven Architecture led to the recognition of a variety of model transformations which are usually classified in the following manner (Koehler, et al., 2007). Transformations are first categorised as *endogeneous* or *exogeneous*. The main difference between the two classes is that while in the former, transformations produce models that belong to the same meta-model as the source, in the latter, transformations usually map models from different domains. For instance, *endogeneous* transformations are usually used to derive the design model from the model produced at the analysis phase, whereas *exogeneous* transformations are usually applied to generate the executable code from the business process models. Also, it is notable that the gap between the source and target model in the latter is greater than that in the former.

Endogeneous transformations are further grouped into *out-place* and *in-place* transformations. While the former produces a completely new model, the latter simply modifies the source model. Additionally, *vertical* transformations refine abstract models into more concrete representations or vice-versa, and *horizontal* transformations remain at the same abstraction level by applying, for instance, refactoring which is a semantics-preserving transformation. Another important classification is to identify whether the transformation is *destructive* or not. In the former, existing modelling elements are usually deleted, whereas in the latter, elements are simply added. These are just some of the types of model transformations that have been identified over the years. In some cases, complete taxonomies of model transformations have been identified as in (Mens & Gorp, 2006).

Our main objective in this project, as regards model transformations, is to simply investigate the usefulness of such transformations at the modelling phase to assist users to produce good quality models at the least amount of time and effort. For this reason, we are only interested in *in-place* transformations. A scenario which clearly illustrates how such transformations are essential at the early stages of development is provided in (Koehler, et al., 2007).

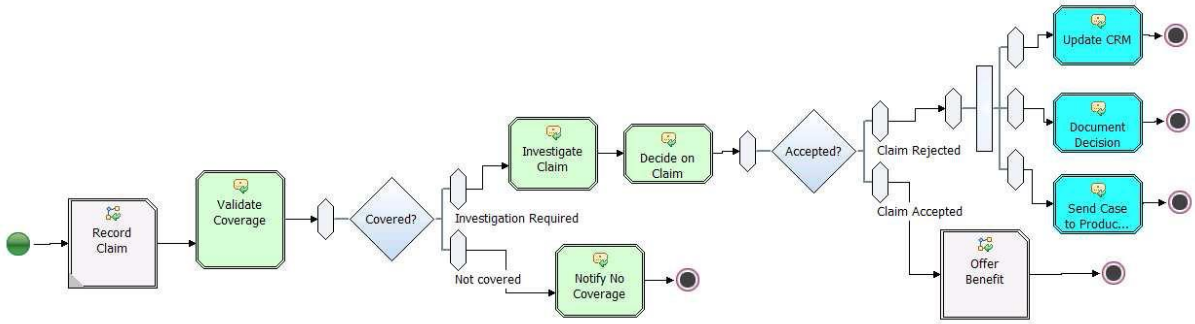


Figure 3. 17: Model obtained from (Koehler, et al., 2007)

Although the model in Figure 3.17, seems to be correct, the fork with the three parallel branches all ending with a different stop node, can cause the termination of the process before the activities on all of these branches would have been fully executed. This is usually the case when the execution of one of the branches finishes prior to the rest. It would be the first to reach its stop node and terminate the entire process. This behaviour is not usually desired and it is in fact one of the anti-patterns which has been identified in (Koehler & Vanhatalo, 2007). Ideally the modeller should have joined all the branches with a join and added a single stop node after this join. Although, with this modification the model is guaranteed to finish all its executing process flows before terminating, the model would still not reflect the underlying implementation of the process. In BPEL, a process must always have just one single stop node. This would terminate the process and return the required output. Thus the modifications to the actual model would be carried out such that besides adding a join to the parallel branches, the other two process flows projecting from the two decisions in the process are merged together with the output of the join, as shown in the Figure 3.18, *'link-style'* BPEL code (Koehler, et al., 2007).

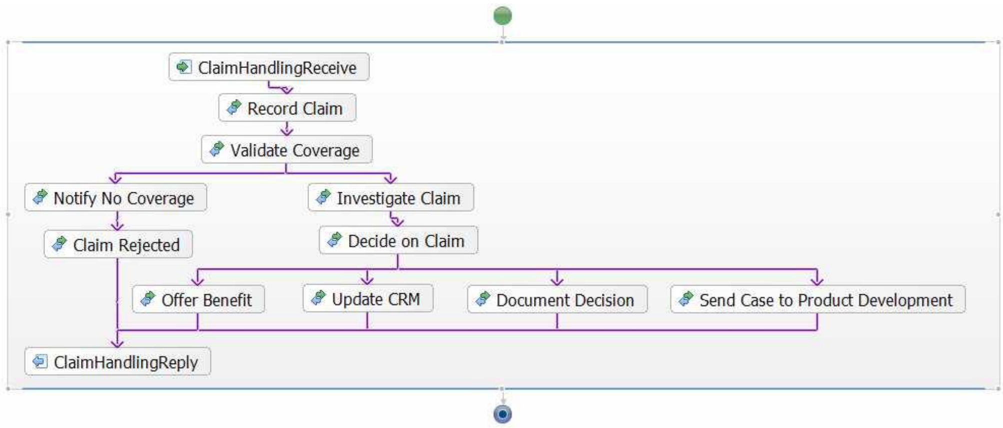


Figure 3. 18: Correct *'link-style'* BPEL code generated for the model in Figure 3.17 (Koehler, et al., 2007)

However as noted in Figure 3.17 and 3.18, two different process models, one developed by the business analyst and another generated at the IT level, would be produced. This is not intended in Business-Driven Development. An abstract representation of the required process must be refined to obtain a more concrete representation and not modified. Thus although it is possible to use some model synchronization techniques (Giese & Wagner, 2006) to update the model produced by the business analyst, in Business-Driven Development the required transformations should be carried out at the modelling phase such that the appropriate model, from which the correct BPEL code is generated, is produced first (Koehler, Hauser, Küster, Ryndina, Vanhatalo, & Wahler, 2006).

To facilitate the application of such transformations, IBM has recently created a *model transformation framework* (Koehler, et al., 2007) for its modelling tool, IBM WebSphere Business Modeler. Their main objective was to create a framework with which different types of model transformations can be implemented. They give in-place transformations special attention, as they provide the required volatility to undo transformations that were applied incorrectly. Moreover such transformations also allow rapid execution such that an immediate feedback is returned to the user to allow him to decide whether to persist the modified model or not. They want the framework to fully integrate the transformations as part of the modelling tool such that the user would view them as normal editing commands and rapidly perform transformations with just a few mouse clicks. However, users cannot define their own transformations and they cannot compose composite transformations.

Some of the transformations which were implemented using this transformation framework include: automatically re-ordering of branches as a horizontal, non-destructive, semantics-preserving transformation to improve the visual representation of the model; sub-process replacement as a horizontal, destructive transformation; cycle removal as a vertical, destructive transformation to construct well-structured cycles; joining and merging of stop nodes as horizontal and vertical, destructive transformations.

3.5. Quality Assurance

In Business-Driven Development, executable code is usually derived automatically from the business process model. Although this reduces the development time of such systems, it also introduces new responsibilities to the modeller. Models must reflect the underlying implementation and moreover they should be quality assured to ensure that control-flow and data-flow errors are automatically deduced early in the development life-cycle that is at the modelling phase. If not, errors would be propagated to the next phases. Moreover, with good quality models which are faithful to the implemented solution, realistic business measures can be obtained through simulation and analysis. Pre and post conditions should be related to every transformation, to ensure the correct application of these transformations and to ensure the generation of valid models. For instance, if in Figure 3.17, the stop nodes of branches coming from the fork are joined with the stop node of the branch emerging from the decision, then the generated model would lead to a deadlock (control-flow error). This is so since according to the semantics of the exclusive decision, only one of the branches executes and thus it is never true that all the four branches would execute in parallel. This means that, using business processes as direct inputs to business-driven development, rather than simply for documentation and discussion, increases the impact of badly designed models on the final implementation and the actual operational efficiency of the organisation.

Since models represent the behaviour of processes, quality assurance techniques must be applied to detect control-flow errors. Additionally, processes also illustrate the flow of data and thus data-flow errors must also be perceived. A *sound* model (according to (Vanhatalo, Völzer, & Leymann, 2007)) is one which exhibits *liveness* (something good will eventually happen) and *safety* (nothing bad will happen) properties. Thus such models must not have deadlocks and must not lack synchronisation. *Deadlocks* are usually present in processes that wait indefinitely for some other operation to complete, leading to non-terminating programs. *Lack of synchronisation* introduces non-determinism. It is not really an error but might be introduced unintentionally. On the other hand, data-flow errors could be introduced if, for instance, data is not available when needed. Figure 3.19 illustrates two unsound models; the first has a deadlock and the second lacks synchronisation

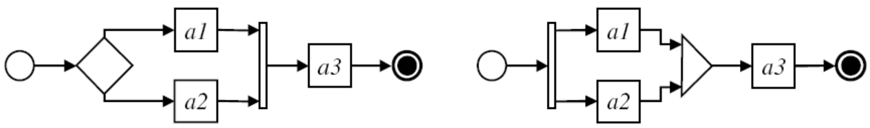


Figure 3. 19: Unsound models - model on the left has a deadlock whereas the model on the right lacks synchronisation (Vanhatalo, Völzer, & Leymann, 2007)

To detect all types of control-flow and data-flow errors, *complete state analysis algorithms* can be employed. Model checkers were used successfully for business process models in (van der Aalst, 2000), (Mendling, Moser, Neumann, Verbeek, van Dongen, & van der Aalst, 2006). Besides verifying that all the possible execution paths satisfy particular properties, these tools also return a trace to indicate where an error was encountered. However, this is only possible through the construction of the entire state space of the process model, which can grow exponential in size and lead to the state-space explosion problem. To mitigate this problem, a technique used in compiler theory, whereby processes are decomposed as a hierarchy of *Single-Entry-Single-Exit* (SESE) fragments is proposed in (Vanhatalo, Völzer, & Leymann, 2007). In this way, rather than model checking the entire process, SESE fragments of the process are checked individually. In the paper, a number of *linear-time control-flow analysis heuristics* were also identified. Thus, if the soundness of the fragment cannot be determined using these heuristics, a model checker would have to be used. This technique was used for IBM's modelling tool and the functionality, to decompose a process into SESE fragments and to apply the heuristics, was incorporated in the transformation framework. In this way they managed to assure the quality of the model in real-time, while the user is producing the model, without any significant delay (less than a second), thus providing immediate feedback and return diagnostic information, to trap errors as early as possible. The observations carried out in (Vanhatalo, Völzer, & Leymann, 2007) (which led to the definition of these heuristics) overlap with the *anti-patterns* identified in (Koehler & Vanhatalo, 2007). By analysing their observations, we were able to identify the most appropriate quality assurance techniques to incorporate in our language.

Other methods that were investigated to quality assure transformed models include the following. In (Küster, 2006), the *systematic validation* of model transformations with respect to termination and confluence was carried out through the verification of a number of criteria. A different approach was adopted in (Varró, Varró-Gyapay, Ehrig, Prange, & Taentzer, 2006) whereby a *Petri net based analysis* method was used to ensure termination. In contrast to the above, in (Küster & Abd-El-Razik, 2006), a number of characteristics were considered and a number of *test cases* were constructed to ensure the validity of transformations.

Since no standards have been developed (Koehler & Vanhatalo, 2007), the question as what makes a business process model, a good quality model, is rather subjective. Besides checking for errors, a number of metrics and criteria were proposed over the years. Modelling tools usually allow users to measure metrics such as ROI or KPI. They also provide analytic functionalities but still they do not address quality requirements. In (Becker, Rosemann, & Uthmann, 2000), these six guidelines to modelling are proposed: *correctness, relevance, economic efficiency, clarity, comparability* and *systematic design*. However, these are not really quantifiable criteria. In (Guceglioglu & Demirors, 2005), the ISO/IEC 9126 Software Product Quality Model are considered and quality measurements are carried out on the *functionality, usability, maintainability* and *reliability*. Still manual intervention is required to carry out the measurements. In (Lange, Dubois, Chaudron, & Demeyer, 2006), an experiment was carried out to investigate how inconsistencies between the static UML diagrams, sequence and class diagram, can lead to misinterpretations and thus errors. The most effective method that has currently been adopted is the identification of patterns and anti-patterns (which overlaps with the control-flow analysis heuristics in (Koehler & Vanhatalo, 2007)). These shall be discussed briefly in the following sub-sections

3.5.1. SESE Fragments & Control-Flow Analysis Heuristics

As mentioned earlier, IBM has opted a technique whereby the model is first decomposed into a hierarchy of Single-Entry-Single-Exit fragments (SESE) and then a number of control-flow analysis heuristics are applied to detect control-flow errors in linear-time. This technique was proposed in (Vanhatalo, Völzer, & Leymann, 2007).

A complex model would be decomposed into SESE fragments as illustrated in Figure 3.20, such that every fragment would have just one single entry point and precisely one exit point.

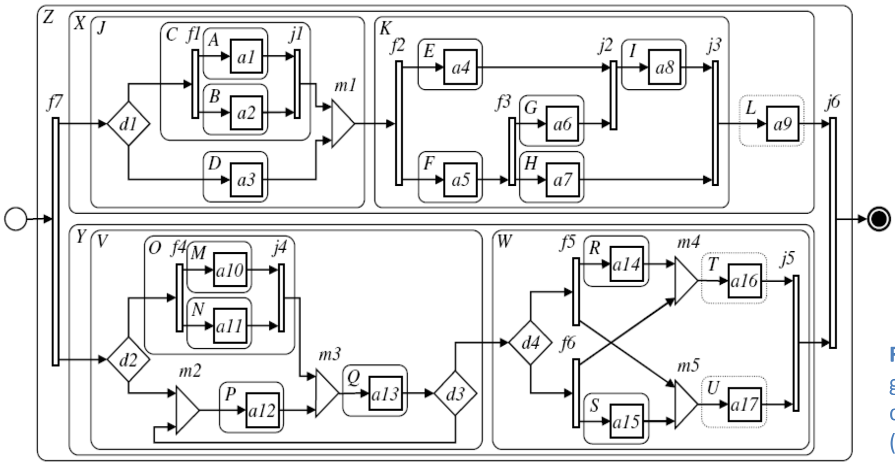


Figure 3. 20: A complex workflow graph representing a process, decomposed into SESE fragments (Vanhatalo, Völzer, & Leymann, 2007)

These fragments are later analysed individually, starting off with the innermost elements. These are classified into four categories, such that, certain fragments are automatically inferred as sound, others as unsound and others, which are more complex, as unrecognizable, in which case, a complete state analysis technique is required. Thus, although this technique is sound, it is not complete, as it cannot state whether a fragment is sound or unsound in all the cases.

An interesting feature of this technique is that both the decomposition and the analysis of the fragments are computed in linear time in terms of the size of the graph representing the model. In this way, it helps to speed up analysis of processes and provide useful diagnostic information to help users fix control-flow errors.

3.5.2. Process Patterns & Anti-Patterns

A set of *workflow patterns* have been defined in (Russell, Hofstede, Aalst, & Mulyar, 2006) and (Russell, Hofstede, Edmond, & Aalst, 2004) with the main objective to identify the types of models that workflow languages and business process modelling languages should support. Using these patterns, the strengths and weaknesses of the available languages can be identified.

The importance and use of *patterns to design behavioural models* was also investigated in other works. For instance, in (Foerster, Engels, & Schattkowsky, 2005), attempts were made to use patterns to capture non-functional aspects such as domain specific quality constraints. In (Novatnack & Koehler, 2004), patterns were used in service-oriented architecture, to facilitate the implementation of business processes. As defined in (Coplien, 2004), patterns have become important to ensure the sound design of software. The author argues that a good pattern is considered to be one that solves a problem and suggests a non-trivial solution, must be a proven concept and it must define some deep system relationships and structure. In this way, patterns have evolved into a "*software engineering problem-solving discipline*" (Coplien, 2004). They originated from the object-oriented paradigm and their use has now been extended to networked and concurrent objects.

Understanding the benefits brought about by the establishment of such patterns to which various models can be mapped to, in (Koehler & Vanhatalo, 2007) a set of *anti-patterns* were defined to measure the quality of process models in an objective manner and to capture some of the most typical modelling errors. Recurring modelling errors were extracted from hundreds of process models which were created between 2004 and 2006 using IBM's modelling tools as well as other tools, and later abstracted to the define anti-patterns. The models that were analysed were obtained from real-world industries such as banking, telecommunications and retail.

One of the most interesting features of this article is that in relation to every scenario and anti-pattern, a solution and a valid pattern is defined. This would help the modeller understand how to avoid such inadequate design patterns and how to produce models of a good quality that are clear, maintainable, usable and comprehensive. The authors investigated bad design practices related to the modelling of the control-flow and the data-flow in the process. Some functionality that is able to take a model as input and return whether it matches one of the anti-patterns or patterns, can easily be developed to check the quality of the models. The control-flow analysis heuristics, discussed in the previous section, use very similar concepts.

For instance, while the process fragment in Figure 3.22 is sound pattern, Figure 3.21 illustrates an anti-patterns which leads to a deadlock; only one of the decision branches executes and thus, it will wait indefinitely at the join, leading to a deadlock.

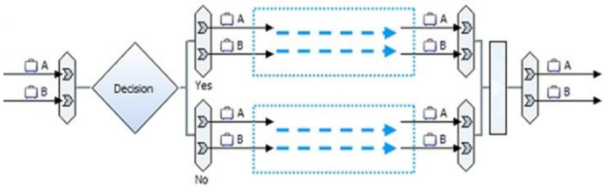


Figure 3. 22: An Anti-Pattern - unsound process fragment that leads to a deadlock

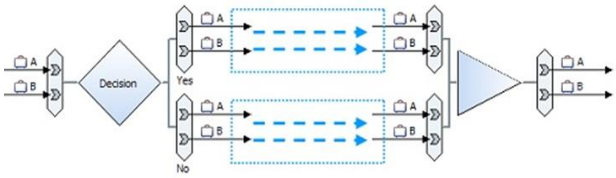


Figure 3. 21: A sound pattern – could be suggested as a solution to the anti-pattern in Figure 3.21

By analysing these patterns, we were able to identify the most important checks that need to be carried out to guarantee models of high quality. These patterns were also useful to help us identify a set of connection patterns that would be helpful to construct readable models of high quality with least amount of effort.

3.6. Conclusion

After reading this chapter, the reader should understand concepts such as Business Process Modelling and Business-Driven Development. The reader must also be aware of the main constructs of IBM WebSphere Business Modeler modelling language and their semantics. Model transformations and their usefulness were also discussed. These were also linked to quality assurance, were various techniques such as the control-flow analysis heuristics and anti-patterns were discussed.

The next chapter introduces our embedded domain specific language which we have designed and developed to be able to model, transform and quality assure business processes in Business-Driven Development. Our language tries to capture the semantics of IBM's WebSphere Business Modeler Advanced modelling language.

Chapter 4

Embedding IBM's WebSphere Modelling Language in Haskell

4.1. Introduction

To assist business analysts in the construction of good quality business process models, we present a domain specific language embedded in the functional language, Haskell (Jones S. P., 2003). With this language, we aim to capture precisely the domain semantics of IBM's WebSphere Business Modeler[1] modelling language (in particular IBM WebSphere Modeler Advanced version 6.0.2[2]), such that in a couple of lines a non-IT technical person would be able to construct a complex model which is type safe, which contains all the details required by IT technical developers and which can easily be interpreted, transformed and quality assured; thus, producing a model from which the appropriate executable code can be derived. The language is simply a set of Haskell modules and the business process models produced by the modeller are simple Haskell functions defined within a Haskell module. In this way, programs written using our embedded language would be compiled and executed using the Haskell compiler. Various features of Haskell resulted to be essential for our language, primarily to embed our own domain-specific terms and type system. Certain construction errors are trapped as early as compile-time and hence they are not allowed to propagate to the other succeeding development stages. To select the appropriate primitive components and combinators for our language, the approaches taken by both WebSphere Business Modeler (WSBM) modelling language and Business Process Modelling Notation (BPMN) (OMG, 2008) were analysed.

Section 2, illustrates how the basic modelling elements are represented as functions or combinators in our language, together with other basic language features. Following this, in Section 3, another important concept is handled. It indicates the data types that are built-in the language and it explains how modellers can easily define and add their own data types. The approaches adopted to embed our domain-specific terms and type system are discussed in Section 4. Following this, the use of type classes in our language is reviewed in Section 5. Three other important techniques used in our language, include the tagging of processes fragment as sub-processes, the use of connection patterns and the possibility to define parameterized models. These are respectively discussed in sections 6, 7 and 8. The final section, before the conclusion, investigates other related work and compares these works to our approach.

Note that a tutorial on our language is also available as Appendix A. A number of sample models defined in our language and in IBM's tool are discussed and analysed in Chapter 6 and in Appendix B. These sample models are also provided with the language in the attached CD.

[1] http://www-306.ibm.com/software/integration/wbimodeler/

[2] http://www.ibm.com/developerworks/websphere/zones/businessintegration/roadmaps/modeler/roadmap_advanced.html

4.2. Basic Modelling Functions & Language Features

A business process model is essentially a collection of basic and complex modelling elements connected together in a particular order with the aim of representing some particular behaviour. Since the main objective of our embedded language is to capture precisely the semantics of IBM's WebSphere Business Modeler[1] modelling language (in particular IBM WebSphere Modeler Advanced version 6.0.2[2]), we tried to maintain the same functionality, modelling elements and way of reasoning as that of IBM's language (see Section 3.3.2), with additional type safety properties and other beneficial features. Since BPMN (OMG, 2008) is the most recent notation whose aim is to capture the features of different modelling tools such that one standard notation would be used, the approaches adopted by this notation were also considered to ensure that the right design decisions are taken for our language. Some of the design issues that were encountered while the language was being developed are discussed in the following sections.

It is important for the reader to keep in mind that a process can be decomposed into a number of connected process fragments or sub-processes. In this way, the modelling elements can be considered as basic combinators, which combine process fragments to create a new composite process fragment. The reader must distinguish between the terms *process fragment* and *sub-process*. The main difference is that a *process fragment* is either a function which abstracts away a collection of interconnected modelling elements or a single modelling element. On the other hand, a *sub-process* is a process fragment, or a group of process fragments, which are packaged into a single process with the required start node and stop nodes. Moreover, different from a common process fragment, a sub-process is assigned a name and can be handled as a single modelling element.

The reader should also be aware that *tasks* and *sub-processes* are collectively referred to as *activities*, whereas *decisions*, *merges*, *forks* and *joins* are collectively referred to as *gateways* or *actions*.

4.2.1. Tasks

Considering that a business process model is effectively a representation of how and when activities are carried out, the most basic and important modelling element is essentially a task. A task should be easy to define, comprehensible and re-usable. Since the main purpose of a task is to take some input and produce some output, a task can easily be represented as a function. Moreover, by pre-defining its input and output types, the type checker of Haskell, which is strongly statically typed, can easily trap errors at compile-time and check the compatibility of process fragments during the construction of the model.

To guarantee the re-usability of components, a task should not be defined in the context of a specific process but as a pure independent function or activity which given certain data as input, carries out a specific operation and produces some output. In this way, it would be possible to define tasks once and use them in any process. However, what sort of inputs and outputs, might be required?

There might be tasks that can execute without requiring any data as input; others might not produce any output; while others might require one or more data items as input and return one or more data items as output. In any case, the task, which is defined as a function, should have all of its input and output types explicitly defined. Thus, if no data is required, the built-in unit type in our language, `NoData`, can be used to illustrate that a task does not have any data inputs or outputs (as illustrated in Figure 4.1).

[1] http://www-306.ibm.com/software/integration/wbimodeler/

[2] http://www.ibm.com/developerworks/websphere/zones/businessintegration/roadmaps/modeler/roadmap_advanced.html

If some data is required then the actual data types should be defined (as illustrated in figures 4.1, 4.2).

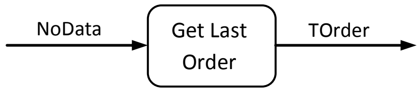


Figure 4. 1: A task named ***Get Last Order***

It does not require any data as input and thus the data type of the input is set to **NoData**

It produces an Order as output and thus the data type of the output is set to **TOrder**

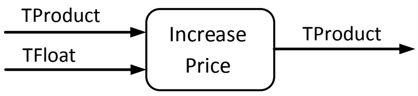


Figure 4. 2: A task named ***Increase Price***

It requires a Product and a Float as input and thus the data type of the input is set to **(TProduct, TFloat)**

It produces a Product as output and thus the data type of the output is set to **TProduct**

However, although these tasks are initially defined as independent atomic activities whose precise context of use would not as yet be known, the modeller must still be aware that the inputs of this task shall be obtained from another process fragment and the output of this task shall be passed on to another process fragment. For this reason, the definition of the task can be defined as illustrated in Figure 4.3 and Listing 4.1, and Figure 4.4 and Listing 4.2.

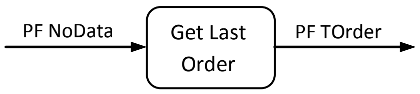


Figure 4. 3: Extending task in Figure 4.1, by adding constructor PF

```
tGLO = task :: PF NoData -> PF TOrder
```

Listing 4. 1: A possible definition for task in Figure 4.3

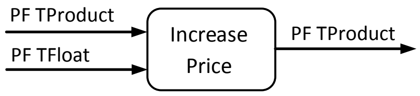


Figure 4. 4: Extending task in Figure 4.2, by adding constructor PF

```
tIP = task :: (PF TProduct, PF TFloat) -> PF TProduct
```

Listing 4. 2: A possible definition for task in Figure 4.4

Issue 1: *Minimum and Maximum values defined for Inputs and Outputs*

All the inputs and the outputs must be defined explicitly such that if more than one data item of the same type is required then different inputs or outputs should be defined for each one (as illustrated in Figure 4.5 and Listing 4.3).

Figure 4. 5: Task *Compare Prices* has two inputs of the same data type

```
tCP = task :: (PF TProduct, PF TProduct) -> PF TFloat
```

Listing 4. 3: Defining the task in Figure 4.5

This is in fact different from WebSphere Business Modeler. In the tool, the user is allowed to take a specific input or output and specify the minimum and maximum amounts of expected inputs or outputs of that specific type. These are defined as properties of the task and are thus not visible on the diagram. This makes the diagram less readable and difficult to comprehend and reason about. For this reason, we have opted to explicitly define the types of every single input and output, such that a task can only execute if all of its inputs are provided and on termination all of its defined outputs are produced. This makes the model more intuitive and easier to reason about.

Issue 2: *Tasks with different possible Inputs and Outputs*

In IBM's WSBM modelling language as well as BPMN, users are allowed to define tasks as illustrated in Figure 4.6.

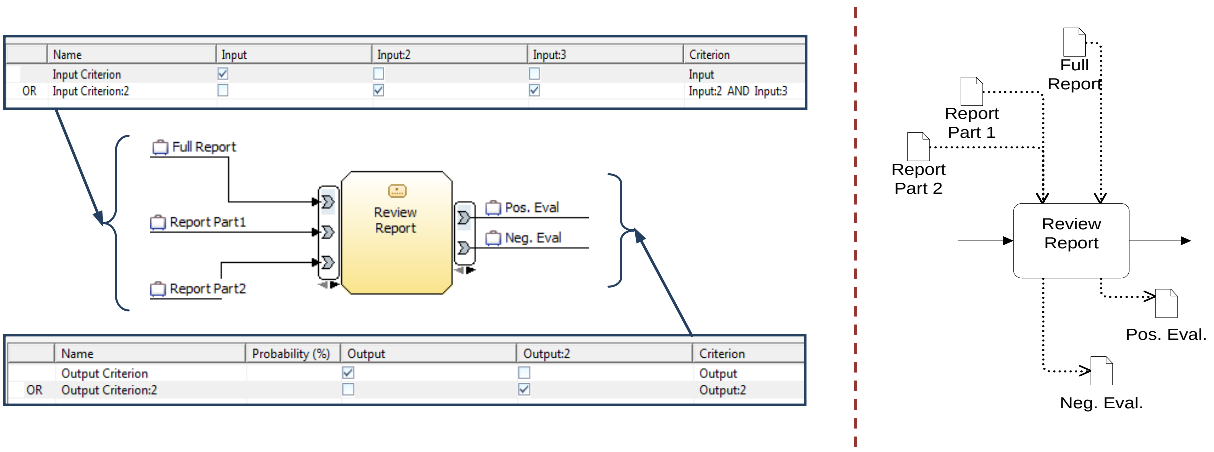


Figure 4. 6: The process fragment on the left was created using activity form in WebSphere Business Modeler (WSBM), while that on the right is depicted using Business Process Modelling Notation (BPMN). They illustrate the same behaviour in different notations. The task *'Review Report'* can have different possible inputs and outputs

The task defined in the Figure 4.6 illustrates that as input, it requires either a data item of type `Full Report` or two data items, one of type `Report Part1` and another of type `Report Part 2`. As output it would either generate a data item of type `Pos. Eval` or of type `Neg. Eval`. In such cases, a mapping is usually defined internally to indicate how the different inputs are mapped onto the outputs and thus what outputs are produced when specifically typed inputs are provided. If the mapping is not explicitly defined, the outputs are returned non-deterministically. In IBM's tool this is defined using input and output criteria while constructing the model using activity form (as illustrated in Figure 4.6). However, such diagrams are quite ambiguous and not so readable. Moreover, as defined in Part1 of (Koehler & Vanhatalo, 2007), it is usually more difficult for a user to specify such criteria. If these criteria are defined incorrectly, new bugs and errors can be introduced. These criteria make tasks less reusable and more difficult to control the correctness of the produced model when these are combined with other process fragments. Consider the process fragment in Figure 4.7.

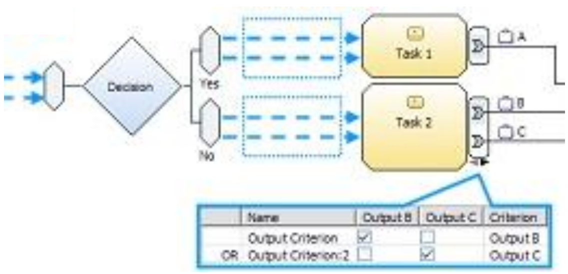


Figure 4. 7: Process fragment with output criteria (Figure 30 in Part 2 of (Koehler & Vanhatalo, 2007))

In Figure 4.7, the task produces only one of the outputs. Thus the task that is connected to it must not expect both outputs as its inputs. Instead it should either make use of one of the outputs of the previous task or have an input criteria claiming that the task can start executing if it receives any one of the data. If not, the second task would wait indefinitely for the other input and thus lead to a deadlock. Similarly, if the outputs of the task had to be connected to the outputs of the process, then the process would have to define an output criterion. Other anti-patterns related to the use of input and output criteria are explained in Part 2 of (Koehler & Vanhatalo, 2007).

To avoid all this, we assume that all the defined inputs are required and all the defined outputs are produced. Thus if the output type of a task is equivalent to the input type of another, then the tasks can safely be connected in sequence without the possibility of introducing deadlocks or other errors. Since Haskell is a pure functional language, we can also guarantee that given specific inputs, the task would always return the same output.

However, if a user still wants to specify that a task can take different types of inputs and produce different types of outputs, then he would have to define this by explicitly defining different tasks. Actually, rather than two tasks, they can be considered as two interfaces of the task, such that if in the future a set of attributes are added to the task, then the two interfaces would be able to refer to the same set of attributes and then define their own input and output types. For instance, the example in Figure 4.8.

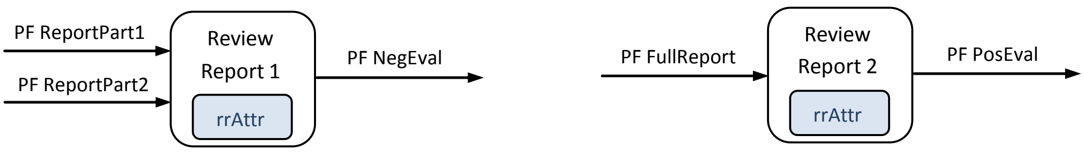


Figure 4. 8: Considering the task in Figure 4.6 and assuming that when data of type (*ReportPart1, ReportPart2*) is received, a data of type *NegEval* is produced, and when data of type *FullReport* is passed as input, a data of type *PosEval* is generated, then in our language tasks *Review Report 1* and *Review Report 2* would have to be defined. If the original task *Review Report* had attributes, then *Review Report 1* and *Review Report 2* would share the same attributes (*rrAttr*). Thus, *Review Report 1* and *Review Report 2* would serve as interfaces to the original task

Choosing the Right Syntax to Define a Task in Our Language

Defining tasks in the following manner is rather intuitive:

```
t1 = task :: PF a -> PF b
```

where `task` is a primitive built-in component defined as a function (or rather a combinator, because it combines a process fragment attached the its input with a process fragment attached to its output), and `a` and `b`, respectively the data types of the inputs and outputs.

However, since the user would want to specify the name of the task, another input argument is required, such that

```
t1 = task  "task1" :: PF a -> PF b
```

For someone who has some knowledge of Haskell, it should be obvious that `:: PF a -> PF b` is part of the type signature to infer and specify the input and output types of the function `task` and `PF a` and `PF b` are parameterised polymorphic types. When creating models using the provided set of primitive modelling elements, a structure using internal constructors would be defined such that further interpretation and analysis can be carried out. Types in Haskell are not first class objects, and thus they are only defined and used to carry out compile time checks, and in this case ensure the appropriate combination of the modelling elements while constructing the model. However, to be able to carry out certain analysis on the internal representation of the models, the input and output types are in certain cases essential. For this reason, rather than specifying the input and output types of the task in the type signature, the user is allowed to specify these as arguments of the function `task`. In this way, the input and output type of the task would still be inferred and thus compile-time checks during construction of models are still possible. Besides this, it would also be possible for the actual types to be stored within the internal abstract representation of the model. This means that both the name and the types of the task would be accessible during analysis. Example:

```
t1 = task "task1" (bvTString :-> bvTString)
```

In this case a task `t1` with the name `"task1"` is defined to have just one basic value of type String as an input and one output of the same type. Note that the infix operator `:->` is used to separate the inputs from the outputs. If the user wants to specify multiple inputs or outputs then this should be specified in the form of a tuple. `bvTString` is a built-in first class object representing the basic type String (these are discussed in Section 4.3). Defining tasks in this manner, independent of the actual context or process where they would be used, it would be possible for the modeller to focus on the data flowing through the task and the required behaviour of the task. Thus, the type `PF a` is abstracted away, and in no way, the task at this level is seen as a combinator of two process fragments. This would ensure proper definition of the task's purpose, as a reusable, atomic component.

In this way, the user would not have to define input arguments for the function `task` as well as a type signature. Thus the user would not have to remember the correct syntax to write a type signature and the different syntax to define a type (that is starting with an uppercase letter) and a function or built-in primitive component (that is starting off with a lowercase letter). If, on the other hand, the user is familiar to Haskell syntax, the use of the infix operator `:->` would still be obvious for him; rather than using `->` with types, `:->` is used with first class objects representing types.

Since the actual name of the task is defined as an input argument of the task, then there is a possibility that the user uses the same name for more than one task. This could lead to some issues later on when the model is interpreted and analysed, since internally one task is distinguished from another, through its name. To elevate this issue, a function `isNameUsed` is provided to allow the user to check whether a particular name has already been used within a particular process fragment.

Other Preliminary Approaches

A different approach which was initially considered was to define the task as a component having just one input argument and one output argument, that is one input pin set and one output pin set, in a similar way as tasks are defined in WSBM modelling language (refer to Figure 4.9).

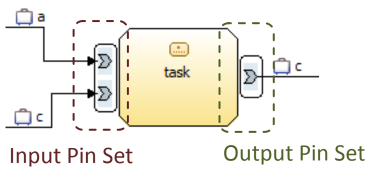


Figure 4. 9: A task defined in WSBM. Note *Input* and *Output Pin Sets*
Different pins in the sets indicate different inputs and outputs

If the task has multiple inputs or multiple outputs as in Figure 4.9, then these would be modelled as pins within the pin set. Thus, considering that the input and output types of the task are defined through a type signature, the definition of the task would be similar to:

```
t1 = task :: PF (a,b) -> PF c
```

However, considering the true semantics of such a definition, this would mean that this task should be connected to a process fragment which produces precisely outputs of type `a` and `b`. Else, the language would have to provide some functionality to be able to group and ungroup inputs (or rather pins), such as:

```
grp2   :: (PF a, PF b) -> PF (a, b)
ungrp2 ::  PF (a, b) -> (PF a, PF b)
```

This means that, for task `t1` to be connected to two other tasks `t0a` and `t0b`, such that the first input is obtained from task `t0a` and the second input is obtained from task `t0b`, as illustrated in Figure 4.10,

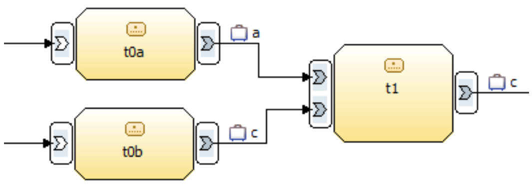


Figure 4. 10: Process fragment created in WebSphere Business Modeler

the user would have to use the function `grp2` and combine the tasks as follows:

```
pf x y = t1 (grp2 (t0a x) (t0b y))
```

In this way, the type of the new process fragment `pf` (which is automatically inferred by the compiler) would be

```
pf :: (PF a, PF b) -> PF c
```

The need to group and ungroup pins representing inputs and outputs can result to be quite tedious especially when tasks have several inputs and outputs or when inputs need to be obtained from various other pre-evaluated tasks, for instance Listing 4.4.

```
t1 = task :: PF a -> PF (b,c)
t2 = task :: PF (b,c) -> PF (a,c)
t3 = task :: PF (a,b) -> PF d

pf x y =  t3 (grp2 ot2a ot1b)
              where (ot1b, ot1c) = ungrp2 (t1 x)
                    (ot2a, ot2c) = ungrp2 (t2 y)
```

Listing 4. 4

In this case, task `t3` requires as input, the first output of task `t2` and the first output of task `t1`. However, since these tasks produce more than one output, first these values need to be ungrouped and then passed on to task `t3`. If, on the other hand, pins and pin sets are ignored and the same process fragment had to be defined using the first approach which we presented earlier, the code in Listing 4.5 would be required.

```
t1 = task ::  PF a -> (PF b, PF c)
t2 = task :: (PF b, PF c) -> (PF a, PF c)
t3 = task :: (PF a, PF b) -> PF d

p x y = t3 (ot2a ot1b)
            where (ot1b, ot1c) =  t1 x
                  (ot2a, ot2c) =  t2 y
```

Listing 4. 5

In this way, the user would not need to include any other function (such as `grp2` and `ungrp2`) except the tasks which he wants to connect. This would also elevate certain implementation issues which arise when such types of inputs are checked at compile-time.

Local and global tasks

If tasks are defined as functions in a module, then they are easily accessible to anyone who has access to the module and thus can be considered as global tasks. If the business process model is defined in another module, the modeller would have to import the module where the task is defined.

If, on the other hand the task, is defined locally within the function representing the business process model, then the task would only be accessible within the function itself.

4.2.2. Gateways to Handle Control and Data Flow in the Process

In BPMN, the control flow and data flow within the process are represented using different types of connectors. These include sequence flow connectors to represent the control flow and associations to represent the data flow. Thus, in the following example (Figure 4.11), it can be noted that gateways always have control as input and output. The data flow between elements is illustrated using associations and data objects.

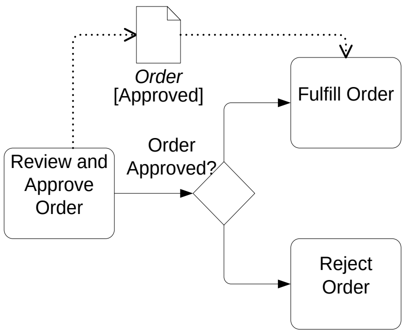
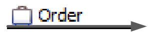


Figure 4. 11: An exclusive decision in BPMN illustrating data flow (the data item *Order*, which is connected to other activities using dotted lines, known as associations) and control flow (indicated with solid line connectors, known as sequence flow connectors)

The main aim of such an approach is to allow the user to either view the process in terms of its control flow or view the model in terms of its data flow or view both flows on the same model. This would help the user to abstract away from certain details and focus on others. With their approach it is rather easy to switch between the two views; to view the control flow only, then all the associations in the model would be ignored; if on the other hand, the user wants to view just the data flow, then all the sequence flows would be ignored.

In WSBM, the data and control are illustrated as one flow in the same model. Thus, if a connector does not have any associated data as the following,

then the connector in that case is illustrating control flow. If, on the other hand, some data is associated such as



then, although the connector illustrates a data flow, this flow also has an inherent control, which would allow a task to initiate its execution, given that the appropriate data is passed on as input. To allow the user to focus on the activities in the process and the data flow, the user can define the model using activity form, whereby the only modelling elements illustrated in the model are tasks and sub-processes. The control flow is handled through input and output criteria (as discussed in Section 3.3.2.4). To specifically analyse the control flow, the user using WSBM is encouraged to use gateway form. However, still, using such a form, both the data and control flow would be visible. Using activity form, various errors are usually introduced when input and output criteria are defined. Thus, although activity and gateway form are available in the language, to help users to focus on specific concepts of the model, users usually get confused when they should use these forms and in most cases, since branching and joining points are more explicitly defined, they end up using and preferring gateway form.

For our language, we decided to force the user to explicitly define the control and data flow in terms of gateways (similar to the gateway form in WSBM). This would prevent ambiguities and errors brought about with the definition of input and output criteria. If users want to abstract away from certain details, then process fragments should be defined as functions or packaged as sub-processes. Users can also make use of the defined connection patterns to easily connect processes and abstract away the implementation details (see Section 4.7 for more details about connection patterns in our language).

Although the approach which is used in BPMN is appropriate for a user to easily abstract away unnecessary details and view just one type of flow in the model, the actual data outputs of the task are never explicitly and fully defined. Only the required data items are illustrated. Moreover, it is easier for the user to introduce new errors. For instance the user might associate some data items to a task, which when considered in terms of the corresponding control flow, it is not really possible for that data to reach that particular task. After all, for an activity to execute in a process, it requires both the data and control. In other words, as illustrated in Figure 4.11, modelling the flow using this approach, diagrams would become more cluttered as two separate flows would have to be defined and the system would have less control over the defined flow to ensure the appropriate combination of elements.

For this reason, we have decided to adopt an approach similar to that used in WSBM, whereby the control and the data flow are considered as one flow. Thus, a data flow has an inherent control flow which can initiate the execution of a task. The flow illustrates how the data changes as it passed through different activities. It also illustrates how gateways divert the flow. An activity can only have access to the data produced as output by other previously executed activities. Thus, considering the following process fragment (Figure 4.12), where *pf1* and *pf2* represent two process fragments.

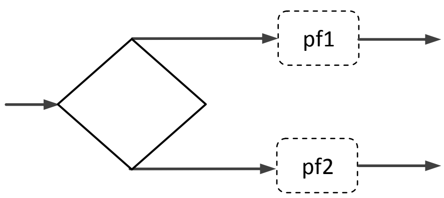


Figure 4. 12

and assuming that some data of type `a` is required to decide which branch should be executed, some data of type `b` is required to execute `t1` and some data of type `c` is required to execute `t2`, then three types of data have to be passed on as input to the decision, that is `(a,b,c)`. Once a branch is selected according to the data of type `a`, the flow is diverted to that branch and the required data is selected and used. The other data flows which are not required by any other activity in the process are terminated with an end node, as illustrated in Figure 4.13.

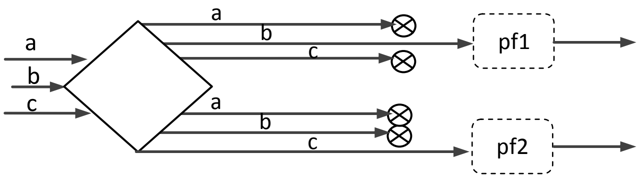


Figure 4. 13: An 2-branch exclusive decision with a flow of type (a,b,c)
Only one of the data flows is maintained as soon as the branch is chosen.
pf1 and *pf2* represent the succeeding process fragments

If this had to be expressed in BPMN then the model in Figure 4.14 would be produced.

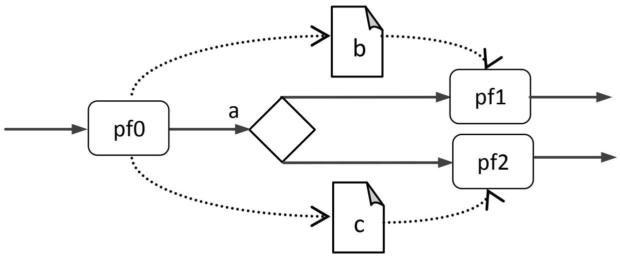


Figure 4. 14: Figure 4.13 in BPMN – *pf0* represents the proceeding process fragment, whereas *pf1* and *pf2* represent the succeeding process fragments

However, although in BPMN this is handled more elegantly as only the required data flows to a fragment, in our language we have a constant control over the type of data that flows through the connectors and thus we can assure that only compatibly typed components are connected. Moreover, we can guarantee that the required input data is always reachable from the other previous elements in the model.

Thus, in our language, the data types of all the incoming branches of a gateway must be equivalent to those of all the outgoing branches. The scope of a gateway is to handle the flow and not to modify the data flowing through it. These constraints make gateways more intuitive and simpler to reason about. For instance, considering the diagram in Figure 4.15

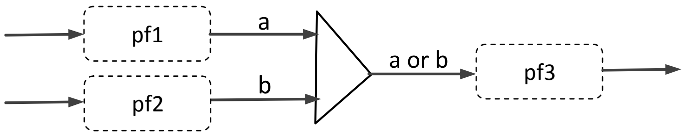


Figure 4. 15

The output type of the merge gateway is only known during the execution of the process and thus it would not be possible to check at compile-time and during the construction of the model, whether the input type of `t3` is compatible with the output of the merge.

Deciding how gateways should be defined

As mentioned previously, the data types of all the incoming branches must be equivalent to those of the outgoing branches. This means that different from a task, multiple inputs or outputs (depending upon the gateway), can be modelled using a list (a list in Haskell must contain elements of the same type). However, this would also mean that the user would have to remember when inputs or outputs should be presented as a list and when represented as a tuple. For this reason, to facilitate the use of our language, we opted to use tuples for all the components. Specific type classes were created to carry out compile-time checks and ensure for instance, that a tuple (and not a single argument) is passed on, that all the elements of the tuple are of the same type and same tuple size etc. (The use of type classes in our language is discussed in Section 4.6). Our language currently supports tuples of up to size 10.

Another design consideration, which we had to make when designing gateways, is how the language should know the number of branches that the user wants. For instance, how many outgoing branches should a fork have? The simplest solution is to provide gateways with specific branches, for instance, `fork2` meaning that the fork has two outgoing branches. However the user would be limited to the provided definitions in the language. A similar solution is to allow the user to specify the number of branches as an input argument, example `fork 2 …` . However, this is not really intuitive; the modeller should focus on the behaviour of the process rather than the how it should be presented. But why should the number of branches be known, at the point when the gateway is defined? Why not allow Haskell's compiler to infer the type automatically? In fact, the best solution, which we thought is most convenient for our language, is to allow the user use gateways without specifying the number of incoming or outgoing branches and then infer the number of branches when this component is attached to others. Obviously, checks through type classes would have to be carried out to ensure the appropriate types for the inputs and the outputs. For instance, in the case of a decision, the number and the data types of outgoing branches must be compatible to the expressions passed on as input. (The use of type classes in our language is discussed in Section 4.6).

Similar to IBM's modelling language, the inputs and outputs can have any number of control connectors added without changing the data type of the input or output.

4.2.3. Events

Events are handled in the same way as in IBM's WSBM modelling language. Thus these include: *start* nodes, *end* nodes and *stop* nodes. Since in our language these are represented as primitive functions, a `start` node is essentially a function that takes a control as input (to active the event) and returns the control to an activity which is connected to it, such that execution within the process would be initiated. The `end` node, on the other hand, takes either a single type of data or a control as input and returns some control as output to indicate the end of a particular flow in the process. Similarly, a `stop` node accepts same input and produces the same output as the end node, with the only difference that it terminates the entire process.

Even though a *start* node does not really require any input (since the user is only interested in its output), this node still needs to be depicted as a function, since it can only pass control to an activity and initiate a flow in a process, when it gets the control from the parent process. Similarly, the output of end and stop nodes must be returned as an output of the process. If these outputs are ignored, the process would end up having dangling outputs. Considering that the final models are analysed from back to front (that is, from the outputs of the function back to its basic inputs), the details of these nodes would be lost.

4.2.4. Repositories

A repository in IBM's WSBM modelling language is used as a store of a single type of data. The data produced by various activities can be stored in the repository. This data is then retrieved and used as an input to various activities. For this reason, a repository can be modelled as a function which takes various inputs of a specific type and produces one single output of the same type.

The repository is also assigned a name so that, similar to a task, while the internal abstract representation of the model is being analysed, different repositories would be distinguished. The only problem with this explicit tagging approach is that the same name can be used for different repositories. To avoid this problem, a function `isNameUsed` is provided for the user to check if the name is already used in the process. Similar to a task, the type of the data, that the repository is meant is to keep, is defined as a first class value that represents the type.

Thus, a repository named "`Products`" that stores data of type `TProduct` can be defined as follows:

```
rProd = repository "Products" biTProduct
```

To use this repository as depicted in Figure 4.16, the code in Listing 4.6 would be required.

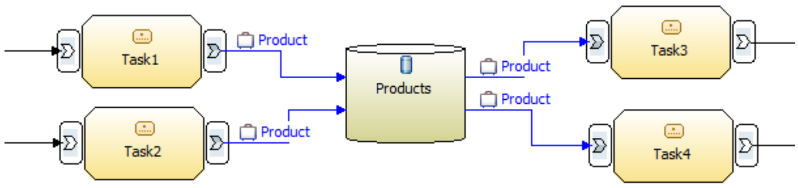


Figure 4. 16: A process fragment (created in WSBM) illustrating how a repository can be filled in and used

```
pf1 x y = let
             orProd = rProd (ot1, ot2)
             ot1 = t1 x
             ot2 = t2 y
             ot3 = t3 orProd
             ot4 = t4 orProd
          in (ot3, ot4, orProd)
```

Listing 4. 6: Defining the process fragment in Figure 4.16 using our language

Global and Local Repositories

A repository in IBM's modelling language can be defined globally or locally. If local, then the activities within the process fill up the repository with data and the data is used by activities within the process itself. If on the other hand the repository is global, then although it stores data produced by activities within the process, this data would still be accessible by other processes, which have access to that repository. However, to ensure that a global repository is always loaded with data before the data is retrieved and used by activities, a process fragment similar to that in Figure 4.17, is always included within processes that make use of global repositories.

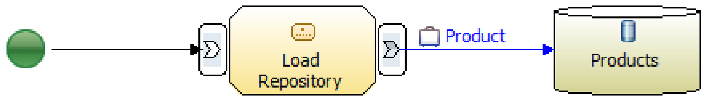


Figure 4. 17: Process fragment to load the content of a global repository before this is used within a process

To mitigate this issue and ensure that a repository is always filled up with data before its data is used as input to other activities, we handle global and local repositories in our language as follows.

If a local repository is required, the repository would be filled in and its output is used by activities within the process. If the defined repository is meant to be global and thus the data saved in the repository must be accessible to other processes, then the output of this repository should be passed out and returned as an output of the process fragment (as illustrate in Listing 4.6). The process which should make use of the data within this repository must accept this as an input. Within the process, activities are allowed to make use of this input as a normal input. If the process would like to add other data to this repository then a new repository would have to be created. This repository would take as input the current contents of the repository (which was passed to the process fragment as an input argument) and all the other required outputs of activities that fill up the repository with new data. If the contents should be accessible by other process fragments, then the output of this new repository must be returned as an output of this process fragment. Although it is different from IBM's modelling language (such that there isn't a true global repository which can be created once and updated by any process), this approach explicitly indicates when and how the contents is being changed by the different processes. Moreover, being modelled as a function, the process fragment which defines or uses the global repository would always produce the same output given a specific input. For this reason, the user would not have to consider any global state or general global memory. This means that we have more control over these changes and thus we can automatically guarantee that a process is never used before it is loaded with data and that it is never loaded unnecessarily such that the contents of the repository is either returned as an output of the process fragment or else it is used by some activity within the process. Thus, in our language, fragments such as Figure 4.17 are not required. Moreover, the user is allowed to abstract away from the idea of a global repository and handle the data stored in this repository as any other data.

4.2.5. Constant Input

In IBM's modelling language, a constant value or data item can be used as an input to an activity, instead of a usual data or control flow from some other previous activity. In our language, this constant is handled as another primitive component, such that, the specifically typed input is represented as a constant in the model.

4.2.6. Sub-processes

Using a compositional approach, a process is essentially a collection of other simpler sub-processes or process fragments connected together to model the required behaviour. Similarly, these sub-processes or process fragments are made up of other fragments which can be decomposed further until the basic atomic modelling elements, such as tasks, are obtained. A collection of modelling elements can essentially be viewed as a process fragment, which is implemented as a function, such that given an input, the required processing is carried out and an output is produced. In this way, the function is used as an abstraction mechanism to allow the user to focus on the behaviour rather than how such behaviour shall be implemented. Moreover, process fragments defined in this manner can be reused at any time by any other function or process fragment.

However, if the user wants to encapsulate a process fragment into a sub-process which has a start node and a stop node, as defined in IBM's modelling language, then an additional function is provided. In this way, a group of process fragments can be passed on to the function such that a start node is added to those inputs which require control, the input and output data types of the new sub-process are identified and the user-defined name is assigned to the block or sub-process. Checks are also carried out to ensure that the process fragments have the appropriate stop nodes to terminate the process. Besides adding the required nodes to ensure the correct construction of such sub-processes, the main advantage of this functionality is that when the internal representation of the model is being analysed, different from other process fragments which are modelled as simple functions, such blocks are explicitly identified. In this way, during analysis, the sub-process can be seen either as one single modelling block or as a collection of basic modelling elements.

By allowing users to explicitly tag the sub-process block, an inherent disadvantage is that users must be careful not to use the same name for more than one sub-process. To avoid this problem, the user can use the function `isNameUsed` to check if the name is already used in the process.

4.2.7. Connectors

In IBM WSBM, the connections between the modelling elements are explicitly modelled as other separate modelling elements. In our language, a connection is inherently inferred when the output of one element is passed on as an input to another. For instance, consider the following sequence of tasks:

Figure 4. 18: Program fragment pf1 in sequence with program fragment pf2

In our language this can either be modelled using the infix function composition operator built-in in Haskell

```
pf = t2 . t1
```

or using the serial connection pattern which we provide in our language (connection patterns are discussed in Section 4.7)

```
pf = t1 ->>- t2
```

4.2.8. Adding Additional Input and Output Control Flows

Although pre-defined global tasks and sub-processes cannot be modified in IBM's modelling language, any number of control inputs and outputs can be added without changing the data flow. This is so, since, as explained earlier, a data flow has an inherent control flow. One of the reasons why an additional input control flow would be required is to prevent the execution of an activity or process fragment before other fragments have executed. If on the other hand, a task, which produces some data as output, needs to pass on just the control to another process, then the data flow must be split up into two flows; data and control.

For this reason, two new components `ctrl_dataFlowCombiner` and `ctrl_dataFlowSplitter` were added to our language and can be used as illustrated in Figure 4.19.

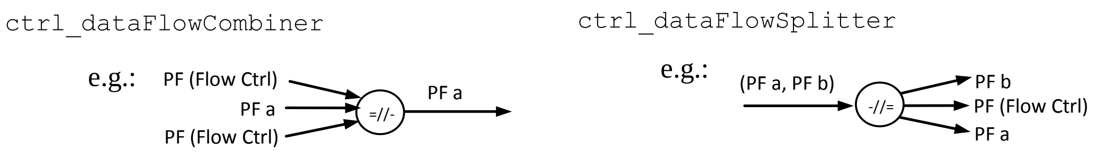


Figure 4. 19: Examples how `ctrl_dataFlowCombiner` and `ctrl_dataFlowSplitter` can be used

While the second is able to take a data flow and split it up into multiple output flows, with one representing the data and the others control, the first does the reverse, such that, given multiple control flows and one data flow as input, the combinator is able to combine these flows into a single data flow. There isn't a limit on the amount of control flows that can be used.

Similarly, if a modelling element has a single control flow, which needs to be replicated, or multiple control flows, which need to be represented as one single control flow, then the following components are also provided in the language to split or combine control flows:

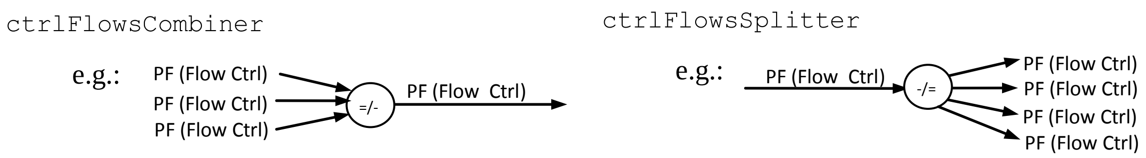


Figure 4. 20: Examples how `ctrlFlowsCombiner` and `ctrlFlowsSplitter` can be used

In this way, pre-defined reusable tasks and sub-processes can be adapted to meet the requirements of the context where they are used.

4.2.9. Specifying Input Source or Output Target Types for a Task

As explained in Section 4.2.1, to ensure the production of reusable tasks, tasks are defined as activities with some data input and some data output. The context where these tasks shall be used is irrelevant and the main focus, when they are defined, is the type of data that is passed in as input and produced as output. Control at this level is never mentioned. In actual fact, if a task does not require any data or does not produce any data, it should be assigned a control input or a control output. However, in our language, when such tasks are defined (not in the context of a process), the unit type `NoData` (rather than control) is used.

When tasks are used within a process, each input must have a specific input source type defined. Similarly, each output must have a specific output target type defined. For this reason, before the actual task is used within a particular context, the input source of every input and output target of every output should strictly be specified. The different inputs and outputs might have different input source and output targets. The only restriction is that the input source and output target types must be appropriate for that specific modelling element (as illustrated in the Table 3.1 in Section 3.3.2.2). Within IBM's modelling tool, the types are specified as attributes of the task. In our language this is done through the use of new components which are provided to the user.

Thus, considering a task (as defined in Section 4.2.1), the inputs of activities can be either a flow or data coming from a repository or a constant data item. The allowed output type is either a flow or a connection to a repository, where the generated output is stored. Assume that the following task has been defined:

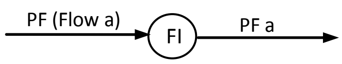
```
t1 = task "task1" (bvTString :-> bvTString)
```

From this definition, it should be noted that task `t1` with the name "`task1`", expects, in terms of data, a *String* as input and produces a *String* as an output. Thus the following type is inferred for this task:

```
t1 :: PF TString -> PF TString
```

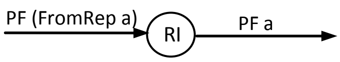
This means that the task is actually expecting a process fragment of type `TString` (that is a String – types are discussed in Section 4.3) as input and shall produce as output a process fragment of type `TString`. Depending upon the context where this task is used, the input source and output target type should be specified by combining to its input and output one of the following built-in primitive components, which, besides specifying the input source and output target type, they also act as type converters, as illustrated in Figure 4.21:

`flowCI`



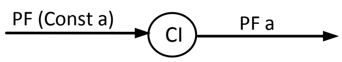
(flow Connector for Input) *to indicate that the input source is a flow*

`repCI`



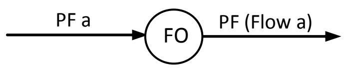
(repository Connector for Input) *to indicate that the input source is a repository*

`constCI`



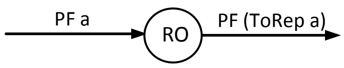
(constant Connector for Input) *to indicate that the input source is a constant*

`flowCO`



(flow Connector for Output) *to indicate that the output source is a flow*

`repCO`



(repository Connector for Output) *to indicate that the output source is a repository*

Figure 4. 21: Components to specify input source and output target types

The input and output types of the typed task are now defined in the following format, `PF c a`; `PF` to indicate that it is a process fragment, `c` to indicate the input source or output target of the process fragment and `a` to indicate the type of data or control which flows through the connector (types are discussed in Section 4.3).

A more interesting case is when a task has no data as input or no data as output. Thus, consider task `t2`. The preliminary definition of the task is

```
t2 = task "task2" (noData :-> noData)
```

whose inferred type would be

```
t2 :: PF NoData -> PF NoData
```

However, within a process, there must be some kind of flow, be it a data flow or a control flow that flows through the activities. For this reason, the only type of input source and output target that is allowed is the `flowCI` and `flowCO` (as illustrated in Figure 4.22).

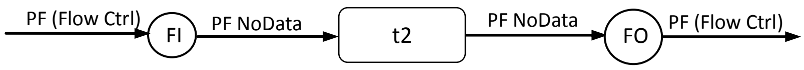


Figure 4. 22: Task `t2` with flow connectors

In this case (Figure 4.22), besides specifying the type of the connector, that is a flow, the type of the element flowing through the connectors is also changed, such that the type `NoData` is converted to `Flow Control` and vice versa.

Thus from the above examples it should be clearly understood that the same pre-defined task, with specified data inputs and outputs, can be re-used multiple times within different processes with different input source and output target types.

Choosing the Appropriate Syntax to Define Input Source and Output Target Types

Since these new components are simple functions, the input and output connection types of a task within a process can be specified in the following manner. For instance, to specify that the input source of task `t1` is a flow and the output target is a repository, the following code is required, to produce a typed version of task `t1` (i.e. `tt1`):

```
tt1 x = flowCO (t1 (flowCI x)
```

or using Haskell's built-in infix functional composition operator `(.)`

```
tt1 = flowCO . t1 . flowCI
```

or using our built-in connection pattern for serial composition `(->-)`

```
tt1 = flowCI ->- t1 ->- flowCO
```

The latter is more readable than the previous definitions since it precisely illustrates the flow within the process and the order in which elements in the process are executed. In the process, the user would then use `tt1` rather than `t1`. However, specifying the input and output connection types in this manner becomes rather tedious especially when the task has more inputs and outputs. For instance, consider the following task which has two inputs (one of type *String* and another of type *Boolean*) and one output (of type *String*)

```
t3 = task "task3" ((bvTString, bvTBoolean) :-> bvTString)
```

Its inferred type would be

```
t3 :: (PF TString, PF TBoolean) -> PF TString
```

If the user wants to specify that the first input should be of type flow and the second should be obtained from a constant and the output should be placed in the repository, then the new typed task, which would be used in the process, is defined in the following manner (see also Figure 4.23):

```
tt3 (x,y) = repCO (t3 ((flowCI x), (constCI y)) )
```

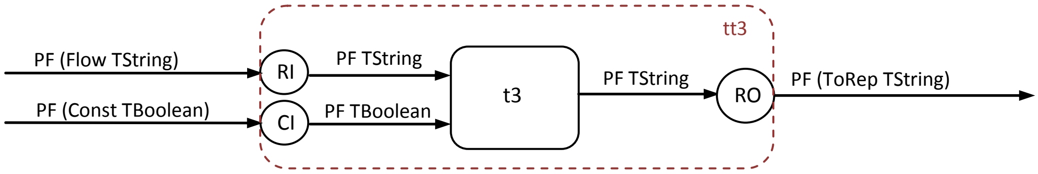


Figure 4. 23: The typed task tt3

In fact, this is the only way how this can be specified. It is not possible to use the infix functional composition operator `(.)` or the built-in serial composition `(->-)`, since the input pair of task `tt3` needs to be split up and passed on as input to functions `flowCI` and `constCI`.

To help users easily specify the input source types and output target types of a specific task two infix operators `(<|)` and `(|>)` are provided. Thus, `tt3` would be defined in the following manner:

```
tt3 = (flowCI, constCI)  <|  t3 |>  repCO
```

If task `t3` is defined locally within the process, then the following definition, which is still readable and intuitive to understand, can be used:

```
tt3 = (flowCI, constCI)  <|  (task "task3" ((bvTString, bvTBoolean) :-> bvTString)) |>  repCO
```

In this way, through Haskell's type system, it is possible to carry out static checks and at compile-time, guarantee that the components which are combined together are of compatible data and connection types.

Input Source and Output Target Types for Various Modelling Elements

Since all the primitive modelling elements in our language are primarily functions, with predefined inputs and outputs, and since these elements are essentially combinators which combine process fragments to form a more complex fragment, then to assure the right combination of fragments, all the elements in the process must have inputs and outputs of type `PF c a`. While the connector type `c` for certain functions is specified by the user (as the case with tasks), for other elements, this type is implicitly defined by the language. The following paragraphs discuss how input source and output target types were defined for other modelling elements.

Similar to a task, if a *sub-process* has some data item as input or output, then the input source and output target types must be specified by the user. If on the other hand, control flows through the sub-process, then the input source and output target would automatically be set to Flow.

Gateways do not alter the data that passes through them. Thus the connection type of the incoming and outgoing branches can vary in a similar way as in IBM's modelling language. However, the type of data must be the same for both the inputs and the outputs.

Events have pre-defined types for their input and output, as illustrated in Figure 4.24:

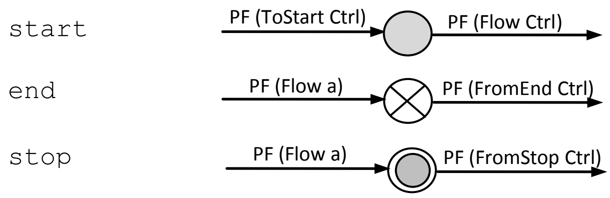


Figure 4. 24: Input source and output target types assigned to event nodes

The `start` node, which initiates activities within the process as soon as the process gets control, has an implicit control input whose connection type is set to the built-in type `ToStart` to indicate that that control flow is specifically addressed to the start node. This would also ensure that no other modelling elements should be connected to its input. This node then returns a control flow. The connection types used for `end` and `stop` nodes are similar, such that, since their main aim is to terminate a particular flow, as input they expect a control or a specifically typed data flow and as output they produce a control, whose connection type is respectively set to `FromEnd` or `FromStop`. These would specifically indicate that that control was produced by an end or a stop node. Even though these nodes output some control, since their connection type is not set to Flow, then no other activity can be connected to them. When the user invokes these events, he can only connect a process fragment to the output of the start node and a process fragment to the inputs of end and stop nodes. The input of start node and the output of end and stop nodes are abstracted away from the user.

To precisely indicate that data is saved or obtained from a *repository*, the flow of data to and from a repository is defined in the following manner (note that in this example it is assumed that the repository has just one input)

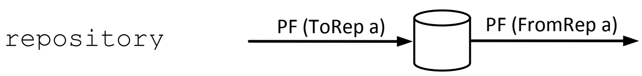


Figure 4. 25: Input source and output target types assigned to a repository that keeps data of type `a` (and assuming that only one fragment is connected to its input)

Thus, the data flowing into the repository is indicated with the type `ToRep a` and the data flowing out of the repository is indicated using the type `FromRep a`. In fact, if an activity expects an input from a repository, the connection type is set to `FromRep a`. Similarly, if an activity produces some data that should be stored in a repository then the connection type is set to `ToRep a`.

The possibility and feasibility of having just a single type `Rep a`, instead of `ToRep a` and `FromRep a` was also considered. However, doing so then the following (Figure 4.26) would be considered a valid process fragment. Thus, for this reason, it was essential to distinguish the flow from and to a repository.

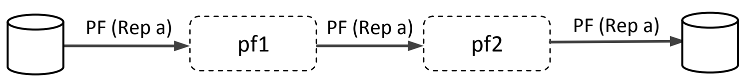


Figure 4. 26: Using Rep a to indicate connection with a repository. This fragment is not correct

Another atomic element in the language is the *constant* primitive component. Thus given an item as a specific instance of type `a`, the `constant` function would return a process fragment of type `(PF Constant a)` as illustrated in Figure 4.27:

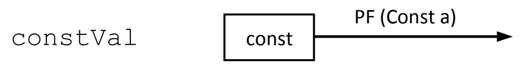


Figure 4. 27: Input source and output target types assigned to constant

In this way, the constant would safely be connected to some input which is expecting a constant of type `a`.

Process Fragments without a Control Flow

It should be noted that data flowing to and from a repository and constant values, do not really have an inherent control flow. If the task such as the following has an input derived from a repository and another defined as a constant, then it is essentially important for the task to have some control flow as an input, such that, as soon as the activity gets the control, it makes use of the provided constant, it retrieves the required data from the repository and only then it carries the required behaviour. In a similar manner, the user should be aware that if an activity does not produce a control or data flow, even if the produced output is stored in a repository, the flow would not be passed on to the other modelling elements and thus it would terminate at that activity, as illustrated in the Figure 4.28.

Figure 4. 28: Task without any input or output flow

Such a task never gets the control and thus, it never executes. To prevent such errors, the user can use the built-in function `hasInFlow` to verify whether the task has a control or data flow as input and thus whether this task has the possibility of being executed. If not, then another function `addInFlow` is provided to automatically add a control flow as an input to the task.

The main issue which was encountered in this case was how this control input should be added. What is the most feasible and intuitive approach, which would allow the user to use this typed task in the process and at the same time keep the integrity of the original definition of the task? The original inputs of the task cannot be modified. The repository and the constant modelling elements do not have a control input and it is not really sensible to add a control input to these components. As explained earlier, for additional control inputs and outputs to be added to activities within the process, two new components were introduced: `ctrl_dataFlowCombiner` and `ctrl_dataFlowSplitter`. Thus the most feasible solution is to ensure a control flow to such process fragments, is to add a `ctrl_dataFlowCombiner` component to the first input, such that, the new type of the task in Figure 4.28 would become

```
ttIP :: ((PF (Flow Const), PF (FromRep TProduct)), PF (Const TFloat)) -> PF (ToRep TProduct)
```

Thus, in this way, tasks can easily be reused. The basic properties of the task are primarily defined, followed by the definition of the different input/output pairs. When the task is then used in the context of a process, the user specifies the input source and output target types, and if addition control inputs or outputs are required then a `ctrl_dataFlowCombiner` or a `ctrl_dataFlowSplitter` with the required number of control flows and the necessary data flow is added to the input or output respectively. This illustrates that tasks defined in this manner are rather flexible and they can easily be reused and adapted to suit the context of any process.

Inferring the types automatically

For the user to easily specify the connection types for the inputs and outputs of the modelling elements, the infix operators `(<|)` and `(|>)` are provided and can be used as illustrated previously. Using these operators, the definition become more readable and easier to reason about and comprehend. However, it is still rather tedious and time consuming for a modeller to specify the connection type for every single input and output. In most of the cases, these connection types can easily be inferred automatically by our embedded type system (which shall be discussed in great detail in Section 4.4).

Thus, if the user wants to emphasize the source or the target type of any of the inputs or outputs of an activity, then he should precisely define the types, such that during construction, only components with that specific connection type and data type are allowed to be connected to the modelling element. For instance, if the task expects a data item of type `a` to be retrieved from a repository, then the user cannot connect a task with the same output type but with a connection type set to a Flow. If on the other hand, the user does not necessarily need to specify the input source or output target type of an activity, then as soon as a repository is attached to one of its inputs, the data type is checked and the connection type of the input is automatically inferred. Similarly if two tasks have compatible types and they are put in sequence (that is the output data type of the first task is equivalent to the input type of the second) then it is not really important for the connection type to be specified.

Thus, assuming Listing 4.7,

```
t1 = task "task1" (bvTString :-> bvTString)
t2 = task "task2" ((bvTString, bvTBoolean, bvTString) :-> bvTBoolean)
r1 = repository "rep1" bvTString
```

Listing 4. 7

the process fragment in Figure 4.29,

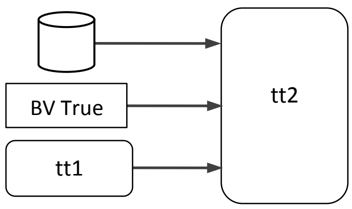


Figure 4. 29

can be defined as in Listing 4.8 by using the serial composition (->-) and parallel composition (`parC`)

```
tt1 = t1 |> flowCO
tt2 = (repCI, constCI, flowCI) <| t2
pf  = parC (r1, const (BV True), tt1) ->- tt2
```

Listing 4. 8

In Listing 4.8, just the connection types of the output of task `t1` and the inputs of task `t2` have been specified such that, the types of the elements that are put in parallel and later on in sequence with task `t2`, match perfectly.

If the lazy serial composition function `(->>-)` is used, then the following would still be valid

```
pf = parC (r1, const (BV True), t1) ->>- t2
```

In this latter case, checks at compile-time are carried out to ensure that the data types are compatible and if so, the connection type is inferred and the required components are added. Thus, this would be equivalent to

```
pf = parC (r1, const (BV True), t1) ->- parC (repCI, constCI, id) ->- t2
```

whereby the `repCI` component is added to the first input of task `t2`, `constCI` is added to the second input, and nothing is added to the third input. In the latter case, the connection type is not explicitly defined as it is inherently obvious that when two activities with compatible types are combined in sequence, then the connection type is a flow.

Thus, whenever the user does not want to specify the input source and output target of the activities, then he can simply make use of the lazy serial composition function `(->>-)` instead of the conventional serial composition function `(->-)` and allow the system to check the data types of the connected components, infer the connection type and add the required component to specify the connection type automatically. In this way, this function helps the user to easily compose components within the process, allow the system to check the compatibility of the types at compile-time and reduce the amount of code is required to define processes. This would make the definitions more readable and easier to reason about and comprehend.

Other Preliminary Approaches

To indicate the connection type, in our final implementation, we use a polymorphic parametric type of the form `c a`, where `c` is the connection type and `a` is the data type flowing on the connector such that a process fragment would have an output of the form `PF (c a)`. Moreover, in the final approach which we adopted, the connection type (be it a flow, repository or constant), is explicitly defined through data constructors and is visible in the internal abstract representation of the model. Thus, it is possible to identify the connection type even when the model is being interpreted and analysed. This was in fact one the disadvantages of one of the preliminary approaches which was considered.

In a previous approach, the specific connection types of the inputs and outputs of the task were immediately defined with the initial definition of the task. For instance

```
t1 = task ::  PF (Flow TString) -> PF (Flow TString)
```

rather than

```
t1 = task "task1" (bvTString :-> bvTString)
tt1 = flowCI <| t1 |> flowCO
```

Using this approach, the same task can only be used within other processes, if the input source and output target match those of the original definition. Moreover, these connection types were not explicitly defined in the internal representation of the model and thus the connection type was not accessible during analysis. In the first approach, a pair of the form `(a, c)` was used instead of the parametric type `(c a)`. Thus the process fragment that produces a flow of type String as output, would be defined as `PF (TString, Flow)`. The main problem with this type is that it is misleading, especially for users who are familiar with Haskell's syntax. Although the process fragment is meant to generate one output, since it is represented as a pair, it is more likely to represent a pair containing an element of type String and an element of type Flow.

4.3. Managing data types

In IBM's modelling languages, users can either use the built-in basic types or define their own complex types, such as business items (discussed in Section 3.3.2.2). These business items can contain a variety of attributes and are usually very specific to the context where they are used. These types make processes more readable and easier for the domain expert to reason about and understand. In the modelling tool, users are allowed to specify these types at any time and use them to define as types for any process within the project.

Thus, within our language, users are allowed to define their own types as a new type in the host language, Haskell. For instance, if the user wants to create a business item to represent an `Order`, then this can be defined as follows:

```
newtype Order = Order ()
```

In this case, the internal attributes of the business item are not defined. To allow the definition of other attributes then it is possible by representing the type `Order` as a Haskell record. The attributes would then be specified as fields within the record such as

```
data Order = Order {custID  :: String
                   ,orderNo :: Long, …}
```

However, due to the simplicity of certain models, the first example is frequently used to define these types.

Since these are defined externally, the system still needs be aware of these new types. Thus, once the type is defined, the user is expected to define this type as an instance of the provided type class `ComplexType`. This user must also indicate that this is a business item, as illustrated below

```
instance ComplexType (BI Order)
```

The class `ComplexType` is a built-in class which collects all the user-defined types. Besides, business items, other complex types such as business services or business service objects might be defined (see IBM's WSBM user manual for more details). For this reason, to be able to identify the different types, built-in parametric types such as `BI a` are provided such that the user would be able to specify that the type `Order`, for instance, is a business item. In this way, besides allowing users to freely define their own types, the system would still be aware of these new types and thus would have full control over the types used in the definition of the models.

Besides complex types, the language also provides the same basic types provided in IBM's modelling language. These include Boolean, Byte, Date, DateTime, Double, Duration, Float, Integer, Long, Short, String, Time (for more details refer to the *Reference* section in IBM's WSBM user manual). To distinguish these types from other complex types, another parametric type `BV a` (for Basic Value) is used. For the user to easily refer to these basic types without having to specify the constructor `BV`, type synonyms are internally defined. Thus, rather than writing `BV String` to refer to the type *String*, the type `TString` can be used. A type naming convention has been adopted, such that, all the types provided by the tool start off with a `T`. Similarly, if the user wants to keep to this convention and abstract away the type constructor, he can specify a type synonym for the new type and use that instead. For instance

```
type TOrder = BI Order
```

These basic types and the parametric types (to represent the different types of data) are defined in a module named `MainTypes.hs` such that an advanced user or a developer can very easily extend the language with new types.

Grouping different types into type classes

While the user-defined types are instance of the internal type class `ComplexType`, the basic built-in types are instances of the class `BasicType`. Another class `DataItemType` inherits the instances of both classes. In this way, to ensure that the data type uses for the modelling elements is valid and previously defined, a compile-time check is carried out to ensure that the type is an instance of this class. Thus, the language types are organised as depicted in Figure 4.30.

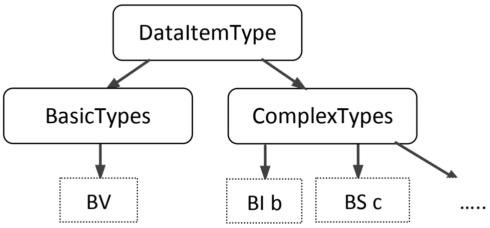


Figure 4. 30: Type classes to handle basic and user-defined complex type

Ideally, we would have preferred not to use parametric types such as `BV a` and `BI a` and instead, create different classes for the different complex types, such as business items, and others which might be added in the future such as business services. Figure 4.31 illustrates this class structures.

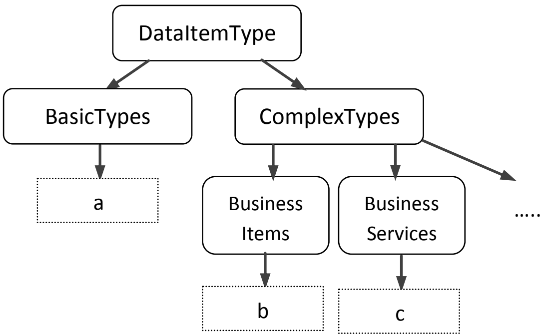


Figure 4. 31: The ideal way how type classes for basic and complex types should have been defined

The issue in this case is that at compile-time, the Haskell compiler would need to specifically match the value to precisely one of the defined instances, before checking whether it is an element of defined classes. However, this is not possible with such a declaration as Listing 4.9.

```
instance (BasicType a)       => DataItemType a
instance (ComplexType a)     => DataItemType a
instance (BusinessItem a)    => ComplexType a
instance (BusinessService a) => ComplexType a
```

Listing 4. 9

For this reason, the solution which we thought is most feasible in our case is to use parametric types such that the different types would be statically distinguishable (Listing 4.10).

```
instance (BasicType  (BV a))  => DataItemType (BV a)
instance (ComplexType(BI a))  => DataItemType (BI a)
```

Listing 4. 10

Specifying the required types as input arguments

As explained earlier, to define the input and output types of a task, first class objects representing types are used. Such objects are already defined for the basic built-in types (in the module `MainTypes.hs`) and are readily available for the user to use. The following naming convention was adopted for these first class objects: the first two characters indicate the data type, that is, whether it is a basic value or a business item, the third character is a '`T`' (to indicate that it is a defined type) and the rest represent the actual name of the type. For instance, the first class object which should be used to indicate a String is `bvTString`. Similarly, when the user defines a new type, he is expected to define the first class object that should be used to represent this type. This is done by the following declarations:

```
      biTOrder = dType :: BI Order
or    biTOrder = dType :: TOrder
if    type TOrder = BI Order          is defined
```

This means that the internal value `dType` (meaning data type) is type casted to the required type. Actually `dType` is the built-in argument in Haskell, `undefined :: a`. Thus, in the above example, by type casting `dType` (that is `undefined`) with the type `TOrder`, the Haskell's compiler would bind the polymorphic type `a` to `TOrder`, such that the type of `biTOrder` is set to `TOrder`.

This type can then be used to define the types of the data that flows through tasks and the data that a repository stores. For instance:

```
t1 = task "task1" (biTOrder :-> biTOrder)
r1 = repository   "rep1"   biTOrder
```

Thus, `task1` takes a business item of type `TOrder` as an input and produces an output of the same type. Repository "`rep1`" stores data of type `TOrder`.

These new types can be defined once and accessed any time. They can easily be added by anyone even if the user is not really an IT technical person or an expert in the domain. Thus, to define a new business item called `Order` then the code in Listing 4.11 is required (note that the second line is optional; however it is usually convenient to define this type synonym to allow the user to refer to the type `TOrder` rather than `BI Order`).

```
newtype Order = Order ()
type TOrder = BI Order
biTOrder = dType :: BI Order        -- or --    biTOrder = dType :: TOrder
instance ComplexType (BI Order)     -- or --    instance ComplexType (TOrder)
```

Listing 4. 11: Defining the user-defined complex type Order

4.4. Embedding the Language Terms and Type System in Haskell

Once the user defines the structure of the model, an abstract representation of the model is internally constructed. With such a representation it is possible to carry out various types of operations and analysis on the model. Thus the model can be interpreted or translate to some other notation, it can be transformed or quality assured. This is only possible through a deeply embedded approach, which was adopted for our language (this approach was introduced in Section 2.4.3.1).

This internal representation of the model is constructed using primitive data constructors defined within an abstract data type, in the host language, such that these constructors act as first class objects in Haskell. For this reason, the data type `PrimPF` (Primitive Process Fragment) was defined (this is a simplified version of the final implementation):

```
data PrimPF = ConstValue Dynamic
            | Task String PrimPF …
            | End PrimPF ……
```

Listing 4. 12: An abstract of the data type `PrimPF`

This is defined as a recursive data type; the end node, for instance, expects as an input another process fragment which is defined in terms of the primitive constructors. Using the above data constructors the process fragment in Figure 4.32

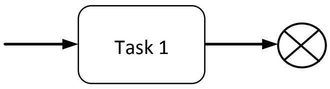


Figure 4. 32

can be defined as

```
pf x = End (Task "task1" x)
```

whereby `x` is essentially another process fragment represented in terms of the data constructors in `PrimPF`. To abstract away these internal constructors and allow the user to define everything in terms of functions, combinators are defined. These functions would then internally invoke the constructors. Listing 4.13 illustrate the combinators that can be defined for the constructors defined in Listing 4.12.

```
constVal :: a -> PrimPF                -- to define a constant
task :: String -> PrimPF -> PrimPF     -- to define a task with a specific
end :: PrimPF -> PrimPF                -- to define an end node
```

Listing 4. 13: The combinators that can be defined for the constructors defined Listing 4.12

The main problem with such combinators is that they do not indicate the data types of the inputs and outputs of the modelling elements and thus, components with incompatible data types can still be combined. For instance, we do not know the type of the input of a task and the data type of its output. In this way, if the user puts a task with an output of type String in sequence with a task with input of type Boolean, the model would still be considered valid. Thus, although these combinators guarantee syntactic correctness and force the user to provide all the required details, the models might still be semantically incorrect, due to the untyped abstract data type `PrimExpr`.

However, the host language Haskell is a strongly statically typed language, and thus, as illustrated in (Leijen & Meijer, 1999) and (Claessen, 2001) and in Section 2.4.3.2 of this document, it is possible to define and embed our own type system, and to use Haskell's compiler to be able to carry out all the required static checks as early as compile-time. These checks would ensure the construction of syntactically correct models, which are also correctly typed. This would act as an additional abstract layer on top of the primitive untyped constructors. This means that after the model is constructed using type-safe combinators, the internal representation would still be the same as that in the first approach. The abstract representation of the model can easily be analysed by recursively pattern matching each of the primitive constructors and by then handling each one accordingly.

This is possible through the definition of a polymorphic type, defined as

```
newtype PF a = PF PrimPF
```

which although has a type variable, it is not used as an argument type of the constructor `PF`. This is in fact a phantom type (phantom types were discussed in Section 2.4.3.2). Thus using this type, the previously defined combinators in Listing 4.13 would be expressed as follows (see Listing 4.14).

```
constVal :: a -> PF a
task :: String -> PF a -> PF b
end  :: PF a -> PF Ctrl
```

Listing 4. 14: A set of safely typed combinators, which should be used instead of those defined in Listing 4.12

Thus assuming that

```
t1 = task "task 1" :: (PF TBoolean -> PF TString)
```

as soon as `t1` is connected to an `end` node (using any of the definitions expressed below),

```
pf x = end (t1 x)      or      pf = end . t1      or    pf = t1 ->- end
```

the compiler would automatically infer the type of the `end` node as `end :: PF TString-> PF Ctrl`. However the internal representation of this model would still remain the same as the one defined before that is

```
pf x = End (Task "task 1" x)
```

Similarly, if `t1` had to be combined in sequence with another task `t2`

```
t2 = task "task 2" :: PF TBoolean -> PF TBoolean
```

that is

```
pf = t2 . t1       or       pf = t1 ->- t2
```

at compile-time, Haskell's type checker would automatically generate an error as the output type of task `t1` is not compatible with the input of task `t2`.

Phantom types are used extensively within our language. For instance, `Ctrl` which is used to define, for instance, the output type of the end node, is a phantom type defined as

```
newtype Ctrl = Ctrl ()
```

This type has one constructor and this constructor has no specific input arguments.

Similarly, if the user wants to define a new type which has no specific attributes, he can do so by defining it as a phantom type that is

```
newtype Order = Order ()
```

However to distinguish a built-in basic type from other user-defined types such as business items, the following types (which are also polymorphic parametric types but without any phantom types) are used

```
newtype BV a = BV a
newtype BI a = BI a
```

such that a user can represent Order as a business item and String as a basic type. Thus, given

```
ord1 :: Order
name :: String
```

a constant value in our language can be defined as follows

```
cOrder = constVal (BI ord1)
cName  = constVal (BV name)
```

These indicate that the first constant is a business item, whereas the second is a basic value. Similarly, since the type synoymns

```
type TString = BV String
type TBoolean = BV Bool
```

are pre-defined, then

```
t1 = task "task 1" :: (PF TBoolean -> PF TString)
```

is equivalent to

```
t1 = task "task 1" :: (PF (BV Boolean) -> PF (BV String))
```

However, within a process, the connection type between elements is also defined to ensure the correct composition of components, according to the required input source and output target types. This was possible through the use of polymorphic types such that rather than a simple data type, as illustrated in the previous examples, the type variable `a` of `PF a`, is one of the polymorphic types, `Flow a`, `ToRep a`, `FromRep a`, `Const a`, where `a` is one of the built-in or user-defined types such as `(BV String)` or `(BI Order)`. Internally, these polymorphic types are defined as follows

```
newtype Flow a    = Flow a
newtype ToRep a   = ToRep a
newtype FromRep a = FromRep a
newtype Const a   = Const a
```

Listing 4. 15: Polymorphic types to represent the connection type of inputs and outputs

In this way, if the input of the previously defined task `t1` is a set to a flow and the produced output is stored in a repository, then the type of the new typed task would be defined as follows

```
tt1 :: PF (Flow TBoolean) -> PF (ToRep TString)
```

Adding an addition level to detect sharing and loops

To detect shared nodes and prevent loops, the observable sharing approach proposed in (Claessen & Sands, 1999) was employed (as discussed also in Section 2.4.3.3). For this reason, every primitive constructor that makes up the model must be referenced, such that it is expressed in terms of the abstract data type `Ref` (the implementation of module `Ref` is available in (Claessen & Sands, 1999)). Thus, the internal constructors must be defined at three different levels.

```
data PrimPF = ConstValue Dynamic
            | Task String RefPrimPF …
            | End RefPrimPF  …

newtype RefPrimPF = RefPrimPF (Ref PrimPF)

newtype PF a = PF RefPrimPF
```

Listing 4. 16: Three abstract data types to represent the models

The first topmost data type captures all the primitive constructors; the second (i.e. `RefPrimPF)` adds a reference for every process fragment; the final type `PF a`, makes use of phantom types to add a layer of type-safety, by specifying the type of the process fragment. These types are later used by the compiler to carry out compile-time checks, during the construction of the model. Note also that as an input, the primitive constructors in the first type are of type `RefPrimPF` and not `PrimPF`. Thus, assuming that the user wants to define a constant, the function `constVal` would take a constant of type `a` (`a` must be an instance of the class `DataItemType` and thus it is either a basic type or a user-defined complex type) and outputs a process fragment of type `PF (Const a),` as illustrated in Listing 4.17.

```
constVal :: DataItemType a => a -> (PF (Const a))
constVal = PF .  RefPrimPF . ref . ConstValue
```

Listing 4. 17: Internal definition of the combinatory, `constVal`

The output in Listing 4.17 is produced, by first passing on the constant input to the primitive constructor `ConstValue` (i.e. `ConstValue c`).) and then evaluate the function `ref` (defined in module `Ref`; see (Claessen & Sands, 1999)) with this input, such that it gets referenced. This is then consumed as input by the constructor `RefPrimPF` (i.e. `RefPrimPF (Ref c)`)). Finally, to ensure that the appropriate type is bound to the structure, the result is passed on to the constructor `PF`. To be able to retrieve the primitive constructor that is `ConstValue c`, then through pattern matching, the constructors `PF` and `RefPrimPF` should be identified so that the referenced structure of type `Ref a` would be retrieved. The function `deref` (also defined in module `Ref`; see (Claessen & Sands, 1999)) would be applied and the constructor `ConstValue c` (of type `PrimPF`) would be returned.

This means that to check whether two structures are shared, then the referenced fragments of type `Ref a` would have to be compared using the operator <=> (also defined in module `Ref`; see (Claessen & Sands, 1999)), such that if they have the same reference, both would be referring to the same structure and thus, the interpreter might not want to re-evaluate it (for instance when a loop is modelled). If on the other hand, the primitive constructor needs to be handled, then the unreferenced structure should be used.

Why these primitives?

For every single modelling element in the language, a corresponding primitive constructor is defined in `PrimPF` data type. Initially it was thought that all the modelling elements can be represented as nodes, such that `PrimPF` would consist of a constructor for constants and another for nodes. However, since various parameters would still have to be checked during the analysis of the model to identify the type of the node, this definition of the abstract data type `PrimPF` is not so feasible. Not to generalize all the modelling elements to one single constructor `Node`, instead of `Node`, other constructors such as `Activity` for tasks and sub-processes and `Action` or `Gateway` for decisions, merges, forks and joins were considered. However, during analysis of the model, rather than simply distinguishing activities from gateways, each modelling element would need to be accurately identified, such that each one would be handled appropriately. For this reason, using these constructors, another input argument would have to be included for each one and checked. In this way, through pattern matching the different modelling elements would be identified and handled accordingly as illustrated in the following function f

```
f (Gateway Decision …) = …
f (Gateway Merge …) = …
```

If the a primitive constructor is defined for every single modelling element, then still the same number of lines of code and checks would have to be carried out (as illustrated in the following example).

```
f (Decision …) = …
f (Merge …) = …
```

This latter approach is more intuitive and more flexible, since every constructor can define its own arguments, irrespective of other constructors. The current definitions can be modified without any restriction and new constructors can easily be added to the data type.

Primitive Constructors with a polymorphic typed argument

In the previous definitions (such as Listing 4.16), the primitive constructor to represent a constant value was defined as follows

```
data PrimPF  = ConstValue Dynamic |…
```

Actually this primitive constructor should be defined as

```
data PrimPF  = ConstValue a | …
```

Since `ConstVal` must represent and keep a constant value of any user desired type, be it a built-in basic type or a complex user-defined type, the actual type of the constant can vary. However, although intuitive, the above definition is not allowed by Haskell's compiler. The main reason is that the compiler cannot infer the type of `a` at compile-time. If `a` was defined as a type variable of `PrimPF`, as in

```
data PrimPF a = ConstValue a | …
```

then the compiler would not generate an error. However, this definition is not a sensible solution for our language; we want constructors of type `PrimPF` and not `PrimPF` a (where `a` is some externally defined type). For this reason, the most feasible solution which we thought would be most appropriate for our language is to convert the input to an item of type `Dynamic` and store the value as an item of type `Dynamic`. The function `toDyn` is used to convert the input and the function `fromDyn` is used reconvert the item back to its original type. These functions and the data type `Dynamic` are provided by the pre-defined module `Data.Dynamic`[1], which is available for Haskell. This approach is analogous to that suggested in (Hinze, 2003). It should be noted that for types to be converted to the structure `Dynamic`, they should also be instance of the class `Typeable`, which is defined in the module `Data.Typeable`[2] (also available for Haskell) . '`deriving Typeable`' should be added to the new type. For instance:

```
newtype Order = Order () deriving Typeable
```

A similar issue was encountered in the implementation of Lava (Claessen, 2001). Due to the problem discussed earlier, types such as `Bool` and `Int` are hard coded and specifically defined in the primitive abstract data type of the language. Thus, if the language had to be extended to support new types, the internal code would have to be modified.

In our case, the users are free to create constants whose types are defined at construction time.

Primitive Constructors having multiple inputs

Some of the modelling elements have more than one input and output and as illustrated earlier these are defined in the form of tuples. Thus, the primitive constructor

```
data PrimPF = ……| Task String RefPrimPF |……
```

Listing 4. 18

is not really complete, since a task can have more than one input. Moreover, the number of inputs that these modelling elements can have is not predefined and the type of a tuple of size 2 is not the same as the type of a tuple of size 3 or any other tuple of any other size. However, within the abstract data type `PrimPF`, the type of arguments of constructors must all be pre-defined. The main reason why inputs are defined in the form of a tuple and not a list, is that the input types might vary. However, one should be aware that once the types of the modelling elements or process fragments are checked, they are individually converted into the type `RefPrimPF`, (also illustrated in Listing 4.18).

For this reason, a recursive data type `GStruct a`, was defined, such that the different input types are converted into one generic structure (see Listing 4.19):

```
data GStruct a =  Single a
                  | Multiple [GStruct a]
```

Listing 4. 19: The abstract data type `GStruct a`

Besides this, a type class named `Generic` was defined as illustrated in Listing 4.20:

```
class Generic a where
      toGStruct   :: a -> GStruct a
      fromGStruct :: GStruct a -> a
```

Listing 4. 20: The type class `Generic`

By creating instances of the class `Generic` and overloading its functions, tuples of different length and different types can be converted to and from a generic structure of type `GStruct RefPrimPF`, and thus, obtain a single type for all the possible inputs. These different typed inputs can then be treated uniformly.

[1] http://haskell.org/ghc/docs/latest/html/libraries/base/Data-Dynamic.html
[2] http://haskell.org/ghc/docs/latest/html/libraries/base/Data-Typeable.html

Listing 4.21 illustrate some of the defined instances of the class `Generic`:

```
instance Generic (PF a) where
         toGStruct   (PF rppf)      =  Single rppf
         fromGStruct (Single rppf) =  PF rppf

instance Generic (a,b) where
         toGStruct (x,y) = Multiple [toGStruct x, toGStruct y]
         fromGStruct (Multiple [x,y]) = (fromGStruct x, fromGStruct y)
```

Listing 4. 21: Some of the instances of the type class Generic

Other sized tuples, which are supported by our language, are defined in a similar manner.

It should be noted, from the definition of the data type `PF a`, that is

```
newtype PF a = PF RefPrimPF
```

that an input of type `PF a` is essentially made up of the data constructor `PF` followed by a process fragment of type `RefPrimPF`. Thus, when such a value is passed on to the data constructor `Single` of the data type `GStruct`, the input would be converted to a structure of type `GStruct RefPrimPF`.

For this reason, primitive constructors with multiple inputs, such as a task, can be defined as follows

```
data PrimPF = … | Task String (GStruct RefPrimPF) |…
```

Similarly, using this technique, generic operations can be defined such that they can operate on any value whose type is an instance of this class.

4.5. Type Classes in our Language

The main objective of our language is to assist users in the construction of high quality models, where errors are trapped as early as construction time. Embedding our language in a strongly statically typed language such as Haskell, we are able to use the type checker of the host language and carry out compile-time checks to ensure the construction of type safe models. For this reason, we embedded our own type system as a layer over the untyped primitive constructors of the model, by using phantom types.

Haskell also provides type classes. These were used extensively in our language to enforce certain type constraints and to ensure the appropriate application and use of the provided modelling elements. The `Generic` type class (defined in the previous section), was one of the most important classes used to represent and handle different types of inputs and outputs in a uniform generic structure. In this case, the class was used to overload specific functions and thus provide different implementations for the same functions depending upon the type being handled.

Type classes to constraint the polymorphic types

Type classes are usually convenient when polymorphic and parametric types are defined. In certain cases, it would be essential to specify that any type that belongs to a particular class is allowed. For instance,

```
constVal :: (DataItemType a) => a  ->  PF (Const a)
```

where the class `DataItemType` is defined as illustrated in Listing 4.22.

```
class DataItemType
instance (BasicType (BV a))   => DataItemType (BV a)
instance (ComplexType (BI a)) => DataItemType (BI a)
......
```

Listing 4. 22: The abstract data type `DataItemType`

This means that, the value that can be specified as a constant, must either be a basic value (whose type is a built-in basic type) or any item with a user-defined complex type, such as business items. Thus, in this case, the class `DataItemType` was simply used to constraint the type that can be bound to `a`. If the type of the value is not an element of the class `DataItemType`, then the compiler would generate an error as early as compile time.

Type Classes with Functional Dependency

In other cases, classes with functional dependency were defined, such that, besides ensuring that the type is appropriate and that it is an instance of the class, given a specific type as input, another type would be inferred. For instance, let us consider the class `ActIODtSet` (*Activity Input and Output Data Set*) (Listing 4.23).

```
class ActIODtSet a b | a->b
instance (DataItemType (BV b)) => ActIODtSet (BV b) (PF (BV b))
instance (DataItemType (BI b)) => ActIODtSet (BI b) (PF (BI b))
instance (ActIODtSet a1 a2, ActIODtSet b1 b2) => ActIODtSet (a1,b1) (a2,b2)
…
```

Listing 4. 23: The type class `ActIODtSet` (contains functional dependency a->b)

A particular instance of this class is defined by two types, where the second is inferred from the first. This class is actually used to check the input and output data types defined for a particular task. Thus, when, for instance task `t1` is defined in the following manner

```
t1 = task "task 1" ((bvTString, biTOrder) :-> biTOrder)
```

the actual data type specified as input to the task is `(TString, TOrder)` which is equivalent to `(BV String, BI Order)`. The data type of the required output is `TOrder` that is `(BI Order)`. Assuming that `TOrder` was defined as a complex type by the modeller before defining this task, then both `TOrder` and `TString` are instances of the class `DataItemType` and thus both are instances of the class `ActIODtSet`. However, after task `t1` is defined using the previous definition, its inferred type would be

```
t1 :: (PF TString, PF TOrder) -> PF TOrder
```

How did the system infer this type? Looking back at the definition of class `ActIODtSet`, the functional dependency `a->b` can be noted at the heading of the class. This indicates that type `b` must be inferred from `a`. In fact, to obtain `PF TOrder` from `TOrder` and `(PF TString, PF TOrder)` from `(TString, TOrder)` then it is enough to define the type signature of the combinator task in the following manner:

```
task :: (ActIOSet a c, ActIOSet b d, …..) => String -> (a :-> b) -> c -> d
```

where `String` is the type of the first argument which represents the name of the task; `a` and `b` represent the input and output types of the task defined using first class objects (in the case of task `t1`, `a` is bound to type `(TString, TOrder)` and `b` to type `TOrder`); `c` and `d` are the input and output types of the actual task (in the case of task `t1`, `(PF TString, PF TOrder) -> PF TOrder`). Thus for `c` and `d` to be inferred from `a` and `b` respectively, a type constraint is set such that `a c` and `b d` must be elements of the class `ActIOSet`. If not, or if the compiler cannot decide how to infer the second type, then an error would be generate at compile-time.

Why Type Classes with Functional Dependency were so Important

Another interesting case where classes with functional dependency were essential to infer the types, is in the definition of the function `packageSubProcess`, which is used to package a number of process fragments into one sub-process. As input, besides the name of the sub-process, the function expects a process fragment (as a function) or a tuple of process fragments. These fragments have their own input and output types. If for instance,

```
sp1 = packageSubProcess "subProcess1" (pf1, pf2, pf3)
```

where

```
pf1 :: (PF TOrder, PF TString) -> ……
pf2 :: (PF (Flow Ctrl)) -> …
pf3 ::  PF (Flow TProduct) -> …
```

then the input type of the sub-process must be `(PF TOrder, PF TString, PF (Flow TProduct))`. In this way, the modeller who later on makes use of the sub-process, would not have to know that internally there is a process fragment that starts requires a control flow (rather than a data-flow) as input. This control flow would easily be obtained from the input data flow. In fact, to split the input data flow into the required flows for the internal process fragments, a splitter component is used (as illustrated in Section 4.2.8). However, the issue in this case is how to infer the type `(PF TOrder, PF TString, PF (Flow TProduct))` from the input types of the process fragments such that the type of `sp1` would be

```
sp1 :: (PF TOrder, PF TString, PF (Flow TProduct)) -> …
```

Considering that the type of `packageSubProcess` is

```
packageSubProcess :: (…) => String -> a -> b -> c
```

where `a` represents the type of the process fragments passed as input and `b` and `c` are respectively the input and output types of the final sub-process. `(…)` indicates the location where the type classes that the input and output types should be instances of are defined.

A class named `SPIOType` is defined such that the input type of every element of every process fragment is recursively analysed. If it is a control flow, it is ignored. Else the type is kept and included in the final tuple which is automatically inferred as the input type of the new sub-process. The output of the sub-process is inferred in a similar manner. All these checks and type inferencing is carried out at compile-time by the Haskell type system such that the type of the sub-process would be automatically inferred before the actual sub-process is returned. Besides being able to carry out the checks statically at compile-time, the compiled-executable code would also result to be faster than that produced by dynamically typed systems.

Type Classes to carry out checks on Tuples

They were particularly useful to handle tuples. For instance, to ensure that all the elements of the tuple are of a particular type or to infer the type of elements within such tuples, to compare the size of tuples, to convert a tuple into a list (when the elements of the tuple are of the same type). In this way, it was possible to use tuples in a controlled manner for all of our modelling elements irrespective whether the types within the tuples are different or equivalent. Our language currently supports tuples of up to the size of 10.

Type Classes to Type Constraint the Inputs and Outputs of Modelling Elements

They were also useful to guarantee the correctness of models during construction, by carrying out compile-time checks on the types of the inputs and outputs of every modelling element. For instance, let us consider the fork gateway. The type of the fork is defined as follows:

```
fork :: () => a -> b
```

where `a` and `b` represent the input and output types of the decision gateway

The following input and output types are all considered valid (Listing 4.24):

```
fork :: PF TString -> (PF TString, PF TString, PF TString)
fork :: (PF TOrder, PF TString) -> ((PF TOrder, PF TString), (PF TOrder, PF TString))
fork :: (PF (Flow TOrder)) -> (PF TOrder, PF (ToRep TOrder), PF (Flow TOrder))
```

Listing 4. 24: Valid ways how a fork can be used

Thus type classes are required to carry out the following checks: 1) ensure that the data type (not necessarily the connection type) of the input is equivalent to that of all the outputs, 2) the input source and output target types (i.e. the connection type) can be different but must be valid, 3) the fork can have multiple input flow which are then passed on to all the outgoing branches, 4) the output is always a tuple of minimum size two. For this reason:

1) A class `PFTupleSpecDt` (*Process Fragment Tuple has elements with Specific Data type*) is required such that `PFTupleSpecDt` b a would ensure that the data type of every element in the tuple `b` is equivalent to the data type (not the connection type) to that of `a`
2) A class `PFTupleForIO` (*Process Fragment Tuple For Input and Output)* can be defined to ensure that the input source and output target types are valid. For instance, none of the elements in the output tuple must be of the form `PF (Const a)` since constant is not an output target
3) A class `PFSetForGt` (*Process Fragment Set For Gateway*) would ensure that the input is either a single control flow or a tuple of multiple data flows (but not multiple control flows). Thus, `PFSetForGt a`
4) A class `Tuple b` would guarantee that the output is always a tuple of minimum size 2.

Considering that these are some of the most important checks that are carried out at compile-time to ensure the appropriate use of the gateway `fork`, the type signature of this combinator would be defined as:

```
fork :: (Tuple b,PFSetForGt a,PFTupleForIO a In,PFTupleForIO b Out, PFTupleSpecDt b a) => a -> b
```

Thus, it should be noted that type classes were used for three particular purposes within our language: 1) to overload functions, 2) to check the validity of parametric polymorphic types, and 3) to infer new types.

4.6. Tagging Process Fragments into Sub-Processes

To allow users to define sub-processes in a uniform manner irrespective of the number of process fragments that need to be packaged, the function `packageSubProcess` is provided. The role of this function is to: 1) infer the new type of the sub-process (as discussed in the previous section), 2) add any start nodes to fragments that have a control flow as input, 3) add a `ctrl_dataFlowCombiner` and a `ctrl_dataFlowSplitter` to split the input data flow appropriately according to the inputs of the fragment and to combine the outputs of the fragments into the appropriate output of the sub-process. This is illustrated in the Figure 4.33.

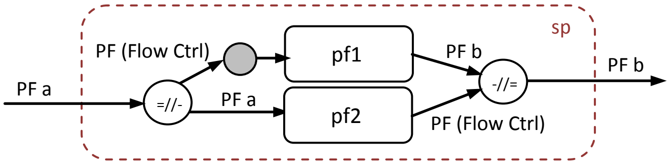


Figure 4. 33: A sub-process packaged by the function `packageSubProcess`. It contains two process fragments, *pf1* and *pf2*. A `ctrl_dataFlowCombiner` and a `ctrl_dataFlowSplitter` are attached to the inputs and outputs of the sub-process to group the inputs and outputs of all the process fragments

This function would then invoke the combinator `subProcess`, such that a primitive constructor `SubProcess` is added to the internal abstract representation of the model. However, different from other primitive constructors, as input, `SubProcess` expects

```
data PrimPF = …|  SubProcess  String  (GStruct RefPrimPF)  (GStruct RefPrimPF)  | …
```

Thus, the combinator `subProcess` is defined as

```
subProcess  nm  pf  x  =  liftStruct  (SubProcess nm)  (pf x)  x
```

whereby `nm` represents the name of the sub-process, `pf` represents the process fragment that should be tagged and `x` represents the input of the sub-process. `liftStruct`, on the other hand, transforms `(pf x)` and `x` into structures of the type `GStruct RefPrimPF` (discussed in Section 4.4), passes these as input to `SubProcess` and later on, lifts primitive process fragment from type `PrimPF` to the required `PF a` type.

This means that if the interpreter during the analysis of the model wants to analyse the internal structure of the sub-process, then the second argument would be analysed as any other process fragment. If the interpreter wants to consider the sub-process as a single modelling element then the second argument would be ignored.

The problem with this approach is that one sub-process is distinguished from another from its name. This means that if the user accidentally assigns the same name to different sub-processes, then issues would be encountered during analysis of the model. To try to elevate this problem a function `isNameUsed` is provided to allow users to check whether the name is already used within that process. To further prevent such naming issues brought about by explicitly tagging the sub-processes, the input and output types of the subprocess are identified by the combinator `subProcess` and added as another input argument to the primitive constructor `SubProcess`. In this way, during analysis, both the name and the type can be checked to ensure that the required sub-process is handled.

Thus, although explicit tagging is not the ideal solution to tag and uniquely identify blocks, even though other solutions were proposed in literature (see Section 2.4.3.4), we still adopted this solution since it is rather effective for our purpose and for our functional approach. We also provide functions to try to elevate issues brought about by explicit tagging.

4.7. Connection patterns

Since in our language models and process fragments are essentially functions which given an input, a particular output is produced, then fragments can easily be connected together by using the output of one fragment as the input of another. Although in certain cases this is the only way how fragments can be combined together since the output can easily be passed on to another other fragment, models constructed in this way are not always that readable and take up more lines of code. The reader would have to check the entire definition to identify from where a particular value is obtained.

Noting that in our host language, Haskell, functions are considered first class objects, it is possible to use higher order functions and thus define connections patterns. The connections patterns are simple functions which take other functions (rather than values) as input, combine then in a certain manner and return the new composed function. This approach was also used in the hardware description language, Lava (Claessen, 2001). Circuits produced using these connection patterns were much more readable, easier to understand and reason about and more concise in terms of definition.

Similarly, a number of connection patterns are defined in our language. Besides the advantages mentioned earlier, with such patterns our main aim was to help users to produce models of a high quality, quicker and with the least amount of effort. The more code required to define a model, the greater the chance of introducing new errors. The symbols used for the defined operators were chosen carefully such that models would be more readable and even though text is used to define the models, with these operators it would still be easy for the modeller to visualize the models. The connection patterns also help to abstract away certain basic modelling elements which can be inferred automatically.

Serial and Parallel Composition

Two of the most basic and important connection patterns are *serial* composition (`->-` or `serial`) and *parallel* composition (`-|-` or `par`), both of which take two process fragments as input. These are defined in Listing 4.25.

```
serial pf1 pf2          = pf2 . pf1
par pf1 pf2 (in1, in2) = (pf1 in1, pf2 in2)
```

Listing 4. 25: Defining connection patterns `serial` and `par`

Although `serial` for instance is simply another way how to express the same behaviour of the built-in function composition `(.)`, it is still important as it makes models more readable. For example it is easier to reason about `pf1 ->- pf2` rather than `pf2 . pf1` as it is more intuitive and it depicts the flow and precise order in which process fragments are executed. Parallel composition is important to group fragments, such that if `pf1` has type `(PF a, PFb) -> PF d` and `pf2` has type `PF c -> PF e` then the new fragment would have type `((PF a, PF b), PF c) -> (PF d, PF e)`. To handle more process fragments, then `serialC` and `parC` (for serial and parallel *Compound* composition) are defined.

Other connections patterns are defined for *gateways* and control and data flows combinators and splitters. For instance:

`-<| … |>=` *for exclusive decision*

E.g.:
```
pf1 -<| "Order Complete?" (branchProp eYes 0.4, branchProp eNo 0.6)|>= (pf2, pf3)
        =
pf1 ->- (exclDecision "Order Complete?" (branchProp eYes 0.4, branchProp eNo 0.6))
    ->- (pf2 -|- pf3)
```

Listing 4. 26: Connection pattern for *exclusive decision*

In Listing 4.26, `"Order Complete?"` is the name of the exclusive decision; `(branchProp eYes 0.4, branchProp eNo 0.6)` represents the expressions `(eYes, eNo)` and the probabilities `(0.4, 0.6)` which are assigned to each of the decision branches; `pf1, pf2, pf3` are process fragments which are attached to the input and output of the decision.

More details about such connection patterns are available in the tutorial provided as Appendix A.

Lazy Serial Composition

Another interesting connection pattern is *lazy serial composition* (->>– or `serialL`). Similar to serial composition, its main purpose is to combine two process fragments in the sequence. However, assuming that

```
pf1 :: PF (Flow TString) -> PF (Flow TString)
pf2 :: PF (TString) -> PF (TOrder)
```

it is not possible to put `pf1` in sequence to `pf2` in the following manner

```
pf1 ->- pf2
```

but it is possible to define

```
pf1 ->>- pf2
```

Thus whereas in `->-` the output type of the first fragment must precisely match the input type of the second, the only restriction that `->>-` imposes is that the data type flowing out of the first fragment must be equivalent to the data type expected as input by the second fragment. Thus, since the output data type of `pf1` and input data type of `pf2` is `TString`, then fragments can safely be composed in sequence using this operator. Similarly, it would be possible to safely compose any of the fragment pairs in Listing 4.27 by using this definition `pf1 ->>- pf2`

```
pf1 :: PF a -> PF b                pf2 :: PF b -> PF c
pf1 :: PF a -> PF b                pf2 :: PF (Flow b) -> PF c
pf1 :: PF a -> PF (FromRep b)      pf2 :: PF b -> PF c
pf1 :: PF a -> PF (Flow b)         pf2 :: PF (Flow b) -> PF c
```

Listing 4. 27: Fragment pairs that can safely be composed in sequence by using the lazy serial composition connection pattern

If the output type of the first and that of the second match perfectly then `->>-` would operate in the same way as `->-` . If, on the other hand the types of `pf1` and `pf2` have a different connection type (such as the process fragments in Listing 4.28), then an error would be generated:

```
pf1 :: PF a -> PF (Flow b)          pf2 :: PF (ToRep b) -> PF c
```

Listing 4. 28: These process fragments cannot be composed in sequence using `->>-`

To connect more than two process fragments in sequence, then `serialLC` can be used. For example:

```
serialLC (pf1, pf2, pf3, pf4)
```

What this operator does, is that, it adds the appropriate component to specify the connection type. For example, if

```
pf1 :: PF a -> PF b          pf2 :: PF (Flow b) -> PF c
```

then

```
pf1 ->>- pf2        =        pf1 ->- flowCO ->- pf2
```

(`flowCO` *since it must specify the output target type of* `pf1`)

If, on the other hand,

```
pf1 :: PF a -> PF (FromRep b)     pf2 :: PF b -> PF c
```

then

```
pf1 ->>- pf2        =        pf1 ->- repCI ->- pf2
```

(`repCI` *since it must specify the input target type of* `pf2`)

From the few examples which have been discussed, it should be obvious that this operator is actually an overloaded function. Although the examples above, illustrate fragments with just one input and one output, fragments with any number of inputs and outputs can be handled in this same manner.

The main advantage of this operator is that, if the modeller does not really need to specify the input source or output target type of an activity, then he could use this connection pattern (rather than `->-`) to compose elements in the process such that if there is an activity or a fragment whose connection type is not defined, then it would be inferred automatically (without the modeller knowing) as soon as it is connected to other fragments. If on the other hand, the user wants, for instance, a task to obtain a particular input from a repository then he should specify the input source by using the operator `<|` (as illustrated in Section 4.2.9). However, he could still use `->>-` to connect such a task to the required repository. Thus with just one operator, the modeller can handle different types of fragments, with the less amount of effort and still produce a safe model.

Issues encountered when defining Lazy Serial Composition

While defining this operator for certain types of the models, some issues were encountered. For instance,

```
pf1 :: PF TString -> PF TString

pf1 ->>- end        or        pf1 ->>- stop
```

Note that the output target type of `pf1` is not specified. Moreover, the type signature of `end` and `stop` node is defined as

```
end  :: PF (Flow a) -> PF (FromEnd a)
stop :: PF (Flow a) -> PF (FromStop a)
```

Since the data type flowing through the end and the stop node is defined by type variable `a`, the actual type is only inferred when this node is composed to another fragment. Let us consider just the end node

```
pf1 ->>- end     that is     (PF TString -> PF TString)  (PF (Flow a) -> PF (FromEnd a))
```

In this case, since `->>-` is an overloaded operator and since `a` cannot be inferred automatically, then it is not possible for the compiler to choose the appropriate implementation of `->>-`, that is

```
pf1 ->- flowCI ->- end
```

To solve this problem then the type of `a` must be inferred before `->>-` is invoked. For this reason, we provide the user with another connection patterns `|><|` such that

```
pf1 |><| end     =    pf1 ->>- end
pf1 |><| stop    =    pf1 ->>- stop
```

with the only difference that in the type signature of `|><|`, a type constraint is set such that the output type of the first process fragment must be an instance of the defined class, which through functional dependency retrieves the data type and passes it on to the class containing the overloaded operator `->>-`. In this way, the appropriate implementation of the function is chosen.

Two other connection pattern which are related to operator `|><|` are

```
endBranch, stopBranch
                endBranch    =   (noActivity  type) |><| end
                stopBranch   =   (noActivity  type) |><| stop
```

`noActivity` is a function which is used to create an invisible component which would allow flows of a specific type to pass through without being modified. This is usually essential to define a decision which has a specific process fragment attached to one of the branches but none to the other. Using this function and the connection pattern `-<|..|>=`, such a decision can be defined as follows:

```
-<| "Order Complete?" (branchProp eYes 0.4, branchProp eNo 0.6)  |>= (noActivity biTOrder, pf)
```

If on the other hand an end or a stop node needs to be attached directly to one of the decision branches, rather than using `((noActivity biTOrder) |><| end)`, `endBranch` can be used:

```
-<| "Order Complete?" (branchProp eYes 0.4, branchProp eNo 0.6)  |>= (endBranch biTOrder, pf)
```

If rather than combining two process fragments in sequence, a value needs to be passed on to a function and this same automatic inferencing behaviours is required, then the connection pattern `*>>-` can be used. For instance, if

```
x :: PF TString
pf :: PF (Flow TString) -> PF b
```

then

```
x *>>- pf   =   x ->- flowCO ->- pf
```

such that, with function `flowCO`, the type of `x` (`PF TString`) is converted to (`PF (Flow TString)`). In this way, it is safely passed on as input to `pf`.

Other Connection Patterns

Such connection patterns are important to allow users to combine one function or process fragment with another in a similar way as graphically composing a model made up of different components. After all, our aim was to create a number of combinators which can be used to combine different fragments. Adopting such an approach the modeller is able to focus on the behaviour of each fragment and abstract away from certain details which are not that relevant to the modeller, while still ensuring the correctness of the models. These patterns were careful selected after various models were analysed and defined in our language. We also scrutinized the various anti-patterns and patterns identified by Koehler and Vanhatalo in (Koehler & Vanhatalo, 2007). This article captures some of the best modelling practises and some of the most common modelling mistakes which were noted after hundreds of models created by different users using different modelling tools were reviewed. For this reason, we try to provide more abstract connection patterns (or rather built-in models) which would help modellers adopt good modelling practise with the least amount of effort and modelling expertise.

For instance, if an inclusive decision had to be defined in the following manner (Figure 4.34),

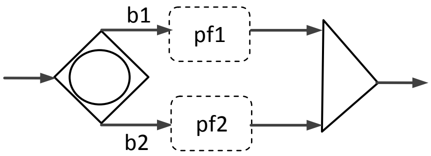


Figure 4. 34: An inclusive decision – merge fragment

then the process might lack synchronisation. If more than one of the branches of the inclusive decision is true then the merge would allow both flows to pass through and thus the process fragment following the merge would be executed twice. To prevent such behaviour, the authors in (Koehler & Vanhatalo, 2007) recommend modellers to explicitly define all the possible combinations of the branches using exclusive decisions and forks. However, considered an inclusive decision with 2-branches then an exclusive decision with 3-branches would be required. Actually, the number of branches of the exclusive decision, grows exponentially in terms of the number of branches of the inclusive decisions. Considering a 3-branch inclusive decision, then a 7-branch exclusive decision would have to be models. Thus an n-branch inclusive decision should be modelled using a (2^n -1) branch exclusive decision, as illustrated in Figure 4.35.

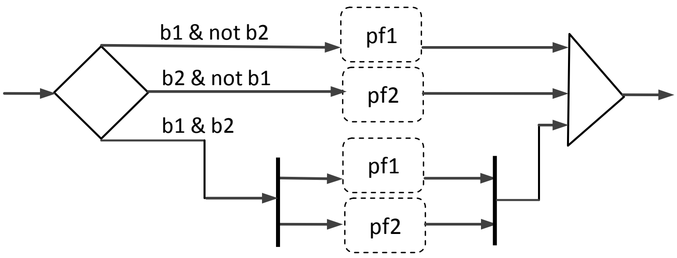


Figure 4. 35: This fragment models the 2-branch inclusive decision – merge in Figure 4.34 using only exclusive decisions and forks

It is not really sensible for a modeller to define the behaviour of an inclusive decision in this manner and if done, there is a great probability that errors are introduced in the definition. For this reason, we provide the function `sound_inclDecision_Merge` (*sound inclusive decision – merge fragment*) to produce a sound inclusive decision – merge fragment which does not lack synchronisation. A 2-branch inclusive decision which is named `"decision1"` and which has branches `(branchProp e1 0.8, branchProp e2 0.2)`, can be defined as

```
sound_inclDecision_Merge "decision1" (branchProp e1 0.8, branchProp e2 0.2) (pf1, pf2)
```

Another important function which ensures the production of a sound structured cycle is `soundCycle` (*merge – exclusive decision structured sound Cycle*), as illustrated in Figure 4.36:

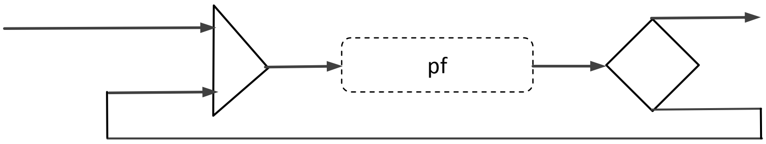


Figure 4. 36. A structured sound cycle

This would be defined as `soundCycle pf (nm, brs)` where `nm` is the name of the decision and `brs` is a pair containing the details of the decision branches. In fact, as defined in (Koehler & Vanhatalo, 2007), this is the only way how a sound cycle can be produced. Other cycles made up of a fork and a join or fork and a decision would lead to a deadlock, whereas a cycle made up of a merge and a fork or an inclusive decision would lead to lack of synchronisation, possibly leading to an explosion of uncontrolled iterations of the process fragment. Thus, by using this connection pattern, the modeller can abstract away from the details of how a sound cycle should be constructed and focus solely on the behaviour of the process fragment that should be repeated until a condition (i.e. the expression assigned to the first decision branch) would be satisfied. This would be similar to the *while..do* structure provided in IBM's tool.

Other connection patterns to produce triangular models such as Figure 4.37

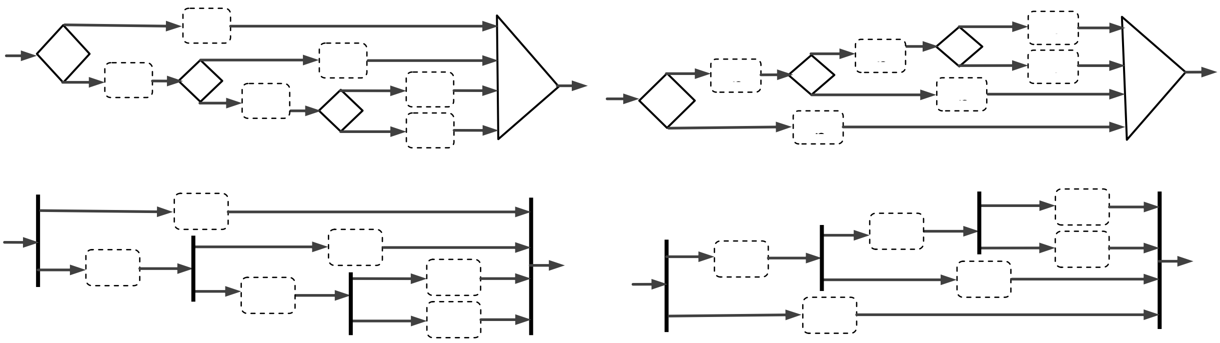


Figure 4. 37: Triangular models

and another to terminate unnecessary data flows on each of the outgoing decision branches such as Figure 4.38

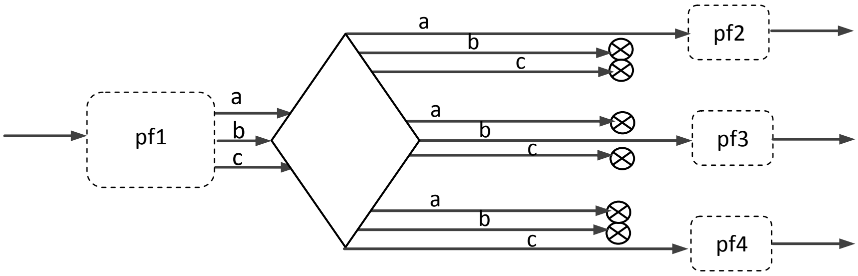


Figure 4. 38: An 3-branch exclusive decision with a flow of type (a,b,c)
Only one of the data flows is maintained as soon as the branch is chosen

are also provided. Their application and functionality is explained in more detail in Section "*Built-In Models*" of the tutorial provided as Appendix A.

In this section, just a few of the connection patterns were discussed. For more details about all the connection patterns, refer to Section A.5 of the tutorial (Appendix A). To note the difference between models defined using the conventional method and using connection patterns, refer to Chapter 6 (‘*Evaluation and Case Studies’*) and Appendix B (for other samples of models defined using our language).

4.8. Parameterized Models

Since definitions of models in our language are essentially functions, it is possible for the user to define a definition with parameters, such that, by providing the value of these input arguments, the required model is constructed. These are known as *parameterized models* or *blocks* and are used in most functional embedded languages, such as hardware description languages, for instance Lava (Claessen, 2001), to provide the right abstraction for the user to focus on the required behaviour rather than the implementation. In this way, by defining the required structures as *parameterized models*, the modeller is able to reuse the same definition with different parameters, to rapidly construct models in a concise and easy to comprehend manner. By ensuring that all the possible fragments generated by these parameterized models are sound, the user would be able to safely use this model within various processes, without having to re-check its soundness. This concept has been introduced in Section 2.4.3.6, where a number of examples of parameterized blocks, defined within a language that handles expressions, were discussed.

In contrast to connection patterns, these models can easily be defined by the modellers themselves, as soon as they realize that similar structures are often used to model different processes within the organization. Even though some to the connection patterns, defined in the previous section such as the pattern that produces sound inclusive decision merge fragments, can be considered as parameterized models, such patterns are not always so straightforward to implement and thus, cannot be defined by the modellers themselves.

Consider the process fragments in Figure 4.39 and 4.40.

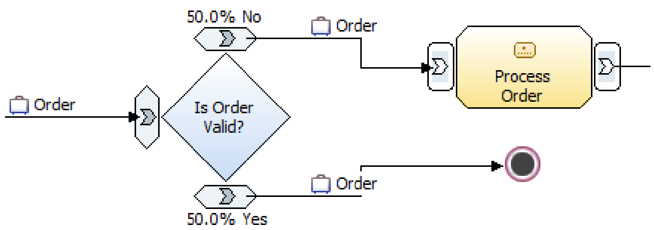


Figure 4. 39: A process fragment handling an order

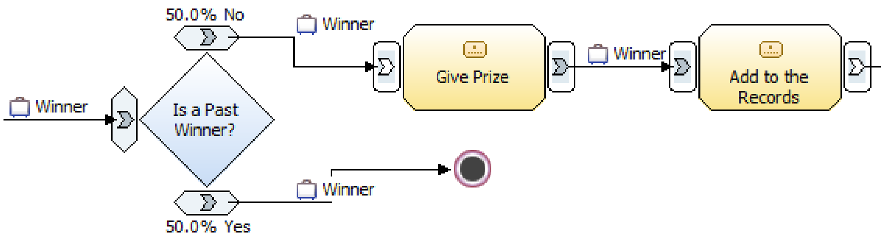


Figure 4. 40: A process fragment to decide whether a prize should be given to a winner

Assuming that the required tasks, types and decision branch expressions, `eYes` and `eNo`, are already defined, these fragments can easily be constructed by the definitions in Listing 4.29 and 4.30.

```
(exclDecision "Is Order Valid? "(branchProp eYes 0.5, branchProp eNo 0.5))
->- (tProcessOrder -|- stopBranch biTOrder)
```

Listing 4. 29: Defining in our language the process fragment in Figure 4.39

```
(exclDecision  "Is a Past Winner? "(branchProp eNo 0.5, branchProp eYes 0.5))
->- ((tGivePrize ->- tAddToRecords) -|- stopBranch biTWinner)
```

Listing 4. 30: Defining in our language the process fragment in Figure 4.40

Since both fragments are made up of an exclusive decision, a stop node and some other process fragment and since such a fragment might be used within other different processes, then it would be feasible for the modeller to define a parameterized model to handle such fragments, as illustrated in Listing 4.31.

```
exclDec_withLowerStop nm brs typ pf  = (exclDecision nm brs)->-(pf-|-stopBranch typ)
```

Listing 4. 31: A parametrized model to define models as in Figure 4.39 and 4.40

`nm` is the name of the decision; `brs` is the tuple defining the properties of the decision branches; `typ` is the type of the data flow; `pf` is the process fragment attached to the upper branch

Using this, then the fragments in Figure 4.39 and 4.40 can be defined as in listings 4.32 and 4.32

```
exclDec_withStopBranch  "Is Order Valid?"
                        (branchProp eYes 0.5, branchProp eNo 0.5)
                         biTOrder
                         tProcessOrder
```

Listing 4. 32: Defining the process fragment in Figure 4.39 using the parameterized model in Listing 4.31

```
exclDec_withStopBranch  "Is a Past Winner? "
                        (branchProp eNo 0.5, branchProp eYes 0.5)
                         biTWinner
                        (tGivePrize ->- tAddToRecords)
```

Listing 4. 33: Defining the process fragment in Figure 4.40 using the parameterized model in Listing 4.31

Comparing the definitions in listings 4.29 and 4.30 with those in listings 4.32 and 4.33 (defined using the parameterized model `exclDec_withLowerStop`), it should be noted that in the latter definitions, the fragments in figures 4.39 and 4.40 are both easily defined in terms of one function. Moreover, as illustrated in Listing 4.31, it is rather straightforward for such parameterized models to be defined.

In a similar way, parameterized models that generate more complex structures can be constructed. Thus, considering the fragment in Figure 4.41, it should be noted that, before defining the exclusive decision merge fragment, the internal fork join models that are connected to the outgoing branches of the decision and incoming branches of the join, should be defined. These should then be composed in parallel and later on put in sequence with the other gateways. For this reason, the definition of a parameterized model that is capable to generate such structures, in Listing 4.34, is essentially made up of two functions.

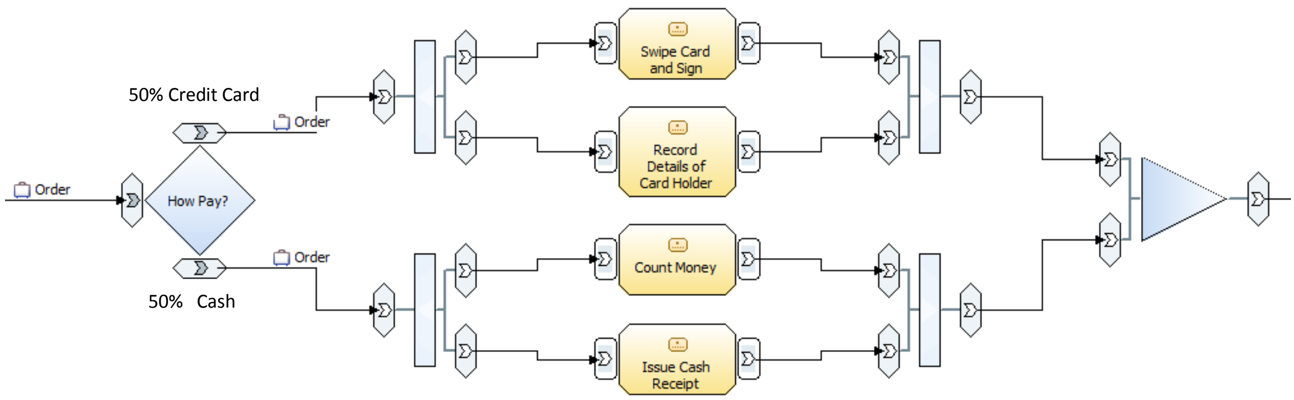


Figure 4. 41: A decision - merge fragment with internal fork -join fragments

```
fork_joins [pfsFJ]           =   fork_join pfsFJ
fork_joins (pfsFJ : pfsFJs)  =  (fork_join pfsFJ)  -|-  (fork_joins  pfsFJs)

exclDecisionMerge_forkJoins nm brs pfsL =
                                         exclDecision_merge nm brs (fork_joins pfsL)
```

Listing 4. 34: Defining the parameterized model `exclDecisionMerge_forkJoins` to define fragments as in Figure 4.41

Since ideally the user of such a parameterized model should be able to pass on the fragments of the internal fork-joins as a list of tuples (each tuple contains the internal fragments for a particular fork join), then a function which is essentially another parameterized model, `fork_joins`, is defined (Listing 4.34). The role of this function is to construct the fork join fragments and compose them in parallel. To do this, the connection pattern `fork_join`, defined in our language, is used. Once constructed, this is passed on to the built-in connection pattern `exclDecision_merge`, which generates the exclusive decision merge fragment. With such a parameterized model, the fragment in Figure 4.41 can be defined as illustrated in Listing 4.35.

```
pf = exclDecisionMerge_forkJoins "How Pay?"
                                 (branchProp eCreditCard 0.5, branchProp eCash 0.5)
                                  [(tSwipeCardSign,tRecordDetailsCardHolder),
                                  (tCountMoney, tIssueCardReceipt)]
```

Listing 4. 35: Defining the model in Figure 4.41 using the parameterized model `exclDecisionMerge_forkJoins`

From the above examples it should be noted that any parameterized model that generates models having a particular structure, can easily be defined by the modellers themselves. Different from IBM's WebSphere Business Modeler, the users are not limited to a simple recording functionality. Instead, any parameterized model can be constructed. These models are important since they help the modellers to abstract away from the implementation details and rapidly constructed good quality models in a concise manner, as illustrated in listings 4.32, 4.33 and 4.35.

4.9. Related Work

Various languages, notations and modelling tools having been developed over the years to assist business analysts to create business process models quicker and with the least amount of effort. The most recent notation is Business Process Modelling Notation (BPMN) (OMG, 2008) whose main objective was to unify the features of all the other languages (such as UML2-Activity Diagram and Event-Driven Process Chain) and to have just one standard notation. IBM has developed its own modelling language and tool (WebSphere Business Modeler[1]) to assist companies in the adoption of business-driven development (Mitra, 2005). However, none of the languages adopted a functional approach, based on higher-order logic rather than first-order logic.

As argued in (Koehler, et al., 2007), a *declarative approach* would be appropriate to define pre and post conditions to assure the quality of the models when in-place transformations are carried out. It would also be possible to allow users to define their own composite transformations. In (Koehler, Hauser, Sendall, & Wahler, 2005), pre and post conditions of out-place transformations were represented in the Object Constraint Language and used successfully to refine the graphical models into the executable BPEL code. As defined by Backus's Turing Award paper (Backus, 1978), the main reason why a declarative and functional approach results to be more effective than other imperative approaches, is that users are able to abstract away from the implementation details and focus on what operations are required rather than how such operations should be implemented. This means that this approach brings about other advantages than simply those identified in (Koehler, et al., 2007) and (Koehler, Hauser, Sendall, & Wahler, 2005). In fact, Sheeran in (Sheeran, 2005) defines how powerful functional languages can be for hardware design. She states that the two are "a *perfect match*". She created the first functional hardware description language known as µFP and following that, various others were developed, some of which, such as Lava (Claessen, 2001), were embedded in functional programming languages, such as Haskell (Jones S. P., 2003). Noting how effectively certain features in Haskell were used to define circuits and identifying the similarities between circuits and business process models, we were inspired to use Haskell as our host language. By analysing various other successfully domain specific languages embedded in Haskell such as Fran's reactive animation language (Elliott & Hudak, 1997) and the results of an experiment conducted for geometric region analysis (Hudak & Jones, 1994), we continued to confirm the appropriateness of such a functional language to embed our own domain-specific language.

Thus, different from the current modelling notations, in our language, models are presented as *functions* such that given an input, the required behaviour is carried out and an output is produced. By passing this value as an input to another process modelled as a function, then the two would be inherently connected. Moreover, since Haskell is a pure functional language, these functions are also pure and thus given a specific input, the same output is always returned, irrespective when and where it is executed.

We have adopted a *combinatorial approach*, as in (Jones, Eber, & Seward, 2000) whereby a combinator library in Haskell was produced to compose various financial contracts. The different contracts produced using these combinators are evaluated uniformly using the same functions. In this way, using these combinators, sub-contracts are combined together to form more complex contracts. An internal abstract representation of these contracts is constructed and later on used to carry out the required evaluation. In our language, by employing such a deeply embedded approach, the basic modelling elements act as combinators. With them any model can be defined. The internal abstract representation of the model is abstract enough such that, even though our language aims to capture the semantics and features of IBM's tool, any interpreter can easily be developed to define the model (at least the features that are common to the different modelling languages) in BPMN or any other notation. The generated BPMN could then be passed on directly to a BPMN engine such that the appropriate executable BPEL

[1] http://www-306.ibm.com/software/integration/wbimodeler/

code would be produced. Similarly, in-place transformations can be carried out and functions that analyse the correctness and soundness of the models can be defined. In this way, errors and anti-patterns that can lead to deadlocks and lack of synchronization are trapped at the modelling phase before the actual executable BPEL code is generated. This would prevent the need to repeat the entire business-driven development life-cycle to adapt models when errors are identified later on in the development process. With such an abstract representation of the entire model, an unlimited number of interpreters and functions can be easily defined to handle the model and extend the language with various out features and functionality. This would not have been possible if a shallow embedded approach was adopted as in the hardware description language, Hawk (Cook, Launchbury, & Matthews, 1998).

To extend the WebSphere Business Modeler Environment, in (Koehler, et al., 2007), IBM presents a *model transformation framework*. Their main objective was to provide an abstract layer over the tool itself such that, by using the Transformation Programming Interface within the framework, specialized developers would be able to use the provided functionalities such as the creation/deletion of elements to easily define new transformations and integrate them into the tool. This serves as a container of plug-ins such that the internal code of the tool would not have to be modified. Besides catering for new transformations, the framework also provides functionality to quality assure the models. However, although it abstracts certain internal implementation details, since the framework is implemented in Java, it still uses first order logic. This means that when models are transformed, analysed or interpreted, developers would still have to take into consideration how the required computations should be implemented rather than simply focusing on what computation is required. Moreover, to carry out checks while the user is constructing or editing the model, linear-time algorithms that do not introduce any significant delay (usually less than a second) such as that presented in (Vanhatalo, Völzer, & Leymann, 2007) would have to be adopted.

In contrast, with our language, we are able to trap errors and incompatibly typed process fragments as early as compile-time. As soon as the model is constructed and compiled, Haskell's type checker would statically analyse the types of the composed elements and generate an error when the types of the connected components are not compatible. This ensures that ill-typed models are identified before any further computation is carried out. Moreover, since the compiler would statically identify the types, the generated compiled code would be optimized and thus executes quicker than that generated by dynamically typed systems.

For this to be possible, we had to *embed our own type system* in that of Haskell. The type system of Haskell is both sound and complete and besides having a type checker, it is also able to infer types at compile-time. In (Leijen & Meijer, 1999) phantom types and polymorphic types are used to ensure the production of type-safe SQL queries for different ODBC database servers. If the request is type-safe, then the types are ignored and the request is represented in terms of constructors of a single abstract data type. Only the typed combinators that are accessible to the user have access to the constructors of this data type. When the request needs to be interpreted then the constructors of one single data type need to be handled. A similar approach was adopted in the latest implementation of Lava (Claessen, 2001), with the only difference that an additional data type is added as an extra layer in between the data type that handles untyped constructs and the type-safety layer that traps ill-typed structures at compile-time. This addition layer ensures that every structure is assigned a non-updateable reference as defined in (Claessen & Sands, 1999). In this way, structures which are shared or contain loops, would be uniquely identified and handled accordingly. The use of non-updateable references through observable sharing is currently the best solution which has been proposed to identify sharing and loops. Since this issue has been encountered in most domain specific embedded languages such as Fran's reactive animation (Elliott & Hudak, 1997) and various hardware description languages, other different solutions were proposed. The simplest solution was that adopted for the hardware description language Hydra (O'Donnell, 1996) where structures were explicitly tagged with a user-defined name. This is rather tedious and names can easily be reused. A monadic approach was later adopted in the first version of Lava (Bjesse, Claessen, Sheeran, & Singh, 1998). However, using monads, the programming style changes and thus definition would look more like imperative programs rather than functions. In our language, we adopted an approach similar to that of Lava (Claessen, 2001). (More technical details about these three different approaches are discussed in Section 2.4.3.3). To enforce our type system, we also made extensive use of type classes (in certain cases with functional dependencies) to be able to carry out checks and computations at the type level and to adopt a generic approach when handling inputs and outputs of different types. In other words, we have applied type classes in various ways as defined in (Jones, Jones, & Meijer, 1997).

Thus, while certain *checks* are carried out at compile-time, others which need to analyse the entire structure, would have to be carried out by specific functions that operate on the abstract representation.

It might not be that obvious how a textual based language can provide the ideal abstraction as other graphical notations such as BPMN or WebSphere Business Modeler. Keeping in mind that it is a domain specific language then it precisely captures the semantics of the domain and uses the appropriate jargon that the domain experts are used to. Moreover, textual definitions can easily be visualised and reasoned about by using *connection patterns*. This approach has been used extensively in hardware description languages, such as Lava (Claessen, 2001). Since functions in functional languages are considered as first class objects, then higher order functions can be used to define such patterns.

To identify the appropriate connection patterns for our language, the workflow patterns proposed in (Russell, Hofstede, Aalst, & Mulyar, 2006) and (Russell, Hofstede, Edmond, & Aalst, 2004) were analysed. These patterns were extracted from various workflow systems and thus indicate how, in most of the cases, the control flow within a process is handled. For instance, one of the basic control patterns, which has been identified, is sequence, which in our language is modelled using the serial composition (`->-`) connection pattern. Similarly, the anti-patterns in (Koehler & Vanhatalo, 2007) were also analysed. Doing so, we defined connection patterns such as `soundCycle`, to produce structured cycles and `sound_inclDecision_Merge` to produce a sound inclusive decision followed by a merge, in terms of exclusive decisions and forks. In this way, connection patterns provide the appropriate abstraction to hide certain implementation details and help users visualize and reason about the model. Less code is required and thus the chance of errors is reduced. This article was also useful to help us identify the features which our language should adopt such as the importance to use gateways rather than input and output criteria to control a flow in a process and reduce errors in models.

An issue which is often encountered when descriptions are defined using a functional approach, is the inability to identify blocks of components during the analysis of the internal abstract structure. In the hardware description language Wired (Axelsson, Claessen, & Sheeran, 2005), a component based approach was adopted, whereas in HeDLa (Pace, 2007), functions explicitly define non-functional information about a group of components. Since in our language, we wanted to package a group of process fragment into one sub-process and consider it as one modelling element, a simpler solution also used in (Caruana & Pace, 2007) was adopted. In this way, a sub-process is another combinator (or function), which is additionally assigned a user-defined name. The problem with this explicit tagging approach is that the user must be careful not to use the same name for different sub-processes. To elevate this problem, users can use the provided function such as `isNameUsed` to check if that name is already used within the process.

4.10. Conclusion

In this chapter, we have presented the domain-specific language which we have embedded in Haskell to construct business process models and to capture the semantics of IBM's WebSphere Business Modeler modelling language. We have also discussed the issues encountered and the approaches adopted to best provide the required features and to provide the ideal abstraction for modellers to easily construct models of a high quality.

We have also compared our language and approaches to other works and concluded that, different from the current modelling languages and notations, with a functional approach, we have managed to develop a language:

- With which various models can be rapidly be produced in a *concise* and *abstract* manner
- Allows users to focus on the required *behaviour* rather than implementation of such the behaviour
- The abstract representation can be *interpreted*, *analysed* or *transformed* in various ways
- Ensures that all the required details, for the *executable* code to be generated, are always provided
- *Quality assures* models by carrying out three types of checks:
  - by *Haskell's* type checker
  - at *construction-time* through its embedded type system
  - by specialised functions that *analyse* the elements in the model

The next chapter investigates different ways how the models in our language can be transformed and quality assured for soundness and how new model transformations and quality assurance techniques can be defined, to extend the language with user specific requirements.

Chapter 5

Transforming and Quality Assuring Models

5.1. Introduction

Once a business process is defined using our language, various computations and systematic analysis can be carried out on the model, irrespective of its internal components. This is in fact one of the advantages of a deep embedded, combinatorial based approach. As soon as a model is defined, an abstract representation is constructed. Various computations can be carried out on any model by analysing its abstract structure. Thus, once a complex process fragment is defined, it would be convenient for the user to view the details and types of the modelling elements which were used, or check whether a particular name has already been assigned to one of the elements. Moreover, the business modeller might want to change specific elements of the process to obtain the required future *'to-be'* process from the current '*as-is'* model. In this case, model transformations would assist the users to automatically transform parts of the model without changing the actual definition. Since in Business-Driven Development (BDD) (Mitra, 2005) the IT solution is directly derived from the business process models, it is important to ensure the quality of the produced models such that data-flow and control-flow errors, which can lead to deadlocks or lack of synchronisation, are trapped at the modelling phase. Our language provides all of these features as functions which carry out some computation on the abstract representation of the model. Other additional functionalities, transformations and quality assurance checks can be defined and easily added to the language, as illustrated in this chapter.

In Section 2, the importance of model transformations and quality assurance in Business-Driven Development is discussed. A brief overview of some of the basic checks and reporting functions, to view details of process fragments, is given in Section 3. The construction of basic model transformations and quality assurance checks in our language are respectively discussed in sections 4 and 5. The construction of composite transformations and the possibility of combining quality assurance with model transformations are then reviewed in Section 6. The final section, before the conclusion, investigates other related works and compares these works to our approach.

5.2. The Importance of Model Transformations and Quality Assurance

Model transformations play a very important role in Business-Driven Development (BDD)[1]. Once a business process model is defined at the modelling phase of the BDD lifecycle, various transformations are applied to be able to refine the abstract model into more concrete representations which are closer to the required executable code. Among various advantages, using such a refinement mechanism ensures that the final IT solution directly satisfies the user's expectations and the business requirements as defined in the business process model.

[1] For more details on Business-Driven Development (BDD), see Section 3.2.2

However, one of the main disadvantages is that, since models are constructed by business analysts who are not IT specialists, the produced models might not be of the required quality. For instance, they might not include all the required details for the IT solution to be derived. To try to avoid such issues, various modelling tools have been developed to assist the users in the construction of such models and ensure that all the details are included. In our language, compile-time checks are carried out to ensure the type-safety of models at construction time. Moreover, since the modelling elements in our language are functions, all the required inputs should be provided for specific elements to be combined and for operations to be carried out successfully.

Modellers can easily introduce data-flow and control-flow errors; for instance, a process fragment expects some data as input which is never provided (data-flow error) or a process waits indefinitely for another process to complete (control-flow error). Such errors can lead to a deadlock (as in the previous examples) or lack of synchronisation (which introduces non-determinism). Ideally a model should be sound whereby, according to (Vanhatalo, Völzer, & Leymann, 2007), it exhibits liveness (something good will eventually happen) and safety (nothing bad will happen) properties. For this reason, various quality assurance techniques such as complete state analysis using model checkers or the use of control-flow analysis heuristics as proposed in (Vanhatalo, Völzer, & Leymann, 2007), should be applied. In other cases, the soundness of models can be verified by identifying patterns and anti-patterns as in (Koehler & Vanhatalo, 2007). In our language, the quality of the models can be assured by using similar techniques. Quality assurance in our language is discussed in Section 5.5.

To assist modellers to rapidly produce good quality business processes, various features can be provided in a modelling language. For instance in our language, using a declarative functional approach based on higher order logic, we were able to provide the modellers with various connection patterns, which make models more readable and abstract away certain implementation details (see Section 4.7). Moreover, parameterized models can easily be defined by the users themselves as discussed in Section 4.8. However, since the requirements of the organisation are constantly changing, the modeller might want to rapidly carry out some transformations on the current *'as-is'* model to produce the future *'to-be'* model. Rather than manually changing the current model and possibly introducing new errors, in-place model transformations should be provided to allow the user to manipulate the source model. Users should be able to compose various transformations to rapidly carry out the required changes without manipulating the original model. However, even though changes on the model are carried out systematically, it is important to assign a number of pre and post conditions to each transformation to ensure that soundness of the produced models. To do so, in our functional language, conditions are defined declaratively by using the provided basic checks (discussed in Section 5.3) and the provided quality assurance functions (discussed in Section 5.5). The construction of composite model transformations with quality assurance is discussed in Section 5.6. Additional primitive model transformations can be defined in our language as illustrated in Section 5.4.

Business process models are the core of the Business-Driven Development (BDD) lifecycle. If the model does not contain all the required details, has errors or is unsound, then the derived IT solution would be incorrect and the entire lifecycle would have to be repeated. Errors are never corrected at the implementation phase. In such cases, the model itself is amended. This ensures the production of models which precisely reflect the underlying implementation. Besides serving for documentation, this is also important to easily manipulate the current solution to satisfy the new demands of the organisation. For this reason, the quality of the models must be assured to avoid the propagation of errors to the next phases of the lifecycle. Thus, pre and post conditions should also be assigned to model transformations to ensure the production of sound models. Model transformations and quality assurance are also respectively discussed in sections 3.4 and 3.5.

5.3. Carrying out Basic Checks on Models

To get the specifications of a process fragment, additional functions are provided. In this way, the modeller would not have to go through the actual definition of the model to check for instance whether a particular modelling element is used or to check whether a name is already assigned to a modelling element. These functions are convenient especially when models are complex or when process fragments defined by other modellers are used. Moreover, some of these basic checks can also be used to define pre and post conditions of model transformations to ensure the production of sound good quality models. Other functions also allow users to compare models.

These functions and checks are provided in *two* modes: 1) to view the result in a user-friendly manner at run-time and 2) to use them mainly in scripts or as a check in pre and post conditions of transformations. The names of the functions defined in the first mode commence with `show`, for example, `showProcSpec` and `showIsNameUsed`, and those defined in the second mode usually start off with `get` or `is`, for example, `getProcSpec` and `isNameUsed`. The main difference between the two is the type of output. A function in the latter mode, for instance, `isNameUsed`, returns a pair containing `True` or `False` as the first element and a string with the properties of the modelling element with that specific name (if such an element is found) as the second element in the pair. In contrast to this, the output of `showIsNameUsed` is an `IO` monad (to be able to display the result on the screen).

Getting Details about the Process

The main function which is provided to get a full report with the specifications of a process is `getProcSpec` or `showProcSpec`. While the latter groups all the modelling elements in the process and displays the details and properties of each one (such as the name and input and output types), the former function returns a record. This record stores the input and output types of the process and lists with the details of the modelling elements in the process. Additional functions are provided to allow the user to obtain the details of just specific modelling elements. For instance, in Listing 5.1, the specifications of a process fragment `pf` are obtained by using `pfProcSpec`. The function `getTasksFromSpec` is then used to retrieve the list with the properties of all the tasks used in the process fragment `pf`.

```
pfProcSpec = getProcSpec pf
tasks = getTasksFromSpec pfProcSpec
```

Listing 5. 1: Using functions `getProcSpec` and `getTasksFromSpec`

The properties of every modelling element are defined as a string. The different attributes in this property string are separated with a hash symbol (‘#’). Since an advanced user might want to retrieve particular attributes to carry out some specific checks or operations, functions such as `getModellingElemType` (to identify the type of the modelling element that is if *Task*, *Start* etc.) or `getNthPropStr` (to get the nth attribute in the property string) or `displayPropertyStr` (to get a string which can be displayed on the screen to view the properties is user-friendly manner) are provided. To get just the type of the modelling elements without the properties, `getModellingElems` (or `showModellingElems`) can be used.

If on the other hand, the user wants to identify elements which have specifically typed inputs and outputs, the functions `inputTypeMatch` (or `showInputTypeMatch`), `outputTypeMatch` (or `showOutputTypeMatch`) and `ioTypeMatch` (or `showIOTypeMatch`) are provided. Besides indicating those elements which precisely match the required types, other elements that have similar types are returned.

Other functions such as `eqPFs` (or `showEqPFs`) and `eqModellingElems` (or `showEqModellingElems`) are also provided to compare the specifications of different processes. Both functions take two process fragments as input. The main differences between these two functions is that while the first compares the types and properties of the modelling elements, the latter just compares the type of the modelling elements. Thus the latter is useful to check if the processes have a similar structure. In both functions, besides a Boolean value, a list of the differences between the elements is returned.

For other provided functions, refer to the tutorial, provided as Appendix A.

Some Important Checks

To assist the user in the construction of models and in the identification of specific properties of a process fragment, basic checks such as the ones in Listing 5.2 are provided.

`isNameUsed`

to check whether a specific name is already assigned to a modelling element in the process

`containsTask, containsRepository, …`

to check whether the process has a specific task, repository, etc.

`isTask, isRepository, …`

to check if a process fragment is a task, repository, etc.

`hasInFlow`

to check whether the process fragment has at least one incoming data or control flow

Listing 5. 2: Basic checks provided in our language

These basic checks can be used to define pre and post conditions of model transformations. Since these checks are essentially functions, composite checks can be defined. Functions such as `isNameUsed` should frequently be used by the modeller to ensure that the name that he shall assign to a modelling element or a sub-process is not already used. Another check which is also important to ensure the construction of good quality models is `hasInFlow`. If a process does not have at least one incoming data or control flow, then even though it retrieves its input data from a repository or constant, it never gets the control to execute and thus is redundant and useless. The function `addDefaultFlow` should then be used to add a default flow.

For other provided functions and checks, refer to the tutorial, provided as Appendix A.

5.4. Constructing Basic Model Transformations

This section illustrates how new basic model transformations can be defined and added to those already available in the language. It should be noted that since these primitive transformations handle the internal primitive constructs of the language, even though it is rather straightforward to define such transformations, programmers that have some experience of Haskell and monads are expected to carry out such extensions to the language.

Since we have opted for a deep embedded approach, once a model in our language is defined using the provided combinators or modelling elements, an abstract representation of the model is internally constructed. Using this internal structure, the models can be analysed by pattern matching the different primitive constructs that make up the model and then handle each one accordingly. Similarly, for the model to be transformed and thus to change specific elements and connections in the model, computation should be carried out on this internal abstract representation.

As discussed in Section 4.4, to detect sharing and loops[1], the observable sharing approach proposed in (Claessen & Sands, 1999) has been employed within our language. Thus, the internal abstract representation of a model is not directly made up of the primitive constructs provided in the language. Instead, it contains referenced objects of type `Ref PrimPF` (where the abstract data type `Ref a` is defined in the module `Ref` (Claessen & Sands, 1999) and `PrimPF` is the abstract data type defining the primitive constructs of our language (refer to Section 4.4)). Thus, to analyse these internal referenced objects and to detect sharing and loops within the models, state monads[2] are used. In this way, if a referenced object is handled for the first time, it is first dereferenced, the actual primitive constructor is obtained and handled accordingly. The state is then updated with a pair containing the reference and the value of this referenced object. The state is essentially a list of pairs, containing the value of previously evaluated referenced objects. The next time this same referenced object needs to be evaluated (usually due to sharing or loops), it would not be evaluated. Instead, the previously computed value would be obtained from the state.

All the required functions to handle these referenced objects and the state that is updated with the new values, are abstracted away in the module `EvalPPF` (*Evaluate Primitive Process Fragment*). Thus, if an advanced user, who has some experience of Haskell, wants to define some other primitive basic transformations which can later be flexibly composed with others to produce more complex transformations, this module should be imported and the function `evalPF` (*Evaluate Process Fragment*) defined in this module should be used. This takes as inputs three functions and the process fragment that should be evaluated (together with its input). The first function is the function that should pattern match each primitive construct and handle them accordingly. The second function defines the computation that should be carried out if the referenced object has been previously defined and thus its value is in the state. (If no specific computation is required and the value should simply be retrieved from the current state, then the built-in Haskell function `id` can be used). The third function defines some operation that should be carried out on the final result. (Similar to the second function, if no specific operation is required, the built-in Haskell function `id` can be used).

For instance, to define the primitive model transformation, `renameRep`, to rename a particular repository, the function `evalPF` is used in the following manner:

```
renameRep c_nm n_nm pf x = evalPF (renameRepPPF c_nm n_nm, id, id) pf x
```

The first two inputs are respectively the current and the new name of the required modelling element and the function `renameRepPPF` is the function that should pattern match the primitive constructs and handle them accordingly. In this case, only repositories should be handled; the rest can be ignored. Listing 5.3 illustrates the implementation of this function for some of the primitive constructs.

```
renameRepPPF:: String->String-> PrimPF -> OpIfInMem(PrimPF) -> State (Mem PrimPF) PrimPF

renameRepPPF _ _ (Token) _ = return (Token)

renameRepPPF c_nm n_nm (FlowCon rppf) op = do
                                    ppf <- evalRPPF rppf (renameRepPPF c_nm n_nm) op
                                    return (FlowCon ppf)

renameRepPPF c_nm n_nm (Repository r_nm t gs) op = do
                                   egs <- evalGStruct gs (renameRepPPF c_nm n_nm) op
                                   let ret = if (c_nm == r_nm)
                                                then (Repository n_nm t egs)
                                                else (Repository r_nm t egs)
                                   return (ret)
```

Listing 5. 3: Part of the implementation of `renameRepPPF` defined to rename elements in the process

[1] The possible approaches to handle sharing and loops in embedded languages are discussed in Section 2.4.3.3
[2] More details about monads is available is in Section 2.2.7

From Listing 5.3, it should be noted that if the primitive construct is a repository, the current name of the repository (`r_nm`) is compared with the supposedly current name of the required repository (`c_nm`) such that if the required repository is found, then its name is changed accordingly. When defining such a function as in Listing 5.3, the functions `evalRPPF` and `evalGStruct` are used to respectively evaluate inputs of type `RefPrimPF` and `GStruct RefPrimPF` (these types were discussed in Section 4.4). The primitive constructor `FlowCon` is not important for this evaluation and thus it is not evaluated and this construct is returned unchanged. It should also be noted that a pattern match on the primitive construct `Token` is defined. This constructor is automatically attached to the inputs of the fragment to stop the recursion.

The renaming transformation is one of the basic transformations which are already defined in our language. The actual transformation in our language is able to rename any modelling element and not simply repositories as in Listing 5.3. In fact, one generic internal function `renameModellingElementPPF` is defined. This is then referenced by other functions such as `renameRep`, which are accessible to the end-user.

Although the underlying details which handle referenced objects are abstracted away, only programmers who have some experience of Haskell and monads are expected to define additional basic transformations and thus extend the language. Checks can be incorporated within the definitions or else defined as pre and post conditions before and after the actual transformation is carried out. These conditions can easily be defined by users who are not necessarily functional programmers, before the actual transformation functions such as `renameRep`, are referenced. Moreover, using these pre-defined basic transformations, composite transformations can easily be defined by the users themselves. To ensure the quality of the produced models, users should declaratively define pre and post conditions, by composing basic checks and quality assurance functions, discussed in the following section. The construction of composite transformations with quality assurance is investigated in Section 5.6.

5.5. Quality Assurance

As illustrated in Section 4.4, besides the language terms, we have also embedded our own type system in Haskell. In this way, since Haskell is a statically strongly typed language, it is possible to carry out compile-time checks to ensure that only compatibly typed process fragments are combined together into a more complex model. However, such checks are not enough to ensure the quality and soundness of a model. To carry out such checks, various modelling elements that make up the model would have to be analysed.

For instance, if the outgoing branches of an exclusive decision are closed with a join (Figure 5.1), then a deadlock is introduced. This is so, since, while only one of the branches of an exclusive decision executes, the following join waits for both branches to complete before control is passed on to the next modelling element. This means that the join will wait indefinitely for an input, which shall never be provided. Similarly, if a fork is followed by a merge (Figure 5.3), the merge passes out control as soon as one of the branches executes, with the consequence that, since all the branches of the fork are evaluated, the elements after the merge are executed for each of the branches. This leads to lack of synchronization. These are in fact two of the anti-patterns which have been identified in (Koehler & Vanhatalo, 2007). In the article, the authors also suggest a corresponding valid pattern for each anti-pattern. Thus, if a decision is followed by a merge (Figure 5.2) or a fork is followed by a join (Figure 5.4), then the model can immediately be considered sound. Similar patterns and control-flow analysis heuristics are recognised in (Vanhatalo, Völzer, & Leymann, 2007). This means that by systematically identifying such patterns and anti-patterns, it is possible to immediately detect sound and unsound models. In other cases, as defined also in (Vanhatalo, Völzer, & Leymann, 2007), if a process fragment cannot be categorised according to one of these patterns or heuristics, the soundness of the model cannot really be confirmed using this approach. In such cases, complete state analysis using model checkers would be the appropriate approach to guarantee soundness. However, a known issue with such an approach is the state explosion problem.

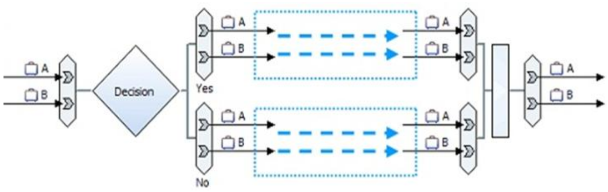


Figure 5.1: An Anti-Pattern - unsound process fragment that leads to a deadlock (Koehler & Vanhatalo, 2007)

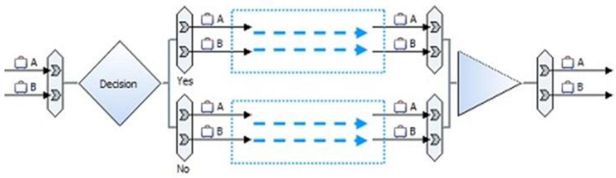


Figure 5.2: A sound pattern –suggested as a solution to the anti-pattern in Figure 5.1 (Koehler & Vanhatalo, 2007)

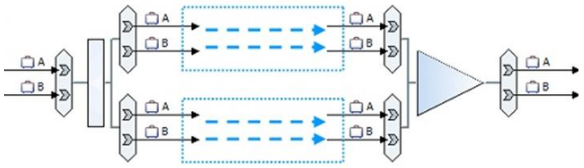


Figure 5.3: An Anti-Pattern - unsound process fragment that leads to a lack of synchronization (Koehler & Vanhatalo, 2007)

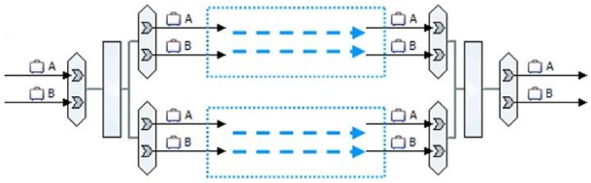


Figure 5.4: A sound pattern –suggested as a solution to the anti-pattern in Figure 5.3 (Koehler & Vanhatalo, 2007)

Noting that in most of the cases the soundness of a model can easily be determined by identifying patterns and anti-patterns in the model, it was decided that, for this first version of our language, it would be sufficient to check for control-flow and data-flow errors by looking for patterns and anti-patterns such as those identified in (Koehler & Vanhatalo, 2007) and (Vanhatalo, Völzer, & Leymann, 2007). As discussed in the previous section, for the model to be analysed, the internal abstract representation made up of the primitive language constructs or rather referenced objects would have to be investigated. Moreover, as illustrated in Listing 5.3, using monads, the style of programming is no longer that intuitive and straightforward. This makes the code less readable and limits the type of analysis that can be carried out on the model. For instance, using this approach, the models can only be analysed from back to front using a depth first search traversal. Besides this, as already mentioned in the previous section, for a user to define functions in this manner, some knowledge of Haskell and monads is considered mandatory.

Representing the Model as a Directed Graph

Noting that during analysis, the actual model is not really changed as in the case with transformations, functionality is provided to generate *a directed graph* made up of a set of vertices and edges to represent the underlying internal abstract representation of the model. The abstract data type `Graph` has been defined (as illustrated in Listing 5.4) to keep the properties of this graph.

```
data Graph = Graph [Vertex] [Edge] (StartVertices, EndVertices)

data Vertex = Vertex Int String
type Edge = (Vertex, Vertex)

type StartVertices = [Int]
type EndVertices = [Int]
```

Listing 5. 4: The abstract data type `Graph`

In this way, when the function `getGraph` is executed for a particular process fragment, the internal abstract representation of the model is analysed as in Listing 5.3, and a graph of type `Graph` is returned. As illustrated in Listing 5.4, this *graph* is made up of a list of vertices, a list of edges and a pair containing the start and the end vertices. A *vertex* is simply made up of a unique identification number and a string containing the properties of that specific modelling element, as defined in Section 5.3. Thus, the actual attributes within this property string are separated with a hash symbol ('`#`') and provided functions such as `getModellingElemType` and `displayPropertyStr` should be used to easily retrieve specific attributes and handle this string. An *edge* is simply a pair containing two vertices, whereby the first vertex is executed before the second. We have also included *start vertices* and *end vertices*. Rather than the actual vertices, the unique identification number of the vertices is kept in these two lists. While the start vertices represent those modelling elements whose inputs are collectively equivalent to the inputs of the process fragment, the end vertices are those vertices whose outputs are collectively equivalent to the outputs of the process fragment. In this way, the modelling elements represented by the vertices in the graph, can be analysed from left to right (i.e. from the start to the end vertices) or from right to left (i.e. from the end to the start vertices).

Various operations and analysis can be carried out on such a simple graph by anyone not necessarily an expert Haskell programmer. Moreover, the elements in the model can easily be analysed depth first or breadth first. To further facilitating the handling of such a model, functions such as `getNextVerticesOnLeft`, `getNextVerticesOnRight`, `getVertexInList`, `getVerticesInList` and others are also provided. If the modeller wants to see the details of this graph in a user-friendly manner then the function `displayGraphDetailsForPF` can be invoked at run-time. An interesting feature of `getGraph` and `displayGraphDetailsForPF` is that besides the process fragment, as input, a Boolean value (`openSP`) is also expected. With this Boolean value, the user can specify whether a sub-process should be considered as one modelling element (`openSP = False`) or whether the sub-process should be opened up such that a vertex for each of the internal modelling elements is constructed, (as if that fragment was never packaged into a sub-process) (`openSP = True`). A user should note that in the graph the vertices representing the modelling elements are numbered from left to right, top to bottom, as defined in the model.

The functions provided in our language to assure the quality of the models, carry out analysis on the graphs generated for the models. In this way, following the heuristics defined in (Vanhatalo, Völzer, & Leymann, 2007), it is possible to identify whether modelling elements are in *sequence* or whether the fragment can be classified as a *well-structured* or *unstructured sequential* or *concurrent branching* fragment. If none of these, then the soundness of the model cannot be confirmed. Similarly, using the graph, *structured cycles* can be identified. Various other control-flow and data-flow patterns and anti-patterns defined in (Koehler & Vanhatalo, 2007) can also be identified by carrying out analysis on such a graph. The following sub-sections illustrate how fragments containing such patterns can easily be identified by analysing the graph generated for the model.

Modelling Elements in Sequence

To identify whether a fragment simply contains a single flow with modelling elements connected in sequence (as in Figure 5.5), it is enough to sort the list of edges and check whether the second vertex in a pair is equivalent to the first vertex of the next edge in the list.

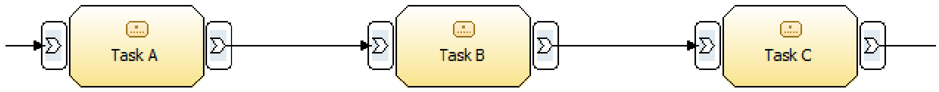


Figure 5.5: A fragment containing a single flow with modelling elements connected in sequence

In cases where a modelling element contains more than one input or output (as in Figure 5.6), the list would contain duplicate edges and even though the elements in the fragment are in sequence, the previous approach would not work. For this reason, it is important for duplicate edges to be removed before carrying out the check explained above, on the edges of the graph.

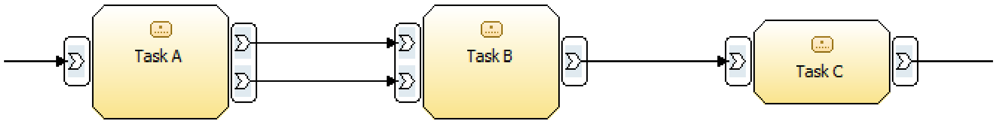


Figure 5.6: A fragment with elements connected in seqence –`Task A` has 2 output, `Task B` has 2 inputs

The function `inSequence` is defined in our language. Given a process fragment, a Boolean is returned to indicate whether elements in the fragment are in sequence.

Structured and Unstructured Sequential and Concurrent Branching Fragments

To be able to detect such fragments, vertices in the graph can either be analysed from start to end or end to start. Although with such a directed graph it is also possible to analyse the modelling elements from the start and from the end vertices simultaneously, since to detect such patterns, it is important to identify the types of gateways that are reachable from a specific vertex, then we have decided to analysis the vertices from end to start. This approach can easily be adapted to handle vertices from start to end. It should be noted that according to (Vanhatalo, Völzer, & Leymann, 2007), *sequential branching fragments* are those made up of decisions and merges, and *concurrent branching fragments* are those that include forks and joins.

Let's consider the identification of *sequential branching*. The vertices in the elements must be analysed from end to start such that as soon as a merge is encountered, a check should be carried out on all the fragments connected to its inputs to identify the decisions and forks that are reachable from this merge. (The first vertex of each fragment can be identified by invoking the function `getNextVerticesOnLeft`). Once the reachable gateways on each incoming branch are obtained, a check is carried out to identify the decision that is commonly reachable by all the branches.

If no such decision is identified and no other gateway is reachable by the branches, then the soundness of the model cannot be identified. Else, if forks are reachable by some of the branches then keeping in mind that a fork followed by a merge leads to lack of synchronisation, then the fragment is considered unsound.

If a particular decision is reachable from all the branches and the number of outgoing branches of the decision (which can be obtained by invoking the function `getDecisionNoOfBranches`; the string with the properties of the decision should be obtained from the corresponding vertex and passed on as input to the function) is equivalent to the number of incoming branches of the merge (as Figure 5.7), the fragment can be considered sound and classified as a *structured sequential branching fragment.*

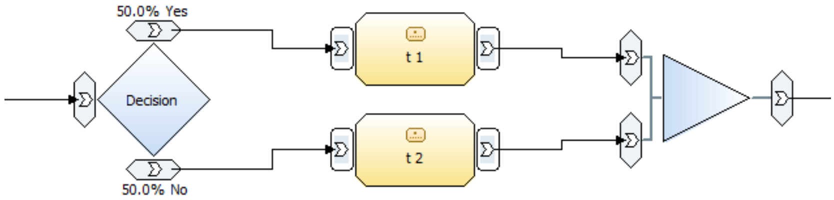


Figure 5.7: A structured sequential branching fragment

If on the other hand, a particular decision is reachable from all the branches but the number of outgoing branches of the decision is not equivalent to the number of incoming branches of the merge, then some other decisions or forks (which are not respectively closed with a merge or a join) are present between the decision and merge along some of the branches. If decisions are present (as in Figure 5.8), then they can safely be closed with the same merge and the model can still be considered sound. In this case, the fragment is classified as an *unstructured sequential branching fragment*.

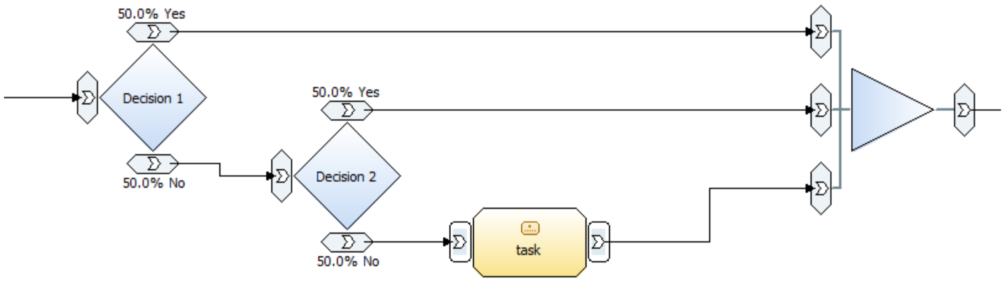


Figure 5.8: An unstructured sequential branching fragment

If forks are present (as Figure 5.9), then it is certainly unsound, as the fragment would lack synchronisation.

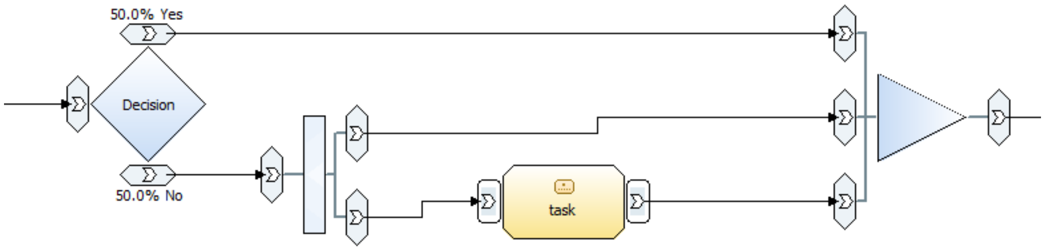


Figure 5.9: An unsound fragment - lacks synchronisation

To detect the types of gateways that lie between the main decision and merge, the list of decisions and forks that are reachable on the specific branch should be analysed; the vertices in the list located before the vertex representing the decision should be check. If at least one of these is fork, then it is unsound.

The other modelling elements in the fragment would have to be analysed in a similar manner. It should be noted that, it is also important to identify the type of the decision. *Inclusive decisions* followed by a merge, lead to lack of synchronisation (if more than one of the outgoing branches if executed) and an inclusive decision followed by a join can lead to a deadlock (if only one of the outgoing branches is executed). Thus such fragments should always be considered unsound.

Similarly, *concurrent branching fragments* (as figures 5.10 and 5.11) can be identified by trying to capture joins and forks instead of merges and decisions. If decisions lie between the detected fork and join, the fragment would be unsound, since decisions closed with a join, lead to a deadlock; the join would wait indefinitely for both branches to executed, when actually only one of the outgoing decision branches of an exclusive decision is evaluated.

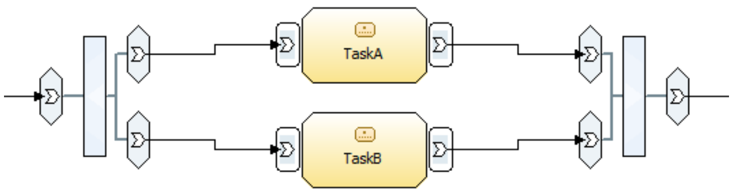


Figure 5.10: A structured concurrent branching fragment

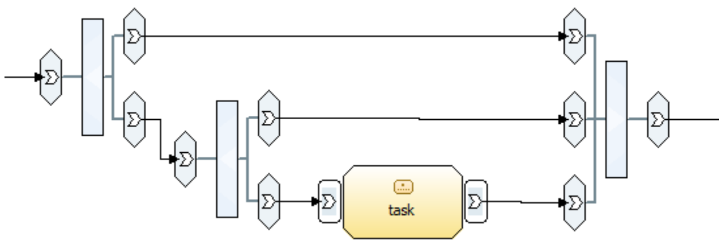


Figure 5. 11: An unstructured concurrent branching fragment

The function `isSound` is defined in our language. Given a process fragment, `Sound`, `Unsound` or `Unknown` is returned together with a string to indicate the type of the fragment.

Structured Cycles

Since the vertices in the graph as numbered from left to right, top to bottom according the position of the modelling elements in the process fragment, a fragment containing a *cycle* can easily be identified, whenever an edge, whose first vertex has a larger numeric identifier than that of the second, is detected. To verify that this truly refers to a previous modelling element, a function can be defined to check whether a path exists from this particular element to itself. By checking the type of this element and the type of the previous element which is re-referenced, it is possible to identify whether the cycle is structured or not.

The type of the element can be identified by using the function `getModellingElemType` (providing it as input the string with the properties of the modelling element represented by the vertex). If an exclusive decision is connected to a previous merge (as in Figure 5.12), then the cycle is *structured*. However, if any other modelling elements are used, then the cycle is considered *unsound* (as in figures 5.13, 5.14, 5.15). This is due to the fact that an inclusive decision or a fork linked to a previous merge would introduce lack of synchronisation (Figure 5.14), leading to an explosion of uncontrolled iterations of the internal process fragment. On the other hand, a fork or an exclusive decision linked to a previous join would introduce a deadlock (figures 5.13, 5.15); while the join waits indefinitely for both incoming branch to pass on control, the branch coming from the fork or the decision, can never pass on control to the join, since they are never executed before the join is evaluated for the first time.

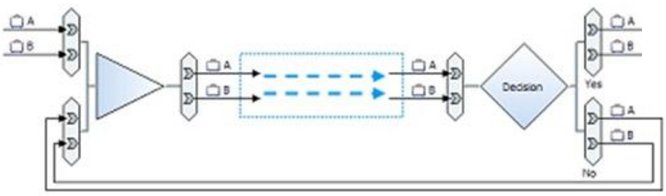


Figure 5. 12: A structured sound cycle

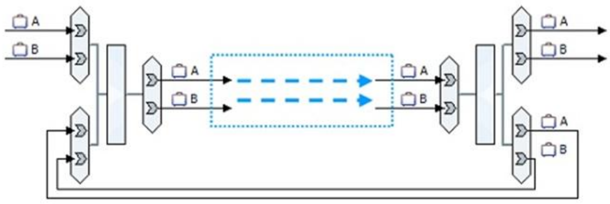


Figure 5.13: An unsound cycle - has a deadlock

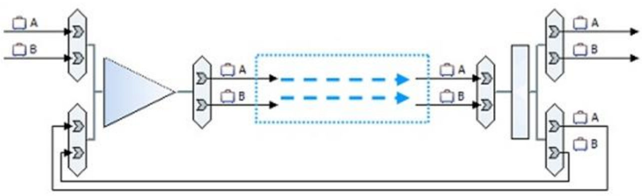


Figure 5. 14: An unsound cycle - lacks synchronisation

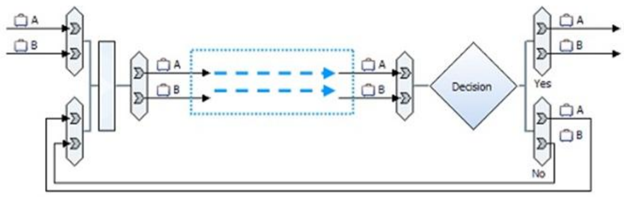


Figure 5.15: An unsound cycle - has a deadlock

Final Remarks

One of the main advantages of such a graph is that different from the approach used in Section 5.4 and in Listing 5.3, the graph explicitly provides a list of edges depicting the connectors in the model. These connectors are not explicitly modelled in our language using primitive constructors and thus by analysing the internal abstract representation, it is not really intuitive and easy to identify all the fragments connected to a particular modelling element.

Noting the ease with which models can be analysed using such a directed graph, our language can be improved by allowing users to define the very basic primitive transformations by using such a graph and then provide some functionality in the language to internally translate the defined transformation depicted on the graph, to the actual transformation on the internal abstract representation of the model. In this way, all the implementation details would be hidden away from the user, thus providing the ideal abstraction for the language to be extended with new transformations and new quality assurance checks, without requiring any particular expertise of Haskell and monads.

5.6. Constructing Quality Assured Composite Transformations

Although model transformations are convenient for modellers to rapidly transform models from the current '*as-is*' to the future '*to-be*' processes, it is important to ensure the soundness and the correctness of the produced models. If a sound model is applied some transformation such as the substitution of a task with another which has different typed inputs or outputs, an ill-typed model would be produced. To ensure the correct application of transformations, pre and post conditions should be defined.

Since our language adopted a functional approach, and since the provided basic and quality assurance checks are essentially functions, it is possible to declaratively define these conditions and compose them into more complex checks, in a similar way as composite functions are defined. For the transformation code to be more readable, it is suggested that, if basic checks are added to the language, the function that is made accessible to the user, would first check that the pre-conditions are satisfied and then invoke the function that carries out the actual transformation. The post-conditions can later on be verified. This is how the provided primitive transformations in our language are defined. If the conditions are satisfied and the transformation is carried out successfully, a triple containing `Successful`, a message and the transformed model is returned. Else, `Failed`, an error message and the untransformed model is returned. In cases where the transformed model leads to lack of synchronization, even though such models are not sound, they are not essentially incorrect and lack of synchronisation might have been introduced on purpose. For this reason, `Warning`, an appropriate warning message and the transformed model is returned. The user would then decide whether to use the transformed model or not.

Let's consider the repository renaming transformation `renameRep` discussed in section 5.4, Listing 5.3.

```
renameRep c_nm n_nm pf x
```

To ensure that the process fragment `pf` contains a repository named `c_nm`, the basic check `containsRepository` can be utilized. To ensure that the new name which shall be assigned to the repository is not already used within the process fragment, the function `isNameUsed` should be used. Since none of the modelling elements or connections are changed, it is not really essential to check the soundness of the model after the transformation is carried out. Thus the quality assured transformation would be defined as illustrated in Listing 5.5.

```
renameRepQA c_nm n_nm pf x = let
                                (nameAlreadyUsed, elemWithSameName)= isNameUsed n_nm pf
                                (hasRep, _) = containsRepository c_nm pf
                             in
                                if (nameAlreadyUsed)
                                   then (Failed, "The name "++ n_nm ++" is already assigned
                                         to the modelling element "++ (displayPropertyStr
                                         elemWithSameName)++" defined in the process
                                         fragment", pf x)

                                   else if (not hasRep)
                                         then (Failed, "The process fragment does not have a
                                               repository named "++c_nm, pf x)

                                         else (renameRep c_nm n_nm pf x)
```

Listing 5. 5: The repository renaming transformation with conditions to assure the quality of the transformed model

In Listing 5.5, it should be noted that appropriate error messages can be defined to precisely illustrate why such results are returned. For instance, besides a Boolean value, `isNameUsed` also returns the specifications of the element within the process which has the same name as that intended to be used for the transformation. If the conditions are not satisfied, the actual transformation function is not invoked and the untransformed model (together with its input) is returned.

Let's consider another basic transformation provided in our language and let's assume that pre and post conditions still need to be defined to quality assure the transformed model:

```
substituteSubProcess  csp_nm  nsp indexL  pf  x
```

The aim of this transformation is to substitute sub-processes named `csp_nm` (current sub-process name) in a process fragment `pf` with a new sub-process `nsp`. If the sub-process in the process fragment is used more than once, the user can specify which of these should be replaced. Assuming that the sub-processes are numbered from left to right, top to bottom starting off with value 1, the index of the sub-processes that should be replaced should be defined in the list `indexL`. The input argument `x` represents the input of the process fragment.

Primarily it is important to check whether the new sub-process `nsp` is truly a sub-process. For this reason, the basic check `isSubProcess` can be used. It is also important to check whether the indices defined in `indexL` are valid, that is the indicated indices must be between 1 and the number of occurrences of such a sub-process in the fragment. To ensure the production of a type-safe model, it is also essential to ensure that the input and output types of the new sub-process are precisely equivalent to that of the current sub-process. In addition to these checks, it is also useful to ensure the soundness of the model before and after the transformation. If the model is not sound then the transformation should not be carried out. If the pre-conditions are satisfied then the transformed model in this case should be sound. However, to guarantee this, the soundness should be checked once again. Thus the quality assured transformation would be defined as illustrated in Listing 5.6.

```
substituteSubProcessQA  csp_nm  nsp indexL  pf  x =

      let
          (isUnTransPFSound, untransSoundMsg) = isSound transPF

           noOfSPinPF= getNoOfSubProcesses csp_nm pf
           typeOfSPinPF = getTypeOfSubProcessInPF csp_nm pf

           typeOfNewSP = getTypeOfSubProcess nsp

          (wasTransDone,transMsg,transPF) = (substituteSP c_nm n_nm pf x)

          (isTransPFSound, transSoundMsg) = isSound transPF

       in
           if (isUnTransPFSound == Unsound)
               then (Failed, "The untransformed model is already unsound
                      ("++untransSoundMsg ++")", pf x)

               else if (not isSubProcess)
                     then (Failed,"The provided process fragment is not a sub- process",pf x)

                     else if ((minimum(indexL) < 1) || (maximum(indexL) > noOfSPinPF)
                           then (Failed,"The indices of the sub-processes that should be
                                  substituted must be defined as an numeric value between 1 and
                                  "++show (noOfSPinPF), pf x)

                           else if (typeOfSPinPF /= typeOfNewSP)
                                 then (Failed, "The input and output types of the new
                                        sub-process are not equivalent to those of the
                                        current sub-process ", pf x)

                                 else if ((isTransDone==Successful)&&(isTransPFSound==Unsound))
                                       then (Failed, "The transformation produces an unsound
                                              model ("++transSoundMsg ++")", pf x)

                                       else (wasTransDone, transMsg, transPF)
```

Listing 5. 6: The sub-process substitution transforming with conditions to assure the quality of the transformed model

Different from the quality assured transformation is Listing 5.5, the soundness of the model is assured before and after the model is transformed.

If on the other hand, the user wants to define *composite transformation* and some pre and post conditions related to this transformation, he can do so in a declarative manner as illustrated in Listing 5.7.

```
tApplySpecialTerms = task "Apply Special Terms to Order" (biTOrder :-> biTOrder)

transOrderProcessing pf x =

   let
      (hasSPOrderVerif, _) = containsSubProcess "Order Verification" pf
      (hasTaskRejectOrder, _) = containsTask "Reject Order" pf

      transf1@(wasTransDone, transMsg, transPF)  =

            if (hasSPOrderVerif)
               then (renameSubProcessQA "Order Verification" "Certify Order" pf x)
               else if (hasTaskRejectOrder)
                    then (substituteTaskQA "Reject Order" tApplySpecialTerms [1] pf x)
                    else (Succeeded, "", pf x)

       transf2 =  renameDecisionQA  "Is Order Valid?"  "Is Order Certified?" pf x

   in transf2
```

Listing 5. 7: Defining the quality assured composite transformation `transOrderProcessing`

The complex transformation defined in Listing 5.7, is made up mainly of two transformations (`transf1` and `tranf2`), which are carried out in sequence. The first transformation is a branching type of transformation and as illustrated it can easily be defined by using the basic checks provided in the language. Depending upon the results returned by these checks, the appropriate transformation is carried out. These checks serve as pre-conditions. Thus if a sub-process named "`Order Verification`" is found, it is renamed to "`Certify Order`". Else, if the process contains a task or tasks named "`Reject Order`", then the first task named "`Reject Order`" (just the first since `indexL=[1])` is substituted with another task `tApplySpecialTerms`. Due to the internally defined pre-conditions of the substitution transformation, the transformation is only carried out if the current task takes an order as input and produces an order as output. Finally the decision "`Is Order Valid?`" is renamed to "`Is Order Certified?`".

It should be noted that the pre and post conditions of the basic model transformations are abstracted away. Thus the user can simply focus on the conditions required to successfully define the required composite transformation. The resulting transformation is readable, easy to comprehend and moreover, ensures the quality of the generated transformed model.

5.7. Related Work

After identifying the importance of model transformations in Model Driven Architectures such as Business-Driven Development, various types of model transformation frameworks have been proposed over the years to carry out different types of transformations to ensure the production of the required executable code from an initial model illustrating the specifications of the system. IBM has recently developed a transformation framework for its modelling tool, *IBM WebSphere Business Modeler* to carry out in-place transformations on business process models (Koehler, et al., 2007).

Their main objective was to create a framework with which various types of model transformations can be implemented. Using such transformations the modellers would be able to rapidly carry out the required changes on the models. After obtaining an immediate feedback about the results of the transformed models, the user is allowed to decide whether to persist the modified model. They want the framework to fully integrate the transformations as part of the modelling tool, such that the user would be able to view them as normal editing commands and rapidly perform transformations with just a few mouse clicks. Their transformation framework provides a *Transformation Programming Interface* (TPI) to allow specialized developers to extend the modelling tool with new transformations. The TPI provides features such as the creation/deletion of modelling elements to allow developers to easily define new transformations and integrate them into the tool. This serves as a container of plug-ins and an abstract layer over the modelling tool itself such that, when new transformations are defined, the internal code of the tool does not need to be accessed. The architecture of this transformation framework is illustrated in Figure 5.16.

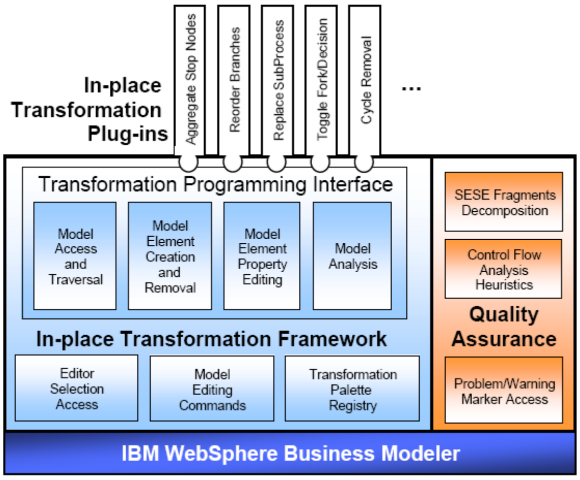


Figure 5.16: The architecture of IBM's transformation framework for WebSphere Business Modeler (Koehler, et al., 2007)

This means that specialised developers are limited to the features provided by the TPI. They cannot define their own primitive and basic transformations that operate on the model. Thus for the TPI to be extended with new features, other specialized developers who have access to the underlying internal code of the tool, would have to be employed.

Although this framework tries to abstract away specific implementation details, since the framework is implemented in Java, it still uses first order logic. This means that when models are transformed, interpreted or analysed, developers would still have to consider how the required computations should be implemented rather than simply focusing on the required behaviour. This is in fact evident in implementation provided for the '*stop node aggregation*' transformation in (Koehler, et al., 2007). For this reason, it is not possible for users who are not really programmers to define their own transformations. Moreover, it is not possible for the modellers to define composite transformations. The framework only provides a simple recording feature which allows users to define and generate a sequence of transformations. Still, no parameters can be defined for such transformations. If such a framework was used to allow users to define composite iterative or branching transformations, complex transformation rules would have to be exposed at the business level. This is due to the fact that pre and post conditions assigned to transformations, to ensure the quality of the transformed models, involve elaborate model analysis.

As argued in (Koehler, et al., 2007), a *declarative approach* is required to be able to define such pre and post conditions. In (Koehler, Hauser, Sendall, & Wahler, 2005), pre and post conditions of out-place transformations were represented in the Object Constraint Language and used successfully to refine the graphical models into the executable BPEL code. As defined by Backus's Turing Award paper (Backus, 1978), the main reason why a declarative and functional approach results to be more effective than other imperative approaches, is that users are able to abstract away from the implementation details and focus on what operations are required rather than how such operations should be implemented. This means that this approach brings about other advantages than simply those identified in (Koehler, et al., 2007) and (Koehler, Hauser, Sendall, & Wahler, 2005).

In contrast to IBM's modelling language and its transformation framework, our language adopts a functional approach based on higher-order logic. Thus, by defining the required models, checks and operations as functions, users are able to focus on the required behaviour rather than the implementation of such computations. By adopting a deep embedded approach, models can be transformed, interpreted and analysed by handling the internal abstract representation of the model. As shown in Section 5.4, new basic transformations can easily be defined by any programmer who has some experience of Haskell and monads. Thus, the users are not limited to just the built-in primitive basic transformations, but they can define and extend the language with their own basic transformations. Moreover, as shown in Section 5.6, modellers can easily define composite checks as pre and post conditions of transformations and composite transformations by using the basic transformations provided in the language. Thus, with our language users are allowed to declaratively define their own transformations and associate pre and post conditions without necessarily requiring specialised developers. Moreover, any kind of composite transformations be it sequential, branching, iterative or even parametric, can be defined.

Additional basic and quality assurance checks can easily be defined. To facilitate the analysis of process fragments, a *directed graph* is generated for the model. This approach is in fact similar to that adopted in (Vanhatalo, Völzer, & Leymann, 2007), whereby the soundness of fragments is verified by checking the soundness of their corresponding workflow directed graphs. Using such graphs, models in our language can be analysed in various ways and traversed depth first or breadth first. Moreover, the connections between the elements are explicitly defined by the edges in the graph.

Various *quality assurance techniques* to be able to identify control-flow and data-flow errors have been investigated over the years. To detect all types of control-flow and data-flow errors, *complete state analysis algorithms* can be employed. Model checkers were used successfully for business process models in (van der Aalst, 2000), (Mendling, Moser, Neumann, Verbeek, van Dongen, & van der Aalst, 2006). Besides verifying that all the possible execution paths satisfy particular properties, these tools also return a trace to indicate where an error was encountered. However, this is only possible through the construction of the entire state space of the process model, which can grow exponential in size and lead to the state-space explosion problem. To mitigate this problem, a technique used in compiler theory, whereby processes are decomposed as a hierarchy of *Single-Entry-Single-Exit* (SESE) fragments is proposed in (Vanhatalo, Völzer, & Leymann, 2007). In this way, rather than model checking the entire process, SESE fragments of the process are checked individually. In the paper, a number of *linear-time control-flow analysis heuristics* were also identified. Thus if the soundness of the fragment cannot be determined using these heuristics, a model checker would not be used. This technique was used for IBM's modelling tool and the functionality, to decompose a process into SESE fragments and to apply the heuristics, was incorporated in the transformation framework. In this way they managed to assure the quality of the model in real-time, while the user is producing the model, without any significant delay (less than a second), thus providing immediate feedback and return diagnostic information, to trap errors as early as possible. The observations carried out in (Vanhatalo, Völzer, & Leymann, 2007) (which led to the definition of these heuristics) overlap with the *anti-patterns* identified in (Koehler & Vanhatalo, 2007).

For this first version of our functional modelling language, we have opted to define quality assurance functions that detect specific patterns and anti-patterns and classify them as *Sequence fragments* and *Structured or Unstructured Sequential or Concurrent branching fragments* as in (Vanhatalo, Völzer, & Leymann, 2007). Besides returning whether these models are sound and unsound and the type of the fragment, they also return the details of the modelling elements that make the model unsound so that the user can easily identify where the required corrections should be carried out. Ideally, our language should be able to generate the required structure to pass on the model to a model checker. In this way, it would be possible to carry out the required complete state analysis in cases were such patterns and heuristics are not able to classify fragments as sound or unsound. This is in fact one of the features, which is intended to be added in other future versions of the language. By carrying out the required analysis on the generated directed graph representing the model, other anti-patterns which are identified in (Koehler & Vanhatalo, 2007) should easily be defined and added to the language.

5.8. Conclusion

This chapter illustrates how basic model transformations, quality assurance checks and quality assured composite model transformations can be defined in our language. By adopting a declarative approach, composition of basic checks to define pre and post conditions and composition of defined transformations, can easily be defined by any user who is not necessarily a programmer or an IT specialist.

The next chapter investigates a number of case studies and evaluates different models defined in our language and in one of the currently available modelling tools, IBM WebSphere Business Modeler.

Chapter 6

Evaluation and Case Studies

6.1. Introduction

In this chapter, a number of models created with IBM WebSphere Business Modeler Advanced v6.0.2[1] are used as case studies to evaluate our functional modelling language. These models are constructed using different approaches and each one is later on analysed. The use of connection patterns provided in our language and their importance to ensure modularity and abstraction is discussed. The benefits brought about by parameterized models are also investigated. In this way, it is possible to identify the effectiveness of our language. Other more comprehensible evaluation techniques are also suggested. These techniques are more complex and require more time, resources and domain experts. Thus, it was not really possible to carry out such evaluation for this first prototype of the language.

The first two case studies are based on two models obtained from the sample projects that are available with the tool[1]. The first model is obtained from the *External Claims Assessor Management (ECAM) project*, whereas the second is obtained from the *ABC project*. These models are very realistic and they were purposely created to help modellers learn how to use IBM's tool. Thus, it was thought that these models would be ideal to help a modeller learn how to define real world processes in our language. For this reason, the processes in these projects as well as those in the third sample project, *Quickstart Finance project*, were defined using our language and provided as Appendix B. The third case study in this chapter, considers a model which was intentionally constructed to illustrate the importance of connection patterns at different levels of abstraction. Finally, two examples of parameterized models are investigated in case study 4.

Although a comprehensible explanation of the construction of these models is provided, more details about the features and modelling elements in our language are available in the tutorial provided as Appendix B. Similarly, if the user is not sure about the semantics of the elements used in IBM's tool, then Section 3.3.2 should be referenced for more details. In certain cases, just a short code snippet is discussed. For the complete code see Appendix B.

6.2. Case Study 1

The model which shall be considered in this case study has been obtained from one of the sample projects that come along with the tool. The selected project is named *External Claims Assessor Management (ECAM) project*. As the name suggests, this project contains business processes of an automobile insurance company. One of the most important processes for this organisation is the '*Auto Claims Handling'* process. This was defined in the sample project and shall be used for this case study (see Figure 6.1). The main aim of this case study is to analyse the different ways how models and modelling elements can be defined using our language, and which of these would be most feasible, for a modeller who is not an IT specialist and who might already be familiar with IBM's modelling tool. At the end, the process fragment is packaged into a sub-process.

[1] http://www.ibm.com/developerworks/websphere/zones/businessintegration/roadmaps/modeler/roadmap_advanced.html

Figure 6.1

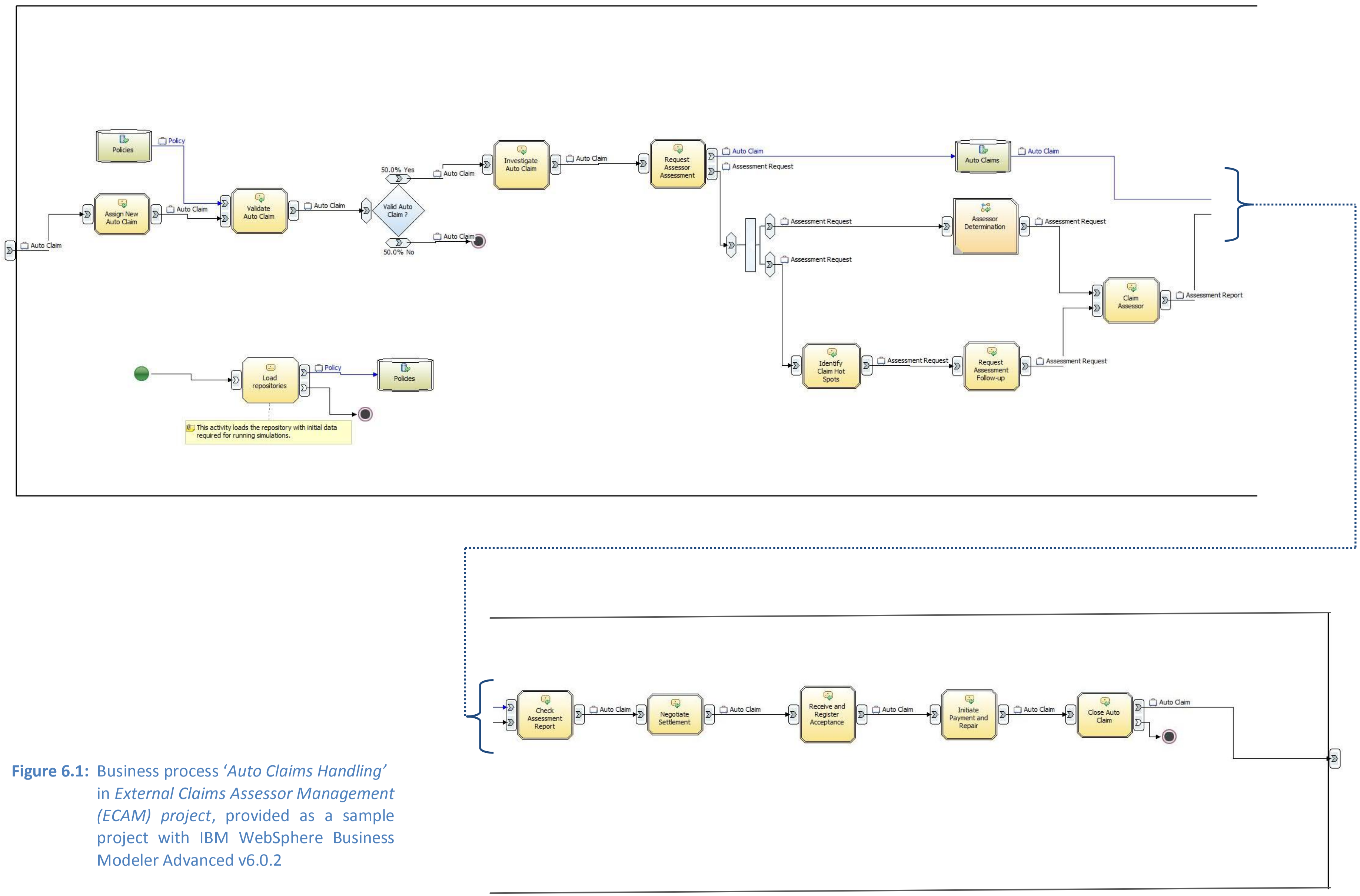


Figure 6.1: Business process *'Auto Claims Handling'* in *External Claims Assessor Management (ECAM) project*, provided as a sample project with IBM WebSphere Business Modeler Advanced v6.0.2

Defining Business Items

The first items that need to be identified and defined are the business items. From Figure 6.1, it should be noted that none of the built-in basic types are used. Instead, `Policy, Auto Claim, Assessment Request` and `Assessment Report` are used as user-defined business items (hence complex types). In IBM's tool, these new types are defined once and used by any process within the project. In a similar way, using our language, these items are first defined and then used for any process by simply importing the module where they were defined. The code in Listing 5.1 illustrates how business item `Policy` is defined. Others such as `Auto Claim, Assessment Request` and `Assessment Report` should be defined in a similar manner.

```
newtype Policy     =  Policy ()     deriving Typeable
type TPolicy        =  BI Policy
instance ComplexType (TPolicy)     -- or  (BI Policy)
biTAssRec  = dType :: TPolicy      -- or   BI Policy
```

Listing 6.1: Defining the new complex type (business item) `Policy` for the process in Figure 6.1

As illustrated in Listing 6.1, the actual type is defined with the first line of code. To indicate that the defined complex type is a business item, then the construct `BI` should be used. To abstract such details and avoid writing `BI Policy` to refer to business item `Policy`, a type synonym is defined (line 2). Line 3, classifies the type as complex and the final line defines the first class object that should be used as a normal argument to refer to that type. Note that in the last two lines, `-- or (BI …)` denotes that if a type synonym (such as `TPolicy`) is not defined, then the actual type (in this case, `BI Policy`) would have to be used. Although four lines of code are required to define one single user-defined type, this makes the system flexible enough to accept new types, classify them (in this case, as complex types and business items) and abstract details by using type synonyms. Yet, these are defined once and used for any process.

Defining Tasks

The next step is to define the required tasks. Similar to complex user-defined types, these can be defined once and used for any process by simply importing the module where they are defined. In this way, these tasks can be considered as global. The user must be aware that at this stage, the input source and output target type should not be defined. The task should not be seen in the context of the process where it shall be used. Instead, the user should focus on the behaviour of the activity and simply specify the input and output data types (rather than the connection type). For this reason, tasks must be defined in a similar manner as those in Listing 6.2. Note that the input and output data types of the tasks are defined in terms of the first class objects that represent the type, and the inputs and outputs are separated with `:->`

```
tAssignNewAC = task "Assign New Auto Claim"   (biTAutoClaim :-> biTAutoClaim)
tVerifyAC    = task "Verify Auto Claim"       ((biTPolicy, biTAutoClaim) :-> biTAutoClaim)
tReqAutoAss  = task "Request Auto Assessment" (biTAutoClaim :-> (biTAutoClaim, biTAssReq))
```

Listing 6.2: Defining some of the tasks for the process in Figure 6.1

As illustrated in Listing 6.2, no particular knowledge of Haskell is required. The user does not need to know anything about the type system of the host language and how types are defined. Instead, first class objects representing the types, are used as normal arguments, in a similar way as properties of activities are defined in IBM's tool. If a user is familiar with Haskell, it is should be intuitive that `:->` is related to typing in a similar way as `->` is used in the type signatures of functions (the only difference is that `:->` handles first class objects (values) rather than actual types).

Defining Repositories

Before a modeller can start the actual construction of the process, the required repositories `'Policies'` and `'AutoClaims'` should be defined. Since there is a process fragment with the task `'Load Repositories'` (Figure 6.2), then this indicates that `Policies` is a global repository. As noted in Figure 6.1, this repository is not updated with new data and it should be defined and filled up by other processes. For our process to have access to this data, then its contents would have to be passed on as an input argument. In this way, the fragment in Figure 6.2 (extracted from Figure 6.1) is not needed. No other task is needed to ensure that a global repository is loaded with data.

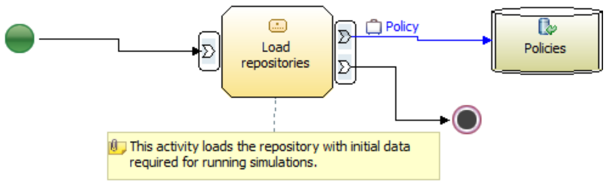


Figure 6.2: Process fragment to load global repository `'Policies'` (part of Figure 6.1)

Contrary to this, repository `'AutoClaims'` in Figure 6.1, is not loaded with data using fragments such as Figure 6.2. However, from the properties of this repository in IBM's tool, it is notable that this is also a global repository. Thus, it is defined, filled up with data and used for the first time, within this process. Later on, its content is generated as an output of the process. In this way, other processes would have access to the contents, without requiring process fragments such as Figure 6.2. The code to initially define these two repositories is in Listing 6.3. The type of items that the repository shall store is defined by using arguments representing the type.

```
rPolicies   = repository "Policies"   biTPolicy
rAutoClaims = repository "AutoClaims" biTAutoClaim
```

Listing 6.3: Defining repositories for process in Figure 6.1

Defining the Input Source and Output Target types of the Tasks to be used in the process

Once the modeller ensures that the general and global items are defined, he can then start constructing the model. If during construction, any other tasks, business items or any other element is required, if global, then he can define them and add them to the rest, else define them locally.

To be able to use the task within the process, the input source and output target types should be defined. Listing 6.4, illustrates how the types of three of the tasks (one with a single input and a single output; another with two inputs and one output; the final with one input and two outputs), can be defined. The user should be aware that `repCI`, `constCI` and `flowCI` are used respectively to specify an input source as a repository, a constant or flow. Similarly, `repCO` and `flowCO` are used to respectively indicate an output target of repository or flow.

```
ttAssignNewAC    =  flowCO . tAssignNewAC. flowCI
ttVerifyAC x y   =  flowCO (tVerifyAC (repCI x , flowCI y))
ttReqAutoAss x   = (repCO y, flowCO z)
                        where (y,z) = tReqAutoAss  (flowCI x)
```

Listing 6.4: Defining input source and output target types of some tasks that are used in the process in Figure 6.1

These definitions can be expressed in a more concise and easier to comprehend manner by using the provided connection patterns `<| … |>` such that even the types of the third task would be defined in a single line, as illustrated in Listing 6.5.

```
ttAssignNewAC    =   flowCI          <| tAssignNewAC |>   flowCO
ttVerifyAC       =  (repCI, flowCI) <| tVerifyAC |>      flowCO
ttReqAutoAss     =   flowCI          <| tReqAutoAss |>   (repCO, flowCO)
```

Listing 6.5: Defining input source and output target types of tasks by using connection patterns `<|` and `|>`

In both cases, compile-time checks are carried out to ensure that only valid input source and output target types are assigned. Once these typed tasks are defined then they can be used to construct the model.

Constructing the Model

Once typed tasks (as in Listing 6.4 or 6.5) are defined, the modeller can start connecting the elements and constructing the model. It should be noted in Figure 6.1, that the process is made up of two process fragments. The smaller fragment (which is also depicted in Figure 6.2) is solely used to load the global repository `'Policies'` before this is used within the process. If the modeller forgets to include such a fragment when defining the model in IBM's tool, then the repository would not be loaded with the expected data and thus lead to a deadlock; fragments which make use of the data stored in this repository would end up waiting indefinitely for data which is never provided.

In our language, the modeller does not need to include any such fragment. Instead, since the process is modelled as a function, it can only get information from its surrounding as an input, and thus the contents of such as repository should be passed in as an input argument. For this reason, this fragment is not included in the definitions of the process in listings 6.6, 6.8 and 6.9.

Since this process contains the sub-process `'Assessor Determination'`, which makes use of the data stored in the global repositories `'Assessor Records'`, `'Assessor Selection Rules'` and `'Assessor QOS History'`, then similar to the global repository `'Policies'`, the contents of these repositories must also be passed on to this process, as input arguments. Thus, the actual input of the process defined in Figure 6.1, is `(irAssRec, irAssSel, irAssQOSHist, irPolicy, x)` whereby `irAssRec, irAssSel, irAssQOSHist` and `irPolicy` are respectively the contents of global repositories `'Assessor Records', 'Assessor Selection Rules', 'Assessor QOS History'` and `'Policies'`, and `x` is the actual data input of type `TAutoClaims`.

In the following sections, the construction of this model using basic modelling elements and connection patterns is analysed.

***Construction 1:** Constructing the Model using only Basic Modelling Elements*

Listing 6.6 illustrates the code that is required to define the process in Figure 6.1 in our language, using only basic combinators as modelling elements.

Listing 6.6

```
pfAutoClaimsHandling (irAssRec,irAssSel,irAssQOSHist,irPolicy,x) =  let

                 -- First 2 tasks
                 ottAssignNewAC = ttAssignNewAC x
                 ottVerifyAC  = ttVerifyAC (irPolicy, ottAssignNewAC)

                 -- Decision with 2 sequential tasks and a stop
                 (oValidAC, oNotValidAC) = exclDecision "Valid Auto Claim?"
                                           (branchProp eYes_FlowIsValidAC 0.5,branchProp eNo_FlowIsValidAC 0.5) ottVerify
                 ostop_NotValidAC_AC = stop (oNotValidAC :: PF (Flow TAutoClaim))
                 ottReqAutoAss @ (ottReqAutoAss_AutoClaim, ottReqAutoAss_AssReq)  = (ttReqAutoAss . ttInvestigateAC) oValidAC

                 -- Auto Claims repository fill up
                 orAutoClaim = rAutoClaims ottReqAutoAss_AutoClaim

                 -- Fork fragment
                 (o1fork, o2fork) = fork ottReqAutoAss_AssReq
                 ospAssessorDetermination = spAssessorDetermination (irAssRec,irAssSel,irAssQOSHist,o1fork)
                 ottReqAssFollowUp = (ttReqAssFollowUp . ttIdentifyClaimHotSpots) o2fork
                 ottClaimAssr = ttClaimAssr (infConType ospAssessorDetermination, ottReqAssFollowUp)

                 -- Final 4 sequential tasks
                 ottCloseAC = (ttCloseAC . ttInitiatePayAndRepair . ttReceiveRegisterAcceptance . ttNegotiateSettlement .
                               ttCreateAssRep)(orAutoClaim,ottClaimAssr)

                 -- Stop attached to final task
                 (opfAutoClaimsHandling, oCtrl) = (ctrl_dataFlowSplitter ottCloseAC)::(PF(Flow TAutoClaim),PF (Flow Ctrl))
                 ostop_ctrl = stop oCtrl

            in   (opfAutoClaimsHandling, orAutoClaim, ostop_NotValidAC_AC, ostop_ctrl)
```

Listing 6.6: The required definition to model the process in Figure 6.1 using basic modelling elements in our language

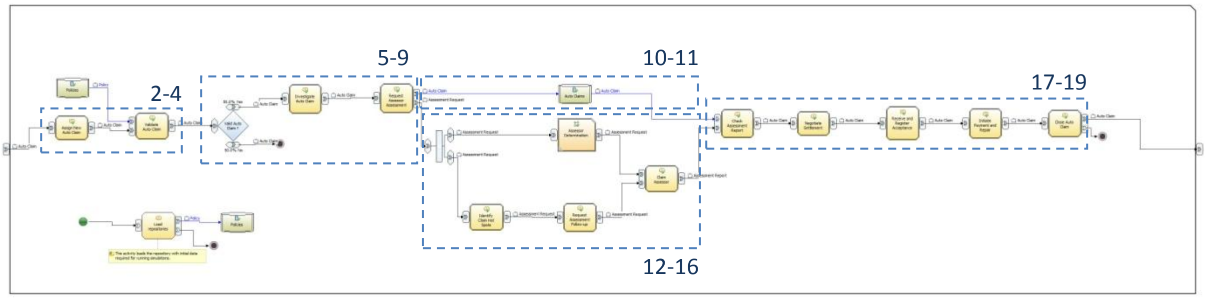


Figure 6.3: The aim of this diagram is to illustrate which lines of code in Listing 6.6 handle fragments of the process. To view the details of the process refer to Figure 6.1

As illustrated in Listing 6.6, two of the inputs, the content of the global repository `'Policies'` (`irPolicy`) and the data item of type `AutoClaims` (`x`) are initially passed on to task `'Assign New Auto Claim'` (`ttAssignNewAC`). The next task that should be executed is "`Validate Auto Claim`" (`ttVerifyAC`) and following that is a 2-branch *exclusive decision*. The properties of the branches of this decision are defined as `(branchProp eYes_IsValidAC 0.5,branchProp eNo_IsValidAC 0.5)`. The function `branchProp` is used to define each decision branch, such that the first argument is possibly the lambda expression which represents the condition that should be satisfied for the branch to be selected and the second is the probability assigned to the branch. In our language, the probability is defined as a value between 0 and 1 (rather than a percentage between 0 and 100% as in IBM's tool). At run-time a check is carried out to ensure that the sum of the probabilities of all of the branches is 1. The code in Listing 6.7 is used to define the expressions assigned to each of the decision branches. In this case, the expressions were defined as simple as possible. The input type of these expressions must be equivalent to the input type of the decision (since the branch can only be chosen, based on the data that is provided as input).

```
eYes_IsValidAC, eNo_IsValidAC :: (PF (Flow TAutoClaim)) -> Bool
eYes_IsValidAC = const True
eNo_IsValidAC  = const True
```

Listing 6.7: Defining expressions `eYes_IsValidAC` and `eNo_IsValidAC` assigned to the decision branches in line 7 of Listing 6.6

The output of the second decision branch is passed on to a stop node (line 8, Listing 6.6). Since, the data type of the outgoing branches of the decision and the input of the *stop* node is not known unless inferred, the second outgoing branch is explicitly typed by using `:: PF (Flow (BI AutoClaim))`. The output of the stop node `ostop_NotValidAC_AC` is passed out as output (line 23, Listing 6.6) to ensure that the details of this branch are not lost during analysis of the model and to avoid dangling outputs (dangling outputs are identified as a bad modelling practise in part 2, scenario 3 of (Koehler & Vanhatalo, 2007)).

To execute tasks `"Request Assessor Assessment"` in sequence with task `"Investigate Auto Claim"`, the *functional composition operator* `(.)` which is built-in in Haskell, is used (line 9, Listing 6.6). The output of the first decision branch is then passed on to this composite function. The five final tasks in the process are put in sequence in a same manner (lines 18 and 19, Listing 6.6). Using this function, this sequence of tasks is expressed in a concise manner.

In line 11 (Listing 6.6), the global repository `'Auto Claims'` stores the data produced as the first output of task `"Request Assessor Assessment"`. Its content is then passed on to task `"Check Assessment Report"`. Since it is a global repository, its contents (`orAutoClaim`) is also passed out as as output (line 23, Listing 6.6). If it was a local repository then this would not have been necessary. The modeller should be aware that, if such a repository is defined by never filled in with data, then an error would be generated at compile-time. This would avoid redundancy and ensure the production of less cluttered models (a bad modelling practise defined in part 2, scenario 3 of (Koehler & Vanhatalo, 2007)). If on the other hand, the global repository already contains some data, then its previous content would have to be passed on as another input to the process and a new repository for this particular process would have to be defined. To specify that this data is the previous contents of the global respository, then the function `previousRepContent` should be used, such that, the actual data passed on as input to the repository, is actual the output of this function. This repository would then be filled up with its previous content and the new data generated by activities in the process. The user must realize that the repository after all is another pure function and so it is not possible to have global variables

Lines 12-16, define the *fork* fragment and the tasks connected to it. One of the outgoing branches is connected to a sub-process named "`Assessor Determination`". In this case, it is assumed that this sub-process has already been defined and is accessible to this process. This is used as any other task. Since this sub-process requires the current of the global repositories `'Assessor Records'`, `'Assessor Selection Rules'` and `'Assessor QOS History'`, then these are passed as input to this process and then diverted to this sub-process as other input argument.

The last element in the process is a *stop node* attached to the last task. If the modeller does not want to package this process fragment (`pfAutoClaimsHandling`) into a sub-process, then this node is not required and the user would still be able to incorporate this process fragment in other processes. Although such process fragments defined as functions provide the same abstraction as a sub-process in IBM's tool, still this fragment cannot be considered as one single element. Thus, during analysis the interpreter would not view this fragment as a sub-process but as a set of connected elements and hence, it is not possible not to consider the internal structure of the fragment. Moreover, the output of a sub-process should not indicate control. However, defining the process as a simple fragment represented as a function then these outputs are important to keep a trace of all the elements in the fragment. For this reason, if the modeller wants to define a real sub-process then the modelled process fragment would have to be packaged in a sub-process. For this to be possible, the fragment must have a valid stop node, to return the required data on termination of the process. For this reason, in line 21 a flow splitter is introduced to split the data flow into a data and control flow. To indicate how the flow is split up, the output of the splitter is typed by using :: `(PF (Flow (TAutoClaim)) , PF (Flow Ctrl))`. A stop node is then attached the control flow (line22, Listing 6.6).

Construction 2: *Constructing the Model using Connection Patterns*

To define process fragments which are more readable and concise, connection patterns are provided in our language. The two most basic and important patterns are *serial composition* (`->-` or `serial`) and *parallel composition* (`-|-` or `par`). Thus, the process blocks indicated in Figure 6.4 can easily be defined using these connection patterns. Other connection patterns for other elements such as decision and fork can be also be used, as illustrated in Listing 6.8.

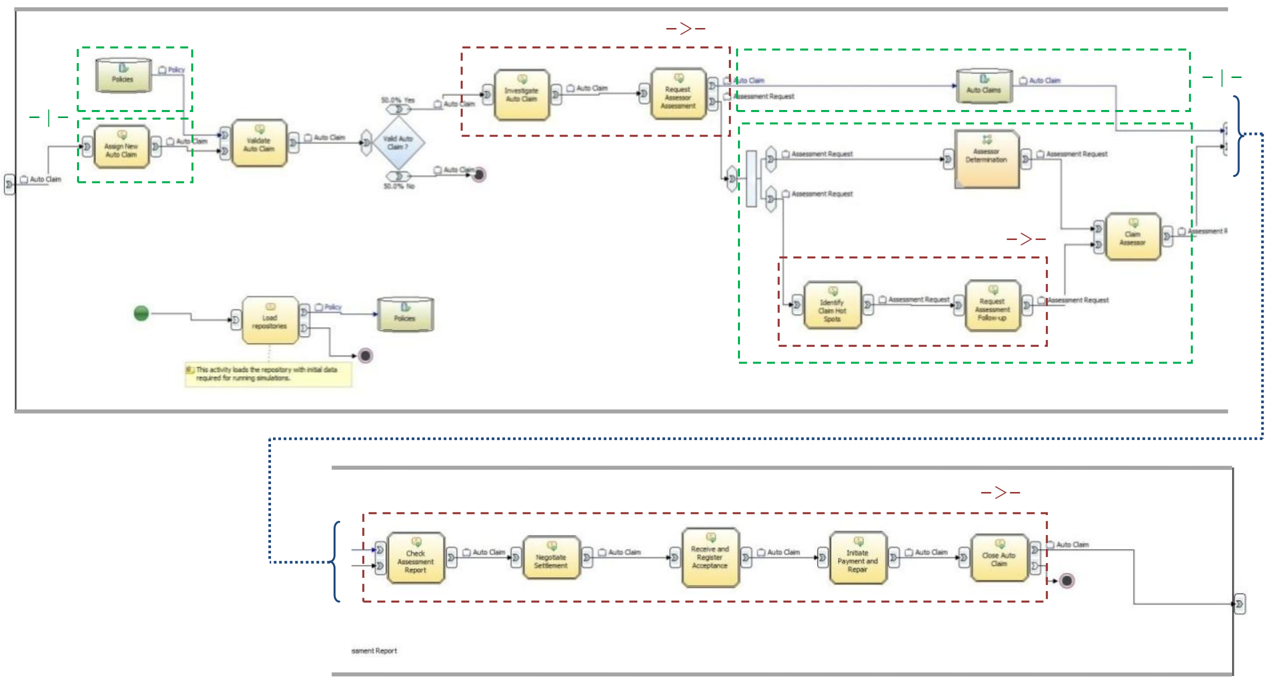


Figure 6.4: Process in Figure 6.1 - Blocks which can be defined using −>− are marked in red, whereas blocks which can be defined using − | − are marked in green. To view the details refer to Figure 6.1

Using connection patterns, instead of Listing 6.6, the code in Listing 6.8 is enough to express the process in Figure 6.1. The process is decomposed into two main fragments, that is, the main stream and the fragment attached to the upper branch of the decision. When these fragments are defined as `spf1` (lines 3-5) and `spf2` (lines 9-13), the input and output arguments are not mentioned anywhere in the definition. In this way, by using connection patterns, the user can focus solely on the required behaviour and thus how the elements (which are essentially functions) should be connected. This means that composite process fragments can be defined (as composite functions), and processes can be modelled in a similar manner as with other conventional tools. Although in Listing 6.6 (lines 15 and 18), the functional composition `(.)` operator was used to put a number of tasks in sequence in a more concise and readable manner, still the tasks in the definition appear in the reverse order. The flow in a process flows from left to right and so it is more intuitive for a sequence of activities to be defined in this order. Thus, `(ttCreateAssRep ->- ttNegotiateSettlement ->- ttReceiveRegisterAcceptance ->- ttInitiatePayAndRepair ->- ttCloseAC)` (line 12, Listing 6.8) is more readable than `(ttCloseAC . ttInitiatePayAndRepair . ttReceiveRegisterAcceptance . ttNegotiateSettlement . ttCreateAssRep)` (line 18, Listing 6.6). If the user wants to immediately spot blocks within the process, then it might be more convenient to use the function `serialC` rather than the operator (->-), for instance, `serialC (ttCreateAssRep, ttNegotiateSettlement, ttReceiveRegisterAcceptance, ttInitiatePayAndRepair, ttCloseAC)`. The precise order of activities would still be denoted.

The connection patterns `-<|..|>=` are used to depict an exclusive decision and `fork_withOutPFs` is used to capture the fork fragment, that is the fork together with the process fragments attached to each of its outgoing branches. Since the second decision branch leads to a stop node and since the type of the flow flowing out of the decision branch would have to be specified, then `stopBranch biTAutoClaim` (line 5, Listing 6.8) can easily be used instead of `stop (oNotValidAC :: PF (Flow TAutoClaim))` (line 8, Listing 6.6).

Since the subprocess `'Assessor Determination'` requires additional input (that is the contents of the repositories `'Assessor Records'`, `'Assessor Selection Rules'` and `'Assessor QOS History'`, which are passed as inputs to this process), besides that produced by the previous fragment, the function provided in our language `getSingledInputsFragment` is used (line 9, listing 6.8). This curries the input function such that it converts it from a function that accepts a tuple as an input (in this case of size 4) to a function which accepts the inputs as single arguments (in this case as four arguments). In this way, by providing the first three inputs, that is the contents of the repositories, we are still able to connect this fragment with others by using the provided connection patterns, such that the remaining input would be obtained from the fragment connect to it.

The parallel composition connection pattern `(-|-)` is also useful to put fragments in parallel. The resulting composite function can be considered as a single fragment and can be attached to any other component which is compatible to its input or output types (as in line 3, Listing 6.8). Since a repository is simply defined as another function, then it can be connected to other fragments in a similar way as any other component. This is illustrated in Listing 6.9, with repository "`Auto Claims`". However, doing so, then it is not possible to retrieve the output and thus the content of this repository and produce it as an output of this process. If the repository was local then this would have been fine and would have made our definition more concise and elegant. However, as stated earlier, the repository "`Auto Claims`" is global and thus is important for its contents to be generated as output. This is in fact the reason why the process was split up into two fragments.

Another convenient connection pattern is used in line 13, Listing 6.8. Usually the output of a `ctrl_dataFlowSplitter` is inferred automatically depending upon the fragments that are connected to it. In other cases, the output type is explicitly specified, as in line 21, Listing 6.6. When modelling the process using connection patterns, then `-//=` is used, to introduce a `ctrl_dataFlowSplitter`. However, since a stop node can accept as input, a data or a control flow and since, in this case, the input needs to be restricted to a control flow, instead of line 13, Listing 6.8, `-//= (id, (id :: (PF(Flow Ctrl)-> PF(Flow Ctrl)))->- stop)` could be defined. To abstract away these details, the connection pattern `-//=.` can easily be used (as in line 13) to: 1) split the input data flow into a data and a control flow, 2) allow the data flow to pass out without being modify and 3) terminate the control flow (produced as the second output) with either an end or a stop node; in this case, a stop node.

Although the definition of the model in Listing 6.8 is rather concise, before such a process fragment is defined, the user is expected to specify the input source and output target types of every single task as illustrated in Listing 6.5. This is rather time consuming and tedious for the modeller. In fact, in IBM WebSphere Business Modeler, most of these types are automatically inferred. For this reason, rather than using the ordinary serial composition (`->-` or `serial`), lazy serial composition (`->>-` or `serialL`) can be used, such that the modeller would not have to specify the input source and output target types of the activities. If on the other hand, the user wants to specifically emphasis some of these types, then he can do so and still use lazy serial composition. Thus the process in Figure 6.1 can be defined as in Listing 6.10.

Since, in Listing 6.10 the input source and output target types of activities are automatically inferred, the initial definitions of the tasks (without the connection types specified) are used. However, to ensure that the first of input of task `"Verify Auto Claim"` is obtained from a repository, either the typed task `ttVerifyAC` is used, or else, the `noActivity` function provided in our language, is utilized, as illustrated in Listing 6.10. The main aim of this function is to allow a particular typed flow to pass through without modifying it. Thus, by specifying the data type and the required connection type by using the connection pattern `<|` and the function `repCI`, we can guarantee that only data obtained from such a repository is fed as input to the task. Thus, by using the second approach, the user can define the process using only untyped tasks, and then specify the input types of the process very elegantly using `noActivity`. This is then put in parallel with the task `"Assign New Auto Claim"` to ensure that the appropriate input for the task `"Verify Auto Claim"` is generated.

Since the data flowing through the decision, in this case, is of type `PF TAutoClaim`, then the input type of the expressions assigned to the decision branches, must also be set to `PF TAutoClaim` (rather than `PF(Flow TAutoClaim)`, as defined in Listing 6.7). Similar to `serialC`, `serialLC` (for `->>-`) is also provided.

Listing 6.8

```
pfAutoClaimsHandling (irAssRec,irAssSel,irAssQOSHist,irPolicy,x) =  let

            -- Main stream
            spf1 = (id -|- ttAssignNewAC) ->>- ttVerifyAC
                   -<|("Valid Auto Claim?",(branchProp eYes_IsValidAC 0.5,branchProp eNo_IsValidAC 0.5))|>=
                  (ttInvestigateAC ->>- ttReqAutoAss, stopBranch biTAutoClaim)

            ospf1 @ ((ottReqAutoAss_AutoClaim, ottReqAutoAss_AssReq), ostop_NotValidAC_AC) = spf1 (irPolicy,x)

            orAutoClaim = rAutoClaims otReqAutoAss_AutoClaim

            -- Upper decision branch
            spf2 = (fork_withOutPFs ((getSingledInputsFragment spAssessorDetermination) irAssRec irAssSel irAssQOSHist,
                  ttIdentifyClaimHotSpots ->>- ttReqAssFollowUp) ->>- ttClaimAssr) ->>-
                 (((getSingledInputsFragment ttCreateAssRep) orAutoClaim) ->>-
                 ttNegotiateSettlement ->>- ttReceiveRegisterAcceptance ->>- ttInitiatePayAndRepair ->>- ttCloseAC)
                 -//=. stop
            ospf2 @ (opfAutoClaimsHandling, ostop_ctrl) = spf2 ottReqAutoAss_AssReq

        in  (opfAutoClaimsHandling, orAutoClaim, ostop_NotValidAC_AC, ostop_ctrl)
```

Listing 6.8: The required definition to model the process in Figure 6.1 using connection patterns in our language

Listing 6.9

```
pfAutoClaimsHandling (irAssRec,irAssSel,irAssQOSHist,irPolicy,x) =  let

        --Main stream
        main = (id -|- ttAssignNewAC) -|- ttAssignNewAC) ->- ttVerifyAC -<|("Valid Auto Claim?",(br eYes 0.5,br eNo 0.5))|>=
               (udb,stopBr biTAutoClaim)

        -- Upper decision branch
        udb = (ttInvestigateAC ->- ttReqAutoAss) ->- (rAutoClaims -|-
              (fork_withOutPFs ((getSingledInputsFragment spAD) irAssRec irAssSel irAssQOSHist, ttIdentifyClaimHotSpots ->-
                ttReqAssFollowUp) ->- ttClaimAssr)) ->- (ttCreateAssRep ->>- ttNegotiateSettlement ->- ttReceiveRegisterAcceptance ->-
               ttInitiatePayAndRepair ->- ttCloseAC) -//=. stop

   in   main (irPolicy,x)
```

Listing 6.9: The required definition to model the process in Figure 6.1 using connection patterns as in Listing 6.8 with the differnce that it does not produce the content of repository `'AutoClaims'` as an output of the process

Listing 6.10

```
pfAutoClaimsHandling (irAssRec,irAssSel,irAssQOSHist,irPolicy,x) =  let
                  -- Main stream
                  spf1 = ((repCI <| noActivity biTPolicy) -|- tAssignNewAC) ->>- tVerifyAC
                          -<|("Valid Auto Claim?",(branchProp eYes_IsValidAC 0.5,branchProp eNo_IsValidAC 0.5))|>=
                         (tInvestigateAC ->>- tReqAutoAss, stopBranch biTAutoClaim)
                  ospf1 @ ((otReqAutoAss_AutoClaim, otReqAutoAss_AssReq), ostop_NotValidAC_AC) = spf1 (irPolicy,x)

                  orAutoClaim = rAutoClaims otReqAutoAss_AutoClaim

                  -- Upper decision branch
                  spf2 = (fork_withOutPFs ((getSingledInputsFragment spAssessorDetermination) irAssRec irAssSel irAssQOSHist,
                         tIdentifyClaimHotSpots ->>- tReqAssFollowUp) ->>- tClaimAssr) ->>-
                        (((getSingledInputsFragment tCreateAssRep) (infConType orAutoClaim)) ->>-
                        tNegotiateSettlement ->>- tReceiveRegisterAcceptance ->>- tInitiatePayAndRepair ->>- tCloseAC)
                        -//=. stop
                  ospf2 @ (opfAutoClaimsHandling, ostop_ctrl) = spf2 otReqAutoAss_AssReq

            in  (opfAutoClaimsHandling, orAutoClaim, ostop_NotValidAC_AC, ostop_ctrl)
```

Listing 6.10: The required definition to model the process in Figure 6.1 using lazy serial composition in our language

Packaging the Process as a Sub-Process

Once the process fragment is defined, it can be packaged by using the function `packageSubProcess` as illustrated in Listing 6.11.

```
spAutoClaimsHandling = packageSubProcess "Auto Claim Handling" pfAutoClaimsHandling
```

Listing 6.11: Packaging process `pfAutoClaimsHandling` into a sub-process

In this way, different from the process fragment `pfAutoClaimsHandling`, `spAutoClaimsHandling` can be handled as a single modelling element (even at the analysed phase) and the control outputs produced by the stop nodes would be abstracted away within the sub-process. This is possible since a flow splitter and combinator are respectively added to the inputs and outputs of the internal process fragments such that the incoming flow is split into the required data and control flows (to initiate all the process fragments), and the outputs of these fragments are then combined into one data or control flow and generated as output of the sub-process. More technical details about the tagging and packaging of process fragments into sub-processes are defined in Section 4.6. The types of the data flowing through process fragment `pfAutoClaimsHandling` and the sub-process `spAutoClaimsHandling`, are illustrated in respectively Figure 6.5 and 6.6. Note that `spAutoClaimsHandling` in Figure 6.6 contains a flow combinator to combine the output flows of the internal process fragment, `pfAutoClaimsHandling`, and to return just the data flows. `spAutoClaimsHandling` does not have a flow splitter since the inputs of the internal fragment are precisely the inputs of the sub-process.

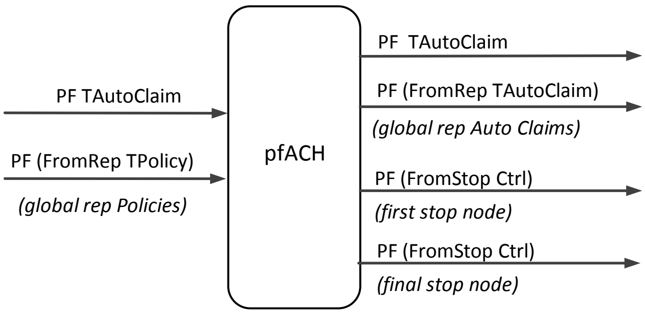


Figure 6.5: The data types flowing through process fragment `pfAutoClaimsHandling`

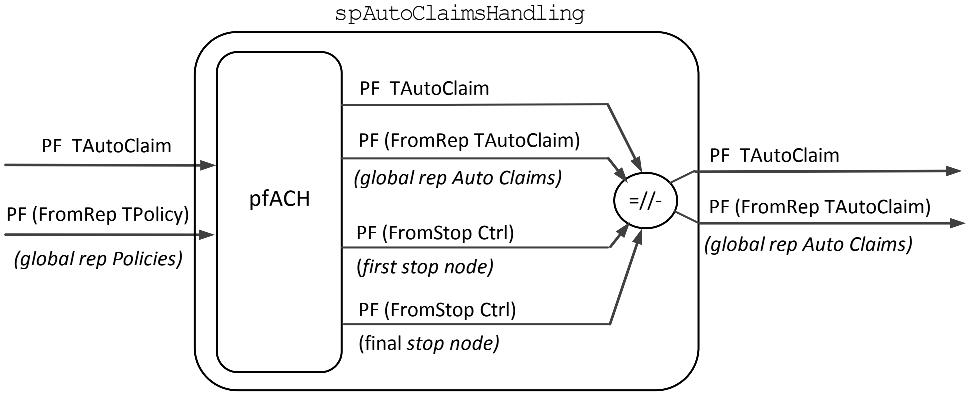


Figure 6.6: The data types flowing through sub-process `spAutoClaimsHandling`

Final Remark

From this case study, it should be noted that it is rather easy for a modeller to define business processes in our language. The different approaches that can be adopted to model the processes have been analysed with particular reference to the importance of connection patterns, to ensure the right abstraction for the construction of models which are more concise, readable and simple to reason about. From the discussed definitions, it should have been noted that the modeller does not require any knowledge of the host language Haskell and the underlying type system which is carrying out all the compile-time checks to ensure the appropriate construction of models. Moreover, parameterized models can easily be defined by the modeller. This technique has been discussed in sections 2.4.3.6 and 4.8. In fact, the parameterized model, `exclDec_withStopBranch`, defined in case study 4, can be used to define the decison in Figure 6.1. In the next case study, a more complex model shall be defined.

6.3. Case Study 2

The model which shall be considered for this case study has been obtained from one of the sample projects that come along with the tool. The selected project is named *ABC project*. This project contains business processes that handle customer orders within a company that sells some product ABC. For this reason, one of the business processes it contains is named '*Customer Order Handling*'. This process shall be used for this case study (see Figure 6.7).

The main aim of this case study is to identify how easy a complex model can be defined, with the least amount of effort, components and expertise, while still ensuring the correctness of the model. Connection patterns play a very important role to provide the required abstraction and modularity to handle such complex models, as shall be illustrated in this case study.

Defining Preliminary Components

As illustrated in the previous case study, before the modeller starts to construct the required process, the necessary new types (mainly business items), tasks and repositories should be defined in a similar way as discussed in case study 1 (and in the tutorial, Appendix A).

Although every activity in the process should have its input source and output target types defined, if the user opts to use the lazy serial composition connection pattern then he can use the activities without setting the connection types, and thus, he would not have to define typed task as `ttAssignNewAC = flowCI <| tAssignNewAC |> flowCO` in Listing 6.5. For the complete code and definition of the required business items, tasks and repository, see Appendix B.

Constructing the Model

Similar to the previous case study, the model shall be constructed using the basic modelling elements and using the connection patterns, such that the two can be compared. As in case study 1, the fragment that loads the content of the repositories shall be ignored; in our language this is carried out automatically as explained in the previous case study and in Section 4.2.4. However from this same fragment, it should be noted that repositories `'Products'` and `'Customer Orders'` are both global. Thus, their content should be passed on as an input argument. Moreover, in the process that shall be modelled (Figure 6.7), additional data is saved to repository `'Customer Orders'`. This means that a repository should be created to keep the previous content (passed on as an input argument), add the new data and later on pass it out as an output. In this way, other processes would have access to the updated data, in this global repository. On the other hand, repository `'Products'` is not updated with data in this process. Thus, the content of this repository can be handled as any other input argument, and at the end, there is no need for its content to be passed out as an output argument.

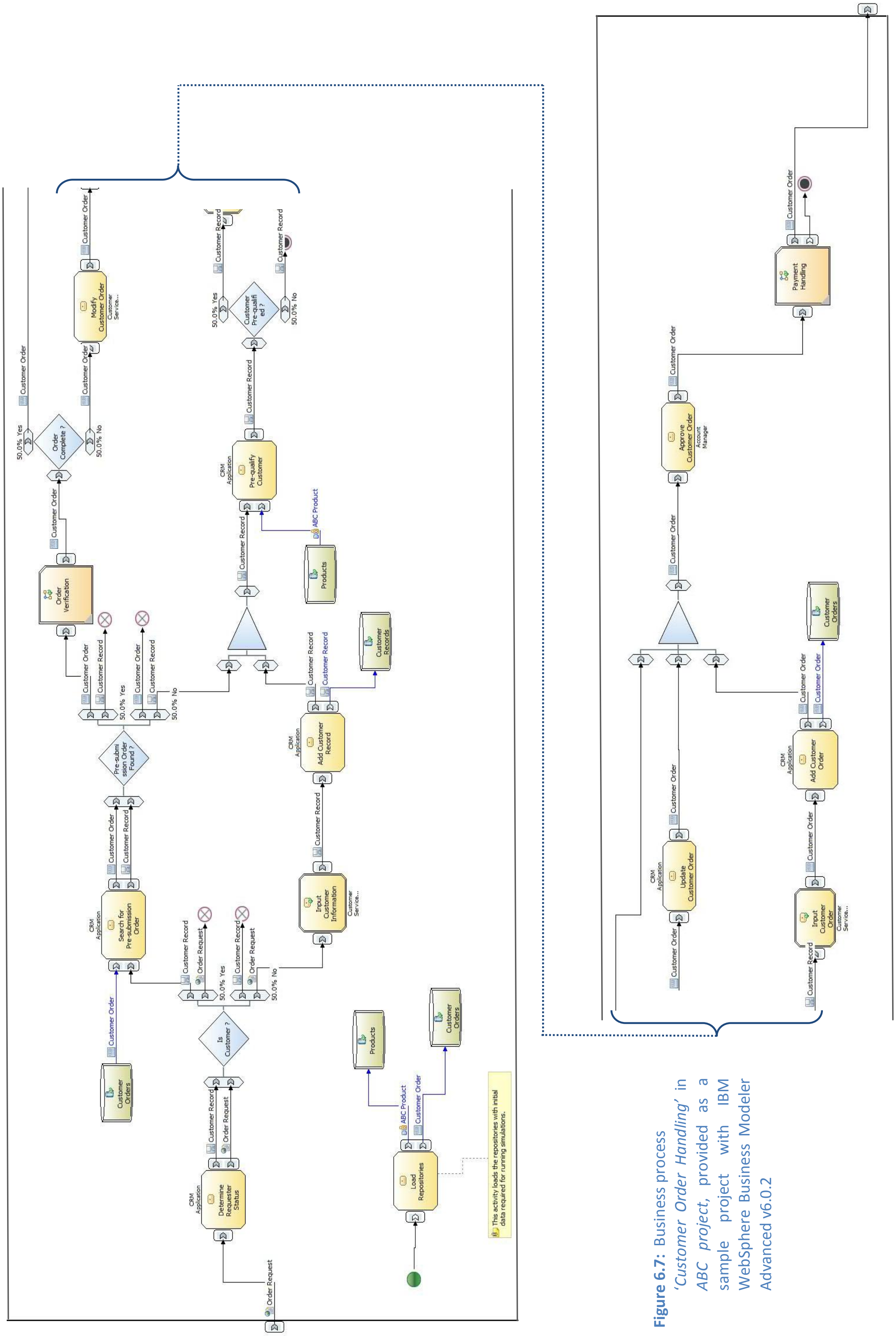


Figure 6.7

Figure 6.7: Business process *'Customer Order Handling'* in *ABC project*, provided as a sample project with IBM WebSphere Business Modeler Advanced v6.0.2

This process (Figure 6.7) also contains another repository named `'Customer Records'`. By looking at the properties of this repository in IBM's tool, one would realize that this is another global repository. Thus, once filled in and used within this process, its content should also be passed out as an output argument.

It is assumed that the sub-processes "`Order Verification`" and "`Payment Handling`" (respectively referenced as `spOrderVerification` and `spPaymentHandling` in the definition) are already defined and are accessible to this process. For the complete definitions of these sub-processes refer to Appendix B.

Construction 1: *Constructing the Model using only Basic Modelling Elements*

Since the model is constructed using the basic modelling elements (that is no connection patterns are used), hence it is assumed that the required typed tasks such as in Listing 6.5 are pre-defined together with any other basic components, such as tasks and repositories (i.e. `rCustOrder, rCustRec`). To be able to handle and reference the different elements in the process, the model is decomposed into logical fragments as illustrated in Figure 6.8. These fragments are later on defined and connected as shown in Listing 6.12.

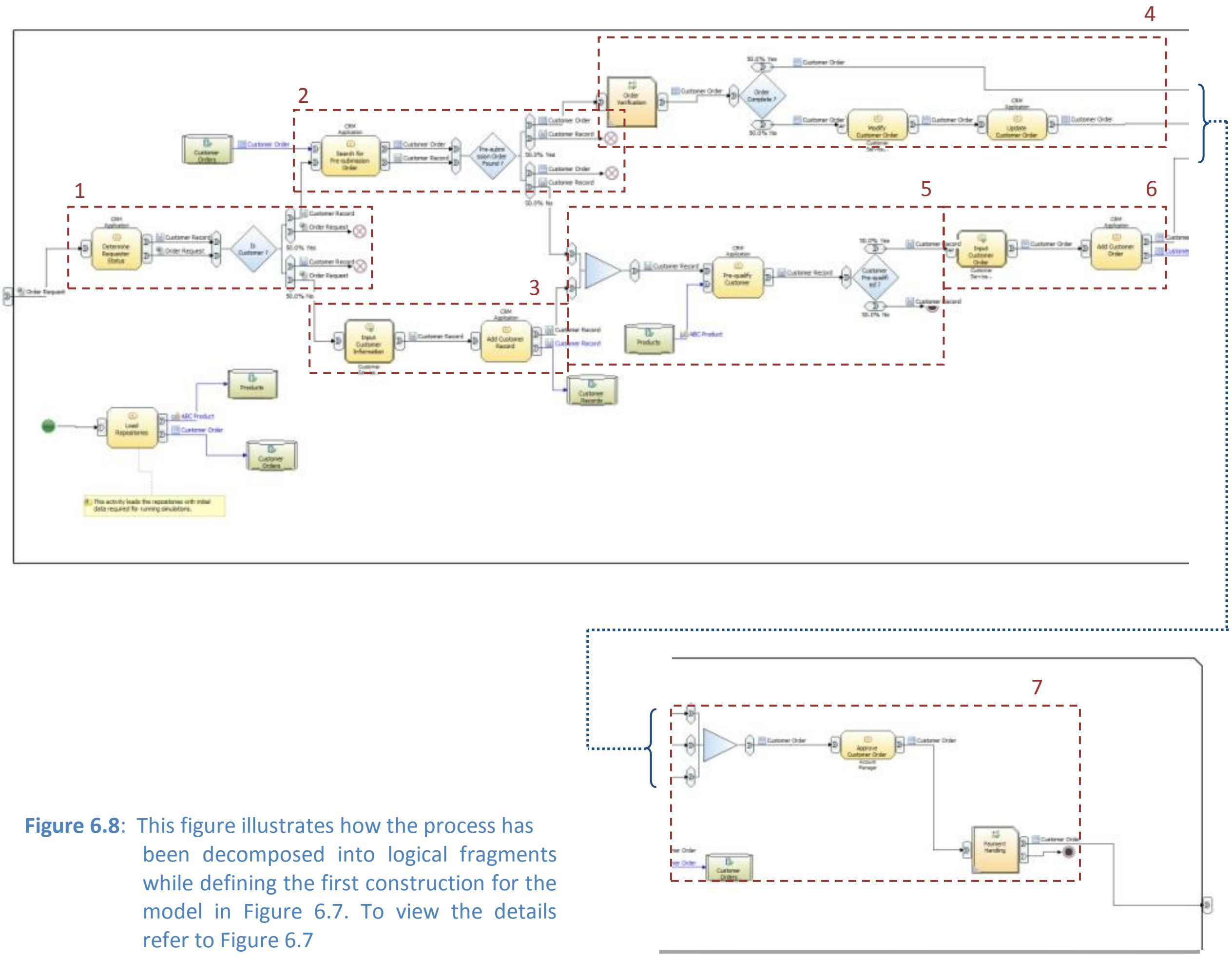


Figure 6.8: This figure illustrates how the process has been decomposed into logical fragments while defining the first construction for the model in Figure 6.7. To view the details refer to Figure 6.7

In our language this process is defined as illustrated in Listing 6.12a and 6.12b. Note that in the definition, the logical fragments depicted in Figure 6.8 are defined and handled individually starting off with the leftmost fragment. The rest are then handled from left to right, top to bottom. Listing 6.13 illustrates how the expressions assigned to every decision branch, should be defined for each of the decisions.

Listing 6.12a

```
pfCustOrderHandling (irProd, irCustOrder, x) = let

        -- Passing data to Repositories
        orCustOrder = rCustOrder1 (o2ttAddCustOrder, previousRepContent biTCustOrder irCustOrder)
        orCustRec   = rCustRec1   (o2ttAddCustRec)

        -- Logical Fragment 1
        ((oIsCust_CustRec,oIsCust_OrderReq),(oNotCust_CustRec,oNotCust_OrderReq)) =
                  exclDecision "Is Customer?" (branchProp eYes_IsCust 0.5, branchProp eNo_IsCust 0.5) (ttDetermineReqrStatus x)
        oend_IsCust_OrderReq   = end (oIsCust_OrderReq   :: PF (Flow TOrderReq))
        oend_NotCust_CustRec   = end (oNotCust_CustRec :: PF (Flow TCustRec))

        -- Logical Fragment 2
       ((oOrderFound_CustOrder,oOrderFound_CustRec),(oOrderNotFound_CustOrder,oOrderNotFound_CustRec))=
                  exclDecision "Pre-Submission Order Found?" (branchProp eYes_OrderFound 0.5,branchProp eNo_OrderFound 0.5)
                 (ttSearchPreSubmOrder (orCustOrder,oIsCust_CustRec))
        oend_OrderFound_CustRec      = end (oOrderFound_CustRec      :: PF (Flow TCustRec))
        oend_OrderNotFound_CustOrder = end (oOrderNotFound_CustOrder :: PF (Flow TCustOrder))

        -- Logical Fragment 3
       (o1ttAddCustRec, o2ttAddCustRec) = (ttAddCustRec . ttInpCustInfo) oNotCust_OrderReq

        -- Logical Fragment 4
       (oOrderComplete,oOrderNotComplete)=
                exclDecision "Order Complete?" (branchProp eYes_OrderComplete 0.5, branchProp eNo_OrderComplete 0.5)
                (tspOrderVerification oOrderFound_CustOrder)
        ottUpdateCustOrder = (ttUpdateCustOrder . ttModifyCustOrder)  oOrderNotComplete
```

Listing 6.12a: The first part of the required definition to model the process in Figure 6.7 using only basic modelling elements. The logical fragments referenced in the definition have been marked on the model in Figure 6.8

```
-- Logical Fragment 5
omerge_CustRec = merge (oOrderNotFound_CustRec :: PF (Flow TCustRec), o1ttAddCustRec)
oCustPreQual_CustRec,oCustNotPreQual_CustRec)=
            exclDecision "Customer Pre-Qualified?" (branchProp eYes_CustPreQual 0.5, branchProp eNo_CustPreQual 0.5)
           (ttPreQualifyCust (omerge_CustRec,irProd))
ostop_CustNotPreQual_CustRec  = stop (oCustNotPreQual_CustRec :: PF (Flow TCustRec))

-- Logical Fragment 6
(o1ttAddCustOrder, o2ttAddCustOrder) = (ttAddCustOrder . ttInpCustOrder) oCustPreQual_CustRec

-- Sub-Process Fragment 7
omerge_CustOrder = merge (oOrderComplete :: PF (Flow TCustOrder), ottUpdateCustOrder, o1ttAddCustOrder)
otspPaymentHandling = (tspPaymentHandling . ttApproveCustOrder) omerge_CustOrder
(opfCustOrderHandling, oCtrl) = (ctrl_dataFlowSplitter otspPaymentHandling) :: (PF (Flow TCustOrder), PF (Flow  Ctrl))
ostop_Ctrl = stop oCtrl

in

(opfCustOrderHandling, orCustOrder,orCustRec, oend_IsCust_OrderReq, oend_NotCust_CustRec, oend_OrderFound_CustRec,
oend_OrderNotFound_CustOrder, ostop_CustNotPreQual_CustRec, ostop_Ctrl)
```

Listing 6.12b: The second part of the required definition to model the process in Figure 6.7 using only basic modelling elements. The logical fragments referenced in the definition have been marked on the model in Figure 6.8

```
eYes_IsCust, eNo_IsCust :: (PF (Flow TCustRec), PF (Flow TOrderReq)) -> Bool
eYes_IsCust = const True
eNo_IsCust  = const True
```

Listing 6.13: Defining expressions which are assigned to every decision branch. This Listing defines the expressions for decision '`Is Customer?`' in Listing 6.12 and Figure 6.7

Lines 3 to 4 in Listing 6.12a, illustrate the data that should be saved in the repositories. Since the global repository `'Customer Orders'` contains the previous content to this repository, which would have been passed on as an input argument to the process (`irCustOrder`), the function `previousRepContent` is used, as shown in line 3 of Listing 6.12a. In this way, the user would be specifying and ensuring the source of that data; in this case, that it is truly coming from the repository `'Customer Orders'`. This function actually transforms data of type `PF (FromRep a)` to `PF (ToRep a)`. The content of the repositories `'Customer Orders'` and `'Customer Records'` are later produced as output of the process as indicated in Line 37, Listing 6.12b. This is not the case with repository `'Products'`. The process does not update the contents of this repository and thus, there is no need to return its content. Instead, this is simply used as any other input argument (as in line 27, Listing 6.12b).

As in case study 1, the type of the input flow to the end and stop nodes is specified as indicated in lines 8, 9, 14, 15, 28. Similarly, when the outgoing branch is directly connected to the incoming branch of another gateway, since both branches need to infer their type, the type of the flow should be specified as in Line 24 and 32. Such typing is also used in line 34 to indicate how the flow should be split. If the process is connected to some other fragment and these types can be inferred automatically, then such typing is not required. Once again, the output of all the end and stop nodes must be returned as output as indicated in Line 37 and 38. If not, the details of fragments connected to these nodes would not be accessible during model analysis. Using this construction approach, the user must ensure that these values are returned.

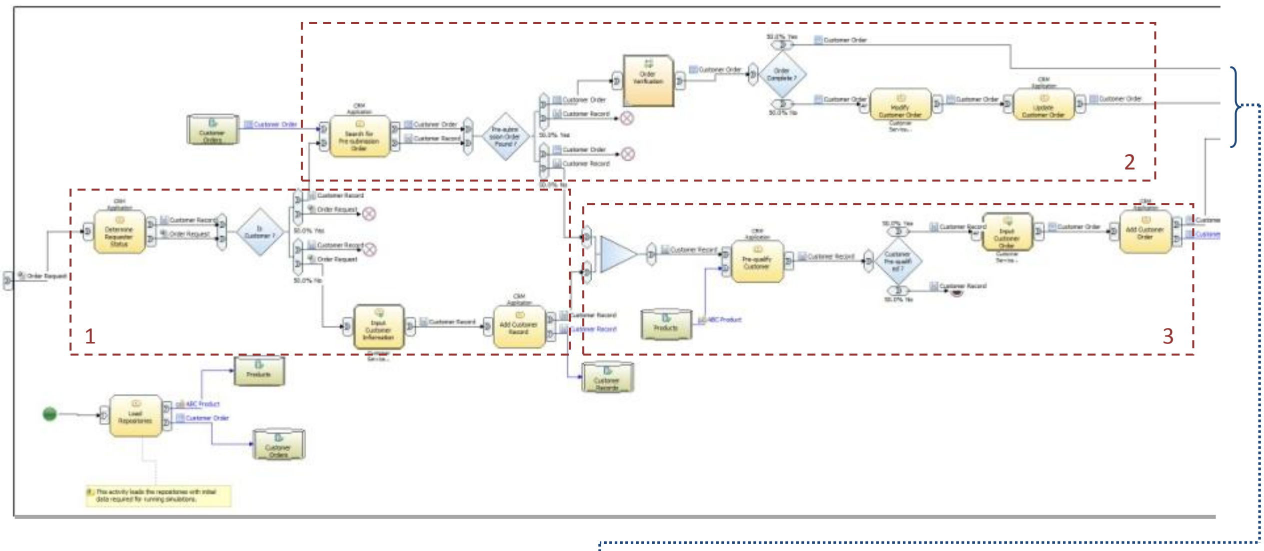
Definitions of processes defined using this approach, are not so concise. Moreover, the modeller should set an appropriate naming convention to name the arguments within the definition to try to make it more readable. It might also be helpful to decompose the process into a number of logical fragments as in Listing 6.12. The modeller might also decide to define these logical fragments as other processes to handle the model at the required level of abstraction. A notable disadvantage of this first construction is that the user needs to define the input source and output target types of all the activities that are used in the process. Thus, for the ideal abstraction and modularity, it is best to use the connection patterns provided with the language, in a similar way as discussed in case study 1 and as discussed in the next construction.

Construction 2: *Constructing the Model using Connection Patterns*

In this definition various connection patterns are used to provide the appropriate abstraction and modularity and to allow the system to handle certain implementation details automatically. Primarily, noting the advantages brought about by lazy serial compositon (->>-) in case study 1, this shall be used again for this definition. The model is once again decomposed into logical fragments such that a function, within the model definition, is defined, to carry out the required functionality. In cases where connection patterns cannot be used to connect these fragments, connections are carried out as in the first construction (which does not use connection patterns), such that the output arguments of one fragment are passed on as inputs to the other. Figure 6.9 illustrates how the model for this construction has been divided into logical fragments. The actual definition of the process is provided in Listing 6.14. The expressions assigned to every decision branch should be defined in a similar way as illustrated in Listing 6.13 with the only difference that, since the activities attached to the decisions do not have their output target type defined, then the input type of each expression must not define the connection type such that, rather than `(PF (Flow TCustRec), PF (Flow TOrderReq))`, the input type of the expressions in decision "`Is Customer?`" would be `(PF TCustRec, PF TOrderReq)`.

```
pfCustOrderHandling (irProd, irCustOrder, x) = let

       -- Passing data to Repositories
       orCustOrder = rCustOrder2 (o2tAddCustOrder, previousRepContent biTCustOrder irCustOrder)
       orCustRec   = rCustRec2   (o2tAddCustRec)

       -- Logical Fragment 1
       spf1 = exclDecision_with2AltDataBranches  tDetermineReqrStatus
                       ("Is Customer?", (branchProp eYes_IsCust 0.5, branchProp eNo_IsCust 0.5))
                       ((noActivity biTCustRec), (tInpCustInfo ->>- tAddCustRec))
       ospf1 @ ( (oIsCust_CustRec,oend_IsCust_OrderReq), (oend_NotCust_CustRec, (o1tAddCustRec, o2tAddCustRec)) ) = spf1 x

       -- Logical Fragment 2
       spf2  =  exclDecision_with2AltDataBranches  ((repCI, noConType) <| tSearchPreSubmOrder)
                       ("Pre-Submission Order Found?", (branchProp eYes_OrderFound 0.5, branchProp eNo_OrderFound 0.5))
                       (dec2UpperBranch, (noActivity biTCustRec))
        dec2UpperBranch = spOrderVerification -<|("Order Complete?", (branchProp eYes_OrderComplete 0.5, branchProp eNo_OrderComplete 0.5) )|>=
                       (noActivity biTCustOrder, (tModifyCustOrder ->>- tUpdateCustOrder))
        ospf2 @ ( ((oOrderComplete_CustOrder, tUpdateCustOrder),end_OrderFound_CustRec),(oend_OrderNotFound_CustOrder,oOrderNotFound_CustRec))
                      = spf2 (orCustOrder, oIsCust_CustRec)

        -- Logical Fragment 3
        spf3  = (merge -|- id) ->- ((flowCI, repCI) <| tPreQualifyCust)
                -<|("Customer Pre-Qualified?",(branchProp eYes_CustPreQual 0.5, branchProp eNo_CustPreQual 0.5))|>=
                ((tInpCustOrder ->>- tAddCustOrder), stopBranch biTCustRec)
        ospf3 @ ((o1tAddCustOrder, o2tAddCustOrder), ostop_CustNotPreQual_CustRec) = spf3 ((oOrderNotFound_CustRec, o1tAddCustRec), irProd)

        -- Logical Fragment 4
        spf4 = merge ->- tApproveCustOrder ->>- spPaymentHandling -//=. stop
        ospf4 @ (ospPaymentHandling, ostop_Ctrl) = spf4 (oOrderComplete_CustOrder, otUpdateCustOrder, o1tAddCustOrder)
  in
       (ospPaymentHandling, orCustOrder,orCustRec, oend_IsCust_OrderReq, oend_NotCust_CustRec, oend_OrderFound_CustRec,
        oend_OrderNotFound_CustOrder, ostop_CustNotPreQual_CustRec, ostop_Ctrl)
```

Listing 6.14: The required definition to model the process in Figure 6.7 using connection patterns. The logical fragments referenced in the definition have been marked on the model in Figure 6.9



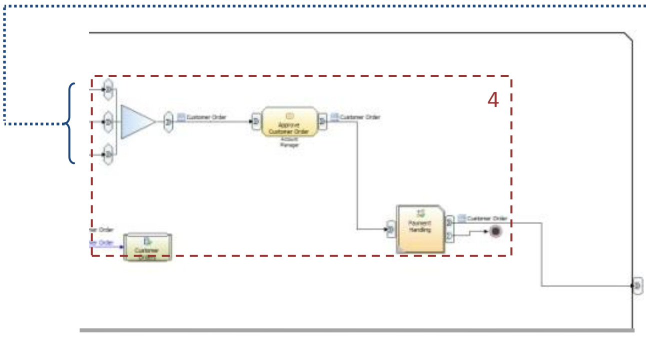


Figure 6.9: This figure illustrates how the process has been decomposed into logical fragments while defining the second construction for the model in Figure 6.7. To view the details refer to Figure 6.7

From Listing 6.14, it should be noted that the process fragment is decomposed into four logical fragments and for each one, a function is defined within the definition. These include: `spf1, spf2, spf3, spf4` (lines 6, 11, 19, 24). Since it is not real possible to combine these using serial or parallel composition, then given the appropriate inputs, the outputs are generated and passed on to appropriate fragments (lines 9, 16, 22, 25). In this way, these sub-process fragments can be handled individually and the user would have to handle less input and output arguments.

A variety of connection patterns, besides lazy serial composition (->>-) and others related to gateways, are utilized. With `exclDecision_with2AltDataBranches`(*exclusive decision with 2 alternative decision branches*) we were able to terminate unnecessary data flows on decision branches. This was used for decisions '`Is Customer?`' and '`Pre-submission Order Found`?'(lines 6 and 11). The first input of this function is the process fragment which is attached to the input of the decision, followed by details about the decision and the process fragments which are attach to its outgoing branches. The user in this way would not have to define the end nodes that terminate the unnecessary data flows. Another connection pattern which is used is `noActivity`, for instance, `noActivity biTCustRec` in line 8. This is used as a dummy component to allow a specifically typed data flow to pass through. This in fact used instead of `:: PF TCustRec` and is especially helpful for users who are not familiar with Haskell.

Final Remark

This case study focuses on the role and importance of connection patterns to define complex models. Comparing the first construction in Listing 6.12 with the second in Listing 6.14, it should be noted that besides being more concise, when connection patterns are used, definitions are more readable and easier to comprehend. Moreover, less input and output arguments have to be handled manually by the modeller. Connection patterns can be used at various levels of abstraction, some of which are able to hide the implementation of entire fragments such as `exclDecision_with2AltDataBranches` (*exclusive decision with 2 alternative decision branches*) and others, which shall be discussed in the next case study.

6.4. Case Study 3

The model which shall be considered in this case study, has been intentionally constructed to illustrate the importance of connection patterns within our language, to handle some of the most commonly modelled fragments and other fragments, which can easily introduce new errors, if constructed manually. Figure 6.10 illustrates the process that is used for this case study.

In Figure 6.10, three main fragments and a final task can be identified, as illustrated in Figure 6.11. These can be handled individually and later on connected in sequence.

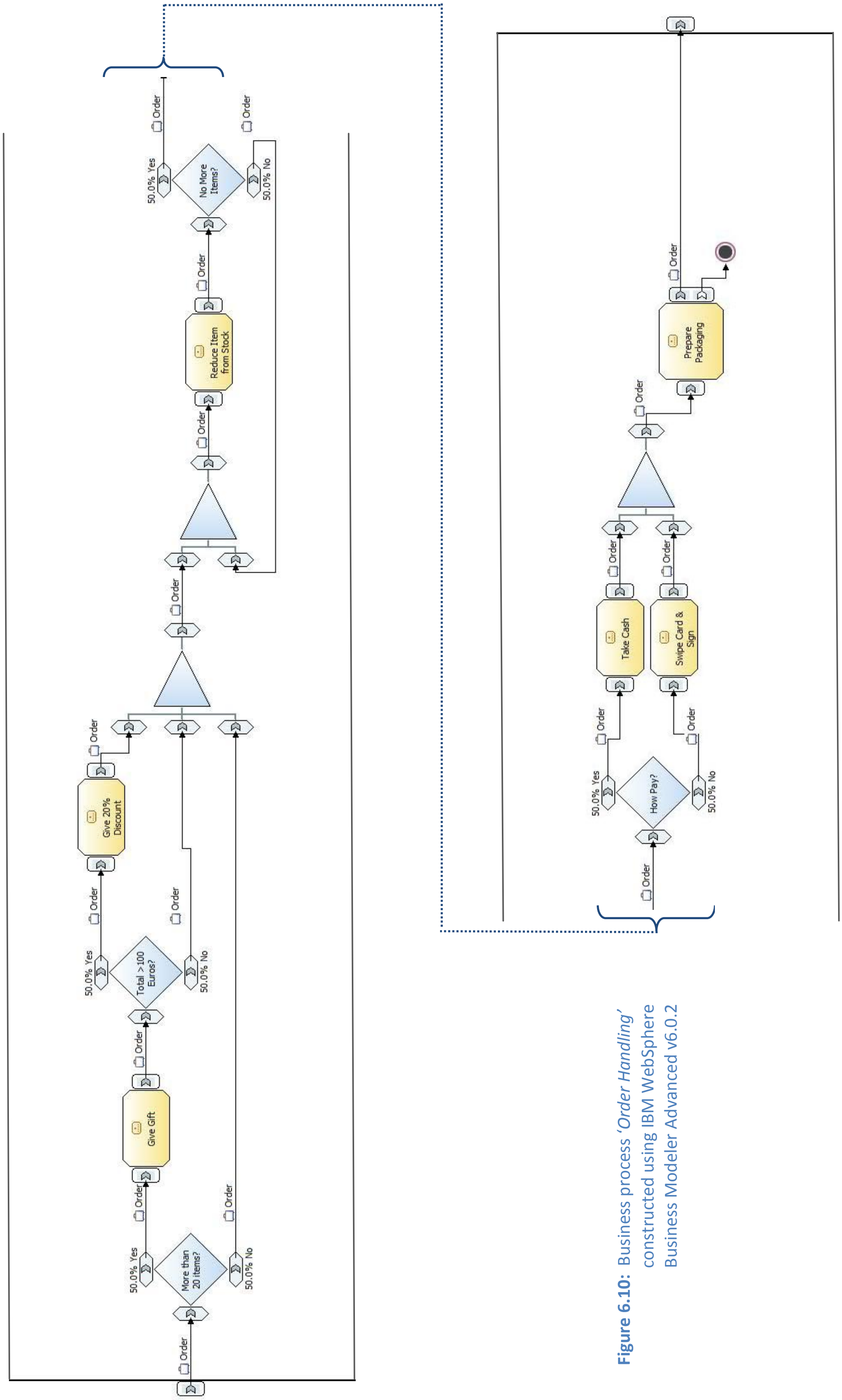


Figure 6.10: Business process *'Order Handling'* constructed using IBM WebSphere Business Modeler Advanced v6.0.2

Figure 6.10

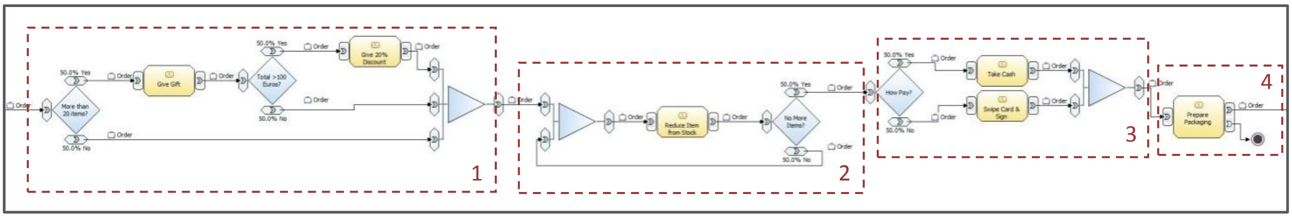


Figure 6. 11: The process in Figure 6.10 is decomposed into sub- fragments. To view the details of the process fragment, refer to Figure 6.10

Assuming that all the required activities and business items are already defined, the modeller can start off constructing the model. As illustrated in Figure 6.11, the process in Figure 6.10 can be decomposed into sub-fragments such that the first block is a *triangular-shaped model* for which the connection pattern `tri_exclDecisions_onUpperBranch_Merge` can be used, the second is a sound *merge - exclusive decision cycle* which can be constructed by using the connection pattern `soundCycle`, and the third fragment is an *inclusive decision followed by a merge*, which can be constructed in a sound manner (not to introduce lack of synchronization) by explicitly using exclusive decisions and forks, in which case the connection pattern `sound_inclDecision_Merge` should be used. The latter decision in the third fragment is an inclusive decision; the customer might decide to pay part by cash and part by credit card, in which case, the fragment would lack synchronization. Finally, the process ends with a task and a stop node. (For more details about these connection patterns, refer to the tutorial, Appendix A)

Construction 1: *Constructing the Model using only Basic Modelling Elements*

If a sound model had to be defined for this process fragment without using connection patterns, then the code in Listing 6.15 should be used. In this listing, it is assumed that the expressions assigned to each decision branch are pre-defined.

Construction 2: *Constructing the Model using Connection Patterns*

If this process fragment had to be defined using the provided connection patterns then the code in Listing 6.16 would be enough to construct the model in a sound manner.

As noted in Listing 6.16, by using the appropriate connection patterns, the definition of the model is much more concise than that in Listing 6.15. Moreover, the implementation details as how to construct a sound cycle and how to construct a sound inclusive decision - merge fragment, are all abstracted away from the user. Thus the user can focus solely on the required behaviour rather than how models should be implemented and how soundness should be guaranteed. In this way, the produced models would not have deadlocks and would not lack synchronisation. If connection patterns were not used and the cycle was not modelled correctly, for instance a join was used instead of a merge, then a deadlock would be introduced. Similarly, if the user had to model the inclusive decision merge block using an ordinary inclusive decision, then the fragment would lack synchronization. Likewise, although it is convenient (makes models less cluttered) to close more than one decision with one merge, if the user by mistake closes also a fork with a merge, then the model would become unsound.

Moreover, the initial untyped tasks (i.e. without the input source and output target types specified) are used and there is no need to explicitly type input and output arguments (as lines 3, 9, 11, 17, 20, 24, 27 in Listing 6.15). Since connection patterns are used, all the types in Listing 6.16 are automatically inferred. Once defined, the sub-fragments are connected together using lazy serial composition (line 13, Listing 6.16). In this way, irrespective whether typed tasks are used, the models are composed using the same operator.

Listing 6.15

```
pfOrderHandling x     =  let

        -- Sub-Fragment 1 - Triangular Model
        (oMoreThan20Items,oLessThan20Items) = exclDecision ">20 Items?" (branchProp eYes_Flow 0.5, branchProp eNo_Flow 0.5)
                                                                    (x :: PF (Flow TOrder))
        ottGiveGift = ttGiveGift oMoreThan20Items
        (oMoreThan500Euros,oLessThan500Euros) = exclDecision "Total >500Euros?" (branchProp eYes_Flow 0.5, branchProp eNo_Flow 0.5)
                                                                      ottGiveGift
        ottGiveDiscount = ttGiveDiscount oMoreThan500Euros
        omerge1 = merge (ottGiveDiscount, oLessThan500Euros :: PF (Flow TOrder), oLessThan20Items :: PF (Flow TOrder))

        -- Sub-Fragment 2 - Sound Cycle
        omerge2 = merge (omerge1 :: PF (Flow TOrder), oMoreItems :: PF (Flow TOrder))
        ottReduceItemFromStock = ttReduceItemFromStock omerge2
        (oNoMoreItems,oMoreItems) = exclDecision "No More Items?" (branchProp eYes_Flow 0.5, branchProp eNo_Flow 0.5) ottReduceItemFromStock

        -- Sub-Process Fragment 3 - Sound Inclusive Decision Merge Fragment
        (oCash,oCredit,oCashCredit)= exclDecision "How Pay?"
                                                        (branchProp eCash_Flow 0.4, branchProp eCredit_Flow 0.5, branchProp eCashCredit_Flow 0.1)
                                                        (oNoMoreItems :: PF (Flow TOrder))
        ottTakeCash = ttTakeCash oCash
        ottSwipeCardSign = ttSwipeCardSign oCredit
        (o1fork,o2fork) = fork (oCashCredit :: PF (Flow TOrder))
        ottTakeCash2    = ttTakeCash o1fork
        ottSwipeCardSign2 = ttSwipeCardSign o2fork
        ojoin = join (ottTakeCash2, ottSwipeCardSign2)
        omerge3 = merge (ottTakeCash, ottSwipeCardSign, ojoin :: PF (Flow TOrder))

        -- Sub-Fragment 4 - Final task and stop node
        ottPreparePackaging = ttPreparePackaging omerge3
        (opfOrderHandling, oCtrl) = (ctrl_dataFlowSplitter ottPreparePackaging) :: (PF (Flow TOrder), PF (Flow  Ctrl))
        ostop_ctrl = stop oCtrl

    in  (opfOrderHandling, ostop_ctrl)
```

Listing 6.15: The required definition to model the process in Figure 6.10 using only basic modelling elements
The logical fragments referenced in the definition have been marked on the model in Figure 6.11

Listing 6.16

```
pfOrderHandling x  =  let

         -- Sub-Fragment 1 - Triangular Model
         pfOrderHandling2=let

         -- Sub-Process Fragment 1 - Triangular Model
         spf1 = tri_exclDecisions_onUpperBranch_Merge ["&gt;20 Items?", "Total &gt;500 Euros?"]
                                                      [(branchProp eYes 0.5, branchProp eNo 0.5),(branchProp eYes 0.5, branchProp eNo 0.5)]
                                                      [(tGiveGift, noActivity biTOrder), (tGiveDiscount, noActivity biTOrder)]

         -- Sub-Process Fragment 2 - Sound Cycle
         spf2 = soundCycle tReduceItemFromStock ("No More Items?",(branchProp eYes 0.5, branchProp eNo 0.5))

         -- Sub-Process Fragment 3 - Sound Inclusive Decision Merge
         spf3 = sound_inclDecision_Merge "How Pay?" (branchProp eCash 0.5, branchProp eCredit 0.5) (tTakeCash,tSwipeCardSign)

   in
         spf1 ->>- spf2 ->>- spf3 ->>- tPreparePackaging -//=. stop
```

Listing 6.16: The required definition to model the process in Figure 6.10 using connection patterns
The logical fragments referenced in the definition have been marked on the model in Figure 6.11

Final Remark

In this case study, it should be evident how convenient and beneficial such highly abstraction connection patterns are, to ensure the production of sound good quality models in the least amount of time, effort and expertise. In this way, modellers are allowed to focus solely on the required behaviour.

6.5. Case Study 4

The importance and the possibility for users to define their own parameterized models have been discussed in greater depth in sections 2.4.3.6 and 4.8. In this case study, two such models are defined, one of which could have been used for the model defined in case study1.

Parameterized Model 1: A decision with a terminating branch

Considering the process fragment in Figure 6.12, the definition in Listing 6.17 would be sufficient to express this fragment.

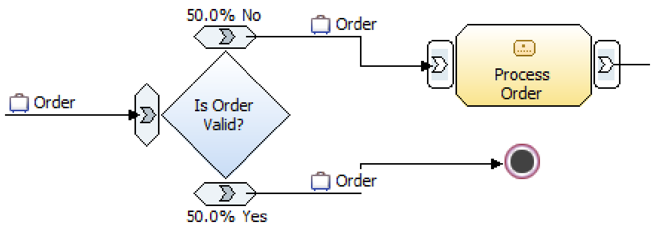


Figure 6.12: A process fragment handling an order

```
pfOrder = (exclDecision "Is Order Valid? "(branchProp eYes 0.5, branchProp eNo 0.5))
          ->- (tProcessOrder -|- stopBranch biTOrder)
```

Listing 6.17: Defining in our language the process fragment in Figure 6.12

If the a parameterized model such as that in Listing 6.18, is defined, then the fragment in Figure 6.12 would be defined as illustrated in Listing 6.19.

```
exclDec_withLowerStop nm brs typ pf  = (exclDecision nm brs)->-(pf-|-stopBranch typ)
```

Listing 6.18: A parametrized model to define models similar to Figure 6.12
`nm` is the name of the decision; `brs` is the tuple defining the properties of the decision branches; `typ` is the type of the data flow; `pf` is the process fragment attached to the upper branch

```
pfOrder = exclDec_withStopBranch  "Is Order Valid?"
                                  (branchProp eYes 0.5, branchProp eNo 0.5)
                                   biTOrder
                                   tProcessOrder
```

Listing 6.19: Defining the process fragment in Figure 6.12 by using the parameterized model in Listing 6.18

In this way, fragments, such as Figure 6.13, which have a similar structure, can rapidly be defined by simply using one function (as in Listing 6.20).

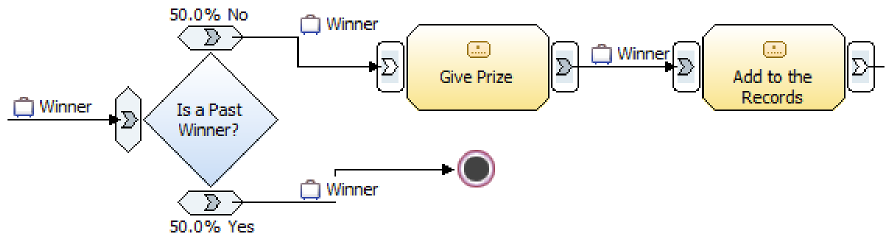


Figure 6.13: A process fragment handling decide to how prizes are awared

```
pfWinners = exclDec_withStopBranch "Is a Past Winner? "
                                    (branchProp eNo 0.5, branchProp eYes 0.5)
                                    biTWinner
                                    (tGivePrize ->- tAddToRecords)
```

Listing 6.20: Defining the process fragment in Figure 6.13 by using the parameterized model in Listing 6.18

Thus, this model can easily be used to define decision `'Valid Auto Claim?'` in *Auto Claims Handling* process, discussed in case study 1.

Parameterized Model 2: *A decision-merge fragment with multiple inner fork-joins*

It is possible to define parameterized models that handle more complex structures such as Figure 6.14. Listing 6.21 illustrates how such a parameterized model should be implemented.

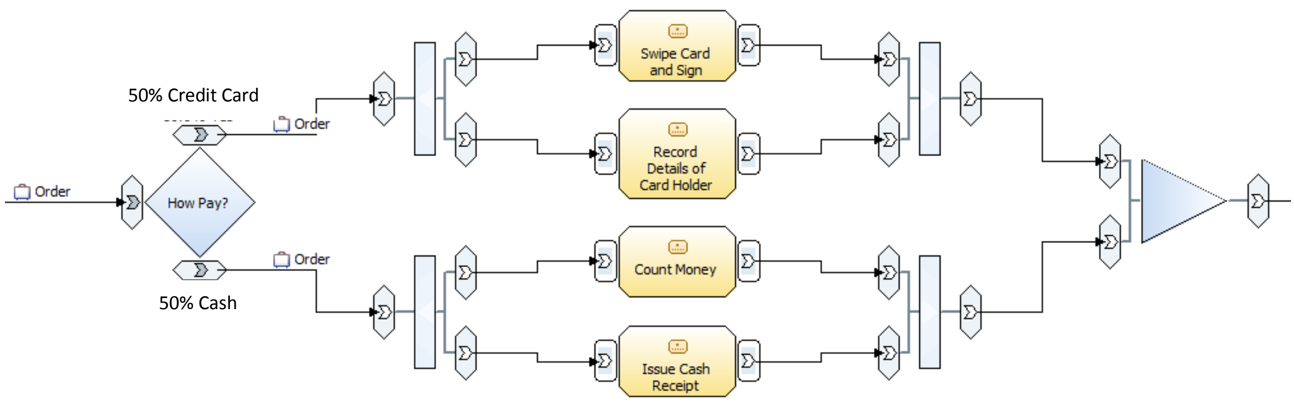


Figure 6.14: A decision - merge fragment with internal fork -join fragments

```
fork_joins  [pfsFJ]          =  fork_join  pfsFJ
fork_joins  (pfsFJ : pfsFJs)  =  (fork_join pfsFJ)  -|-  (fork_joins  pfsFJs)

exclDecisionMerge_forkJoins nm brs pfsL =
                                        exclDecision_merge nm brs (fork_joins pfsL)
```

Listing 6.21: Defining the parameterized model `exclDecisionMerge_forkJoins` to define fragments as in Figure 6.14

Since ideally the user of such a parameterized model should be able to pass on the fragments of the internal fork-joins as a list of tuples (each tuple contains the internal fragments for a particular fork join), then a function which is essentially another parameterized model, `fork_joins`, is defined (Listing 6.21). The role of this function is to construct the fork join fragments and compose them in parallel. To do this, the connection pattern `fork_join`, defined in our language, is used. Once constructed, this is passed on to the built-in connection pattern `exclDecision_merge`, which generates the exclusive decision merge fragment. With such a parameterized model, the fragment in Figure 6.14 can be defined as illustrated in Listing 6.22.

```
pf = exclDecisionMerge_forkJoins "How Pay?"
                                 (branchProp eCreditCard 0.5, branchProp eCash 0.5)
                                 [(tSwipeCardSign,tRecordDetailsCardHolder),
                                  (tCountMoney, tIssueCardReceipt)]
```

Listing 6.22: Defining the model in Figure 6.14 using `exclDecisionMerge_forkJoins`

Final Remark

In this case study, the benefits of a functional modelling language are evident. Using IBM's tool, users are only allow to record a sequence of operations. With our language, users are free to identify their own commonly used fragments and easily implement them as parameterized models, such that given specific input, the required model is constructed. These also provide the ideal abstraction and modularity that such a modelling tool should provide.

6.6. Final Remarks on Case Studies

The above four case studies illustrate that using our language, any model which is usually defined using IBM WebSphere Business Modeler, can easily be constructed by non-IT specialists in a concise, readable and easy to comprehend manner. Although business processes in our language are defined textually rather than graphically, it is still straightforward for the reader to visualize and understand the models.

Using a functional approach, users are allowed to focus on the definition of the required behaviour, rather than the implementation of such behaviour. Models can easily be decomposed into process fragments with a specific functionality. These can be defined in the form of functions and later on reused and composed in a variety of ways to obtain the required complex behaviour. Since functions are pure, irrespective of the context where they are used, given specific inputs, the same outputs are always generated. In fact, tasks can easily be reused within any processes with different input source and output target types (as illustrated and discussed in case study 1).

Even though all the modelling elements are essentially functions, the user does not require any knowledge of functional programming or Haskell. Using connection patterns, the user does not need to explicitly handle the input and output arguments of these functions. Instead, he can focus on how these should be connected to construct the required model and thus handle them as normal modelling elements in IBM's tool. One of the main differences is that connectors in our language are not considered as modelling components. The modeller does not need to define connectors. Instead, connections between fragments are induced as soon as the output of one fragment is passed on as an input to another, or when fragments are connected together using some connection patterns, such as serial composition (->-). In this way, a fragment can be used only if all of its inputs are defined. This means that it is not possible to have dangling inputs. This is important since, as defined in scenario 3, part 2 of (Koehler & Vanhatalo, 2007), dangling inputs lead to deadlocks. On the other hand, to avoid dangling outputs, as illustrated in the case studies, connection patterns should be used where possible such that all the outputs are automatically induced, without explicitly having to define the output arguments (as in case study 1, Listing 5.9).

In fact, similar to hardware description languages, connection patterns play an important role to ensure the proper abstraction and modularity when defining models (Sheeran, 2005). Besides connecting process fragments to create more complex ones, connection patterns are also important for other purposes.

Case study 1 illustrated how typed tasks can be defined with a single line of code by using the connection pattern `<|...|>`; for instance, `flowCI <| task1 |> flowCO`, to specify that the input source and output target types of `task1` are defined as flows. Even though `flowCI` and `flowCO` are functions, using them in this manner, for a modeller who is familiar with IBM's tool, these can be considered as simple arguments which can be used to specify the properties of modelling elements. The modeller does not really need to know anything about our embedded type system or typing in Haskell. In fact, in cases where the modeller needs to specify the type of the data flowing between modelling elements, the provided function `noActivity` can be used. This can serve as a dummy component to allow a data flow of a specific type to pass through without being modified. From this, the types of other elements can be inferred (example, case study 2, Listing 6.14, Line 8).

Another important connection pattern that was analysed in case study 1 is lazy serial composition. This reduces the amount of code that is required for a model to be defined, such that the user does not need to specify the input source and output target types of activities. In this way, the connection type is automatically inferred depending upon the fragments connected to it. This would also eliminate errors that can possibly be introduced when connection types are defined by the user.

Other highly modular connection patterns that provide the appropriate abstraction for the user to create sound models which are type safe with the least amount of effort and expertise are introduced in case studies 2 and 3. The advantage of patterns such as `soundCycle` to produce sound structured cycles, `tri_exclDecisions_onUpperBranch_Merge` to produce triangular models and `sound_inclDecision_Merge` to produce sound inclusive decision - merge fragments, is evident in the defined models. The need for such patterns was identified from (Koehler & Vanhatalo, 2007).

The difference between models defined using and not using such connection patterns is notable in all the case studies. The more concise the definitions, the easier to comprehend and to reason about and the less the errors within the model. The operators used for the connection patterns were chosen carefully such that, even though different patterns are provided, by looking at the operator, its semantics would immediately be recognized. Although most of the checks are carried out at compile-time during the construction of the model, the user does not really need to know anything about Haskell's type system.

Similar to connection patterns, with our modelling language, parameterized models can be defined. While connection patterns are usually provided by the language, parameterized models can easily be defined by the users themselves, such that, if fragments having a particular structure, be it iterative, branching or sequential, are often used to model processes within an organisation, then such a parameterized model would be helpful to abstract away the implementations of such models. Depending upon the defined parameters, the required model would be constructed by simply evaluating one function. This feature is currently not available or not fully supported in modelling tools such as IBM WebSphere Business Modeler.

The processes within the three sample projects provided with the IBM's tool (including these case studies) have been defined using our language and are provided as Appendix B. Every model is defined using different approaches. In this way, the reader can compare the definitions and identify the differences between them.

6.7. Other Evaluation Techniques

In the previous four case studies, evaluation was based on the appropriateness of our language to define models which are usually created in IBM WebSphere Business Modeler. Although this is important, for a more comprehensive evaluation of our language, other evaluation techniques should be employed. Due to the amount of resources, time, effort and domain users that are required, it was not really possible to carry out such evaluation for this first prototype.

Primarily, it would be interesting to identify how long users take to learn our language and to start defining models. Their feedback, comments and first impressions are important to further enhance the language and adapt it to the users' requirements and expectations. Following this, modellers should be allowed to use our language and IBM's tool for a couple of months. Evaluation should then be carried out to check how many models were defined using our language and how many were created using IBM's tool. In this way, it would be possible to identify the modellers' preferences. A study similar to (Koehler & Vanhatalo, 2007) should be carried out. In (Koehler & Vanhatalo, 2007), hundreds of models that were produced between 2004 and 2006 by various modellers were analysed. Thus, it would be interesting to analyse an approximately equal amount of models and try to extract some anti-patterns and bad design practises. These should then be compared to those identified in (Koehler & Vanhatalo, 2007), to check whether modellers who are using our language, are still producing the same mistakes and errors identified in this article. In this way, it would be possible to analyse the effectiveness of our language to ensure the production of good quality models with the least amount of effort and user-knowhow. If it is noted that there are specific errors which are being carried out when models are defined with our language then addition checks, basic transformations or features can be added to our language. Besides analysing the types and amount of errors produced when using our language, it is also interesting to compare the time it takes for a modeller to create a good model with our language and with IBM's tool.

Such techniques would ensure a comprehensible evaluation of our language. However, various domain experts need to be employed and various models would have to be analysed. This would also help us to evolve our language and add the required features to meet the user's demands and expectations.

6.8. Conclusion

Although four case studies have been discussed in this chapter, various other models have been defined in our language and provided as Appendix B. After analysing these examples, it should be evident that using our language any business process model can be constructed in a concise and readable manner. This is possible through the use of connection patterns and parameterized models. Moreover, the produced models are guaranteed to be of a high quality. Through our embedded type system, errors are identified as early as construction time, when the script defining the model is compiled. This ensures that errors are trapped at the modelling phase and are not allowed to propagate to the succeeding stages.

Chapter 7

Conclusions

7.1. Introduction

With our functional modelling language, we have managed to develop a language which is able to capture precisely the domain of business process modelling and allow users (who might not necessarily have any knowledge of Haskell), to model, transform and quality assure business processes in Business-Driven Development (BDD). The defined models are readable, easy to comprehend and most importantly type-safe. By defining and using the provided quality assurance checks, the soundness of the processes is guaranteed and thus the derived IT solutions should be correct. Since our language has been successfully embedded in Haskell, we were able to adopt a functional approach and inherit the infrastructure (such as the type system), tools (such as debuggers and compilers) and features (such as type classes, higher-order functions) of the language without necessarily having to re-implement them. For this reason, we were able to focus more on the semantics of our domain.

7.2. Achievements

We managed to achieve all our objectives and goals. Primarily, by embedding the domain specific language, we were able to capture precisely the semantics of the Business Process Modelling domain, such that models are rapidly defined in a concise and abstract manner. The definitions are readable, easy to comprehend and reason about. Connection patterns play an important role to help in the construction of readable definitions, as illustrated in the case studies analysed in Chapter 6 and Appendix B.

By embedding the language in Haskell, the models, quality assurance checks and transformations are essentially functions which can easily be composed and defined. Since it is based on higher-order logic, the user does not need to take into consideration the implementation of the computations. Instead, the users are allowed to solely focus on the required behaviour. Different from the previous modelling tools, users are able to define their own parameterized models and transformations. By generating a directed graph for the models, various types of analysis can be carried out with greater ease. Moreover, quality assurance can be combined to model transformations by declaratively defining pre and post conditions for each transformation. These conditions as well as transformations can easily be composed of other previously defined checks or transformations.

Various features of Haskell resulted to be essential for the implementation of our language, one of which is Haskell's type system. By embedding our own type system in that of Haskell, we are able to carry out compile-time checks to ensure the construction of type-safe models. Thus, certain errors, such as the composition of incompatibly typed modelling elements (i.e. when the output types of the first element are different from the input types of the second element), are trapped as early as compile-time. This prevents users from carrying out other operations on ill-typed models and thus, prevents the propagation of such errors to the next phases of the Business-Driven Development life-cycle. In this way, with our language we are able to quality assure models by carrying out three types of checks: 1) by Haskell's type checker, 2) at construction-time through our embedded type system and 3) by specialised functions that analyse the components in the model to for instance, check the soundness of the model.

In this way, we have managed to provide a functional modelling language with which good quality models can be defined by business analysts in a rapid and concise manner and which can later on be analysed and interpreted in various ways. Thus we have managed to capture the domain semantics of IBM's WebSphere Business Modeler Advanced v6.0.2[1]. Parts of the models are also exportable to IBM's modelling tool.

7.3. Future Work and Enhancements

The language is very flexible and can easily be extended with additional functionality. Since a deep embedded approach has been adopted for our language, the defined models can be interpreted and analysed in an infinite variety of ways. Different extensions require different types of expertise and can be carried out at different levels of abstraction.

If new modelling elements need to be added to the language, a Haskell programmer would have to modify the internal data type constructor `PrimPF` and the other corresponding functions. If on the other hand, new built-in types need to be defined, then the same type of declaration used to define user-defined complex types, would have to be defined within the language's `MainTypes.hs` module. For new interpreters or primitive transformations to be constructed, the programmer must know the name of all the modelling elements which would have to be pattern matched and handled accordingly.

To capture the full semantics of the IBM's WebSphere Business Modeler, other modelling elements such as notifications and swimlanes which are available in IBM's tool, can be added to the language. Additional attributes can be added to the modelling elements, such as the time and the resources required to carry out the specific activities. If, during a comprehensive evaluation of the language (as discussed in Section 6.7), it is noted that users are carrying out specific errors, additional compile-time checks and functions to detect such errors, can be added. Similar to any other language, it is important to consider user's feedback and suggestions and try to evolve the language accordingly to ensure that user's requirements and expectations are met.

Although initially it was thought that additional functionality should be provided to simulate the process, this is not really necessary, because once the model is transferred to the modelling tool, IBM's simulation functionality can easily be used. It would be more interesting to provide interpreters that define the model in other notations such as Business Process Modelling Notation (BPMN) (OMG, 2008). In this way, our language (at least the basic modelling elements) would be applicable for any other tool. A BPMN engine can then be used to automatically generate the executable BPEL code. Using our language the defined transformations can be interpreted into the code provided in IBM's transformation framework (Koehler, et al., 2007). In this way, composite transformations would be created using our language and then integrated as part of IBM's modelling tool.

Various types of analysis can be carried out on the models. For instance, similar to the hardware description language Lava (Claessen, 2001) , the model can be passed on to a model checker for complete state analysis and to ensure the production of sound models which are free from deadlocks and lack of synchronisation. To mitigate the state space explosion of model checkers, the quality assurance techniques

[1] http://www.ibm.com/developerworks/websphere/zones/businessintegration/roadmaps/modeler/roadmap_advanced.html

which are integrated as part of IBM's transformation framework (Koehler, et al., 2007) can be used to decompose the process into Single-Entry Single-Exit (SESE) fragments (Vanhatalo, Völzer, & Leymann, 2007) and model check every single fragment individually. In this way, different from the linear-time control-flow heuristics in (Vanhatalo, Völzer, & Leymann, 2007) and the anti-patterns in (Koehler & Vanhatalo, 2007), the soundness of any type of model can be verified. It would also be interesting to carry out parameterized verification on both the models and the transformations.

Definitions in our language, define the structure of models. Although the functional higher-order logic approach helps users to focus on the required behaviour rather than the implementation details, it might be beneficial for users (especially new modellers) to abstract away from the semantics of the modelling elements and describe the behaviour rather than the structure of the models, in a manner which is more suitable for their organisation. A similar approach was adopted in (Claessen & Pace, 2002), whereby new languages embedded in Lava can be defined. Such languages would have their own abstract data type with the required constructs and the corresponding combinators with which the behaviour rather than the structure of the circuits would be defined. An interpreter is then developed so that from a behavioural description, the structural description in terms of Lava's combinators would be derived. In a similar manner, it would be possible for each organisation to have its own language embedded in our modelling language, to define the required domain behaviour. The structural model would then be derived from these definitions and thus the provided functionality in our language would still be accessible.

One of the most important features of our language is that most of the checks are carried out at compile-time such that errors are trapped as early as construction time. However, since our language is embedded in the functional language Haskell and since the error messages generated by the compiler are not really specific to our domain language, hence it would be essential to create a user-friendly graphical user environment for our language. This can possibly include a text editor with which models can be defined. In this way, error messages returned by Haskell's compiler would be trapped and handled accordingly, such that user-friendly messages, which are specific to our domain, can be returned to the user. These messages would help the user to better identify the errors and correct the models.

Thus, since the models defined in our language can be interpreted in various ways, it would be important to define the denotational semantics of the language to ensure that the language constructs are always interpreted as originally intended.

7.4. Final Remarks

By choosing an appropriate host in which to embed our language, we have managed to capture precisely the semantics of the business process modelling domain, without necessarily implementing the domain independent infrastructure of the language. Instead, we were able to focus on the domain dependent components and the issues encountered while embedding the language. Using a combinatorial approach, models can easily be constructed and later on analysed. In this way, we managed to develop a functional modelling language which provides the ideal abstraction for business modellers to construct good quality models from which the underlying IT solution can be derived in the Business-Driven Development lifecycle. The models can also be transformed and quality assured for soundness.

References

Arthur, L. J. (1986). *UNIX shell programming* . New York: Wiley-Interscience.

Axelsson, E., Claessen, K., & Sheeran, M. (2005). Wired: Wire-aware Circuit Design . *Proceedings of Conference on Correct Hardware Design and Verification Methods (CHARME).* Springer Verlag.

Backus, J. (1978). Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Commun. ACM* , (pp.613-641).

Becker, J., Rosemann, M., & Uthmann, C. v. (2000). Guidelines of business process modeling. *Business Process Management: Models, Techniques, and Empirical Studies, Volume 1806 of LNCS.*, pp. 30-49.

Bentley, J. (1986). Programming pearls: little languages. *Commun. ACM* , (pp.711-721).

Bjesse, P., Claessen, K., Sheeran, M., & Singh, S. (1998). Lava: Hardware Design in Haskell. *Proceedings of International Conference on Functional Programming* (pp. 174-184). ACM Sigplan.

Caruana, G., & Pace, G. (2007). Embedded Languages for Origami-Based Geometry. *Proceedings of Computer Science Annual Workshop.* Malta: Departments of Computer Science and AI, University of Malta.

Chamberlin, D. D., & Boyce, R. F. (1974). SEQUEL: A structured English query language. *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control* (pp. 249-264). Ann Arbor: ACM.

Cheney, J., & Hinze, R. (2003). *First-Class Phantom Types.* Cornell University.

CIO. (2004). *Outlook 2004: Priorities 1Q InformationWeek Research, January 2004 Merrill Lynch CIO Survey Results.*

Claessen, K. (2001, April). Embedded languages for describing and verifying hardware. Dept. of Computer Science and Engineering, Chalmers University of Technology. Ph.D. thesis.

Claessen, K., & Hughes, J. (2000). QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices* , (pp.268-279).

Claessen, K., & Pace, G. (2002). An Embedded Language Framework for Hardware Compilation. *Proceedings of Designing Correct Circuits 2002.* Grenoble, France.

Claessen, K., & Sands, D. (1999). Observable Sharing for Functional Circuit Description. *Proceedings of Asian Computer Science Conference (ASIAN)* (p. 12). Springer Verlag.

Cook, B., Launchbury, J., & Matthews, J. (1998). Specifying superscalar microprocessors in Hawk. *Proceedings of Formal Techniques for Hardware and Hardware-like Systems.* Marstrand.

Coplien, J. (2004). *A pattern definition.* Hillside.net.

Cordina, J., & Pace, G. (2006). Functional HDLs: A Historical Overview. *Proceedings of Computer Science Annual Workshop.* Malta: Department of Computer Science and AI, University of Malta.

Elliott, C., & Hudak, P. (1997). Functional Reactive Animation. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97), SIGPLAN Notices 32(8):*, (pp. 263-273).

Elliott, C., Finne, S., & Moor, O. d. (2000). Compiling Embedded Languages. *Proceedings of SAIG*, (pp. 9-27).

Fasbinder, M. (2007). WebSphere Business Modeler Advanced Simulation. *IBM developerWorks article* .

Feurzeig, W., & Lukas. (1972). LOGO--A programming language for teaching mathematics. *Educational Technology 12* , (pp.39-46).

Foerster, A., Engels, G., & Schattkowsky, T. (2005). Activity diagram patterns for modeling quality constraints in business processes. *Proceedings of the MoDELS Conference. Volume 3713 of LNCS.*, pp. 2–16. Springer.

Giese, H., & Wagner, R. (2006). Incremental Model Synchronization with Triple Graph Grammars. *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006. Volume 4199 of LNCS.*, pp. 543–557. Springer.

Grima, M., & Pace, G. (2007). An Embedded Geometrical Language in Haskell: Construction, Visualisation, Proof. *Proceedings of Computer Science Annual Workshop.* Malta: Departments of Computer Science and AI, University of Malta.

Guceglioglu, A. S., & Demirors, O. (2005). Using software quality characteristics to measure process quality. *Proceedings of 3rd International Conference on Business Process Management. Volume 3649 of LNCS.*, pp. 374-379. Springer.

Halbwachs, N., Caspi, P., Raymond, P., & Pilaud, D. (1991). The synchronous data-flow programming language (LUSTRE). *Proceedings of the IEEE* , (pp.1305-1320).

Hindley, J. R., & Seldin, J. P. (1980). *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* New York: Academic Press, Inc.

Hinze, R. (2003). Fun with Phantom Types. In J. Gibbons, & O. de Moor (Eds.), *The Fun of Programming* (pp. 245-262). Palgrave Macmillan.

Hudak, P. (1989). Conception, Evolution, and Application of Functional Programming. *ACM Comput. Surv.* , (pp.359-411).

Hudak, P. (1996). Haskore Music Tutorial. *Proceedings of Advanced Functional Programming, LNCS 1129* (pp. 38-67). Springer-Verlag.

Hudak, P. (1996). Building domain-specific embedded languages. *ACM Computing Surveys* , (pp.196).

Hudak, P. (1998). Modular domain-specific implementation and exploration framework for embedded software platforms. *DAC '05: Proceedings of the 42nd annual conference on Design automation* , (pp.254-259).

Hudak, P., & Fasel, J. H. (1992). A gentle introduction to Haskell. *SIGPLAN Not.* , (pp. 1-52).

Hudak, P., & Jones, M. P. (1994). *Haskell vs. Ada Vs. C++ vs Awk vs \ldots An Experiment in Software Prototyping Productivity.*

Hughes, J. (1990). Why functional programming matters. *Research topics in functional programming* , pp. (pp.17-42).

IBM. (2006). *Redpaper: Best Practices for Using WebSphere Business Modeler and Monitor.* IBM's International Technical Support Organization (ITSO).

IBM. (2007). *WebSphere® Business Modeler V6.0.2: Process modeling overview and enhancements.* IBM Software Group.

Jones, M. P. (2000). Type Classes with Functional Dependencies. *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems* (pp. 230-244). London: Springer-Verlag.

Jones, S. P. (2003). *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press.

Jones, S. P. (2007). A taste of Haskell - a three hour tutorial about Haskell. *O'Reilly Open Source Convention* . Portugal.

Jones, S. P., Eber, J.-M., & Seward, J. (2000). Composing contracts: an adventure in financial engineering. *ACM SIG - PLAN Notices* , (pp.280-292).

Jones, S. P., Jones, M., & Meijer, E. (1997). Type classes: exploring the design space. *Proceedings of the Haskell Workshop 1997.*

Kamin, S. (1998). *Research on domain-specific embedded languages and program generators.* Electronic Notes in Theoretical Computer Science, Elsevier Press.

Kay, M. (2005, April). XSL Transformations (XSLT) Version 2.0. W3C.

Koehler, J., & Vanhatalo, J. (2007). Process anti-patterns: How to avoid the common traps of business process modeling, Part 1 modeling control flow, Part2 modeling data flow. *IBM WebSphere Developer Technical Journal 10.2, 10.4* .

Koehler, J., Gschwind, T., Küster, J., Pautasso, C., Ryndina, K., Vanhatalo, J., et al. (2007). Combining Quality Assurance and Model Transformations in Business-Driven Development. *Proceedings of Applications of Graph Transformations with Industrial Relevance 2007 (AGTIVE 2007)*, (pp. 1-16).

Koehler, J., Hauser, R., Sendall, S., & Wahler, M. (2005). Declarative techniques for model-driven. *IBM Systems Journal , 44* (1), (pp.47-65).

Koehler, J., Hauser, R., Küster, J., Ryndina, K., Vanhatalo, J., & Wahler, M. (2006). The role of visual modeling and model transformations in business-driven development. *Proceedings of the 5th International Workshop on Graph Transformation and Visual Modeling Techniques, Elsevier(2006)*, (pp. 1-12).

Küster, J. (2006). Definition and validation of model transformations. *Proceedings of Software and Systems Modeling (SoSyM) 5(3)*, (pp. 233-259).

Küster, J., & Abd-El-Razik, M. (2006). Validation of Model Transformations - First Experiences Using a White Box Approach. *Proceedings of MoDELS Workshops. Volume 4364 of LNCS.*, pp. 193-204. Springer.

Lämmel, R. (2006-2007). Google's MapReduce Programming Model -- Revisited.

Landin, P. J. (1966). The next 700 programming languages. *Commun. ACM* , (pp.157-166).

Lange, C., Dubois, B., Chaudron, M., & Demeyer, S. (2006). An experimental investigation of UML modeling conventions. *Proceedings of MoDELS Conference. Volume 4199 of LNCS.*, pp. 24-41. Springer.

Leijen, D., & Meijer, E. (1999). Domain specific embedded compilers. *Proceedings of Domain-Specific Languages*, (pp. 109-122).

Liang, S., Hudak, P., & Jones, M. (1995). Monad transformers and modular interpreters. *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (pp. 333-343). San Francisco: ACM.

Matthews, D. C. (1987). Static and Dynamic Type Checking. *Proceedings of Workshop on Database Programming Languages* (pp. 67-73). Cambridge: Computer Laboratory,University of Cambridge.

Meijer, E., & Drayton, P. (2004). Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. *Proceedings of OOPSLA'04 Workshop on Revival of Dynamic Languages.* Vancouver: Microsoft Corporation.

Meijer, E., Beckman, B., & Bierman, G. (2006). LINQ: reconciling object, relations and XML in the .NET framework. *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (p. 706). Chicago: ACM.

Mendling, J., Moser, M., Neumann, G., Verbeek, H., van Dongen, B., & van der Aalst, W. (2006). Faulty EPCs in the SAP reference model. *Proceedings of the 4th International Conference Business Process Management (BPM 2006). Volume 4102 of LNCS.*, pp. (pp.451–457). Springer.

Mens, T., & Gorp, P. (2006). A Taxonomy of Model Transformation. *Electr. Notes Theor. Comput. Sci. 152* , (pp.125-142).

Micallef, L., & Pace, G. (2007). Embedded Domain Specific Languages for Business or Personalization Rules. University of Malta, Assigned Practical Task, CSA 3009.

Mitra, T. (2005). Business-driven development. *IBM developerWorks article* , http://www-128.ibm.com/developerworks/webservices/library/ws-bdd/.

Moggi, E. (1989). Computational lambda-calculus and monads. *Proceedings of the Fourth Annual Symposium on Logic in computer science* (pp. 14-23). Piscataway: IEEE Press.

Nilsson, H., Courtney, A., & Peterson, J. (2002). Functional reactive programming, continued. *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell* (pp. 51-64). Pittsburgh: ACM.

Novatnack, J., & Koehler, J. (2004). Using patterns in the design of inter-organisational systems - an experience report. *Proceedings of the Workshop on Modeling Inter-Organisational Systems. Volume 3292 of LNCS.*, pp. 444–455. Springer.

O'Donnell, J. (1987). Hardware description with recursive equations. *Proceedings of IFIP 8th International Symposium on computer Hardware Description Languages and their Applications*, (pp. 363-382). North-Holland.

O'Donnell, J. (1993). Generating netlists from executable circuit specifications in a pure functional language. *Proceedings of Functional Programming* (pp. 178-194). Glasgow: Springer Workshops in Computing.

O'Donnell, J. (1996). From transistors to computer architecture: Teaching functional circuit specification in hydra. *Proceedings of Functional Programming Languages in a Education.* Springer Verlag.

OMG. (2005). *Unified Modeling Language: Superstructure 2005, Version 2.0 formal/05-07-04.* Object Management Group (OMG).

OMG. (2008). *Business Process Modeling Notation Specification 2008, Version 1.1 formal/2008-01-17.* Object Management Group (OMG).

Pace, G. (2007). HeDLa: A Strongly Typed, Component-Based Embedded Hardware Description Language. *Proceedings of Computer Science Annual Workshop.* Malta: Departments of Computer Science and AI, University of Malta.

R. M. Herndon, J., & Berzins, V. A. (1988). The Realizable Benefits of a Language Prototyping Language. *IEEE Trans. Softw. Eng.* , (pp.803-809).

Rhiger, M. (2003). A foundation for embedded languages. *ACM Trans. Program. Lang. Syst.* , (pp. 291-315).

Russell, N., Hofstede, A. t., Aalst, W. v., & Mulyar, N. (2006). Workflow Control-Flow Patterns : A Revised View. *BPM Center Report BPM-06-22* .

Russell, N., Hofstede, A. t., Edmond, D., & Aalst, W. v. (2004). Workflow Data Patterns. *QUT Technical report, FIT-TR-2004-01* .

Sheeran, M. (1984). muFP, a language for VLSI design. *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming* (pp. 104-112). Austin: ACM.

Sheeran, M. (1990). Describing Hardware Algorithms in Ruby. *Proceedings of Functional Programming.* Glasgow: SpringerWorshops in Computing.

Sheeran, M. (2005). Hardware Design and Functional Programming: a Perfect Match. *j-jucs , 11* (7), (pp.1135-1158).

Sloane, T., Mernik, M., & Heering, J. (2003). *When and how to develop domain-specific languages.*

Steele, G. (1990). *Common Lisp: The Language.* Newton, MA, USA: Digital Press.

Thompson, S. (1999). *Haskell: The Craft of Functional Programming (2nd Edition).* Addison Wesley.

van der Aalst, W. (2000). Workflow verification: Finding control-flow errors using Petri-net-based techniques. *Proceedings of Business Process Management, Models, Techniques, and Empirical Studies, London, UK* (pp. 161–183). Springer-Verlag.

Vanhatalo, J., Völzer, H., & Leymann, F. (2007). Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. *Service-Oriented Computing – ICSOC 2007* , (pp.43-55).

Varró, D., Varró-Gyapay, S., Ehrig, H., Prange, U., & Taentzer, G. (2006). Termination Analysis of Model Transformations by Petri Nets. *Proceedings of Graph Transformations, Third International Conference. Volume 4178 of LNCS.*, pp. 260-274. Springer.

Wadler, P. (1990). Comprehending monads. *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming* (pp. 61-78). Nice: ACM.

Wadler, P. (1992). The essence of functional programming. *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (pp. 1-14). Albuquerque: ACM.

Wadler, P., & Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (pp. 60-76). New York: ACM.

Wadler, P., Hudak, P., Hughes, J., & Jones, S. P. (2007). A history of Haskell: being lazy with class. *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages* (pp. 12-1--12-55). New York, NY, USA: ACM.

Wahli, U., Avula, V., Macleod, H., Saeed, M., & Vinther, A. (2007). *Redbook: Business Process Management: Modeling through Monitoring Using WebSphere V6.0.2 Products.* IBM's International Technical Support Organization (ITSO).

Wildmoser, M., & Nipkow, T. (2004). Certifying Machine Code Safety: Shallow versus Deep Embedding. *Proceedings of Theorem Proving in Higher Order Logics (TPHOLs 2004)* (pp. 305-320). Springer.

Appendix A

A Tutorial on our Language

A.1. Introduction

This is a tutorial on our language. Its main objective is to introduce all the features of the language and assist a new user to start off using the language. It should also be used as a reference guide, for users to be aware of the specific restrictions and checks that are carried out at compile-time to ensure the correctness of the constructed models. Users must keep in mind that the language captures precisely the semantics of the modelling language used within IBM WebSphere Modeler[1] (in particular IBM WebSphere Modeler Advanced version 6.0.2[2]). Thus various concepts, features and the modelling elements themselves are similar to those in this modelling tool. For this reason, we hope that such a tutorial serves as a good introduction to our language to anyone who uses this modelling tool.

Although a model in our language is effectively a Haskell program, if someone simply wants to construct models and use the functionality provide by our language, then no knowledge of Haskell or any other programming skills are required. If an advanced user wants to extend the language with new built-in types or functionality, then some knowledge of Haskell is assumed.

For complete sample models, refer to Appendix B, '*Sample Models and Case Studies*'. These samples are also provided on the CD, together with a directory named '*ProjectTemplate'* which has the necessary template scripts, which should helpful for a modeller to start defining models of a new project.

A.2. Getting Started

We are presenting a language and not really a notation; so you must be prepared to use the keyboard rather than the mouse! For instance, Figure A.1 can be defined using the one line definition in Listing A.1

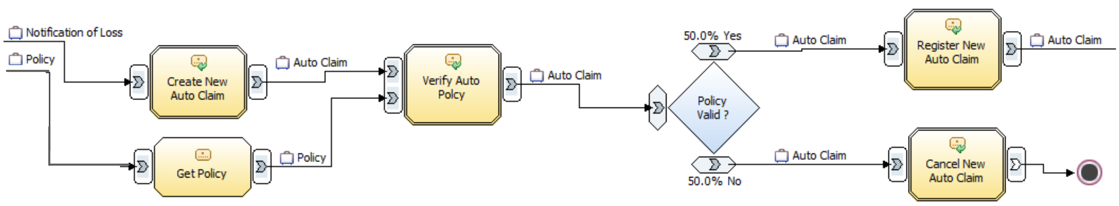


Figure A. 1: Handling an auto-claim submission *(similar to example provided as a sample with tool)*

[1] http://www-306.ibm.com/software/integration/wbimodeler/
[2] http://www.ibm.com/developerworks/websphere/zones/businessintegration/roadmaps/modeler/roadmap_advanced.html

```
pfACSubmission = (tCreateNewAC -|- tGetPolicy)->>- tVerifyAutoPolicy
                   -<|("Policy Valid?", (branchProp eYes 0.5, branchProp eNo 0.5) )|>=
                   (tRegisterNewAC, (tCancelNAC |><| stop))
```

Listing A. 1: Defining the process fragment in Figure A.1 using our language

Writing Scripts

Thus, you shall have to write simple scripts with any text editor of your preference. Each document or script shall represent a module (actually a Haskell module) where a number of definitions, types, modelling elements, process fragments, models and global processes shall be defined. For instance, if we want to create a module named `MyProcesses`, the following must be included in your script:

```
module MyProcesses where
       import FuncBPML
```

The name of module must start off with an uppercase letter. Following the module heading, it is important to specify the modules that need to be imported and used. Thus to use our language it is essential to import module `FuncBPML`. After that, you can start defining your processes and definitions. Before running the scripts, it is important to save the script using the same name as that of the module, in this case `MyProcesses.hs`.

Together with the language a, number of sample models are provided. These also include the same sample models which come along with IBM WebSphere Business Modeler Advnaced v6.0.2. These are defined in our language and should help new users learn how to use our language. Since a modeller might want to organize processes in projects, in a way similar to IBM's tool, then a directory named '*ProjectTemplate*' is provided. This contains two script files for the modules `Types`, where the user-defined types can be defined, and `Tasks`, where the tasks used by the processes in the project can be specified. This directory contains a sub-directory named '*TemplateProcess'*. This directory contains a template script file for a process, to illustrate how a user should start off defining a new process. If the user wants to control the resources which can be exported to other projects, another script file in '*TemplateProcess'* is provided. This module simply imports specific modules and indicates what can be exported. In this way, resources in different projects can be shared.

Running the Scripts and the Language Functions

The language is merely a collection of Haskell modules and the models defined with our language are effectively Haskell programs. These are thus dynamically compiled and executed using a Haskell compiler. The compiler which was used during the development of the language was GHC v6.8.2 (Glasgow Haskell Compiler version 6.8.2, released on 12th December 2007). This version of the compiler should preferably be installed. This can be downloaded for free from[1]. If other more recent versions of the compiler are released, they might still be fine for our language. More details about GHC can obtained from[2].

[1] http://www.haskell.org/ghc/download_ghc_682.html
[2] http://www.haskell.org/ghc/

It is important to note that most of the checks are carried out at compile-time and thus, if the combined models are not compatible then an error would be generated by the compiler. Thus, it is wise for the modeller to compile the scripts at various stages while constructing the models.

Once the compiler is installed, the user should open the ghc interpreter, by the command `ghci` and load the required modules. It is important that the compiler knows where the language libraries are located as well as the modules that are imported by the currently loaded module. If these modules are located in different directories then it is important for the user to open the ghc interpreter, by using the `-i` option as indicated below:

```
ghci -I".\\FuncBPML";".\\MyProjects\\ABCProj"
```

The different directory paths must be separated by a semicolon and there should not be any spaces between the arguments. The compiler will then search for modules within these directories.

The module should then be loaded by using the `:l` option followed by the name of the module. If there are errors in the definitions within the module, then the compiler would fail and displays an error message. If not then '*ok*' is displayed and the user can run the functions he defined or the functions provided in the languages.

A.3. Basic and Complex Types

IBM WebSphere Business Modeler (WSBM) provides a set of built-in basic types. It also allows users to define their own complex types such as business items. In our language, the same basic types are provided to the user and new complex types can be defined.

Basic Types

The following table (Table A.1) illustrates the basic types defined in IBM WebSphere Business Modeler and the same types defined in our language.

IBM WSBM	Our Language		
	Name of Type	*Type as an Input Argument*	*Example of a Defined Instance*
Boolean	TBoolean	bvTBoolean	BV True
Byte	TByte	bvTByte	BV 2
Date	TDate	bvTDate	BV (Date {wday = Monday, day = 2, month = May, year = 2008})
DateTime	TDateTime	bvTDateTime	BV (DateTime (Date {wday = Monday, day = 2, month = May, year = 2008}) (Time {hours = 10, mins = 2, secs = 90}))
Double	TDouble	bvTDouble	BV 0.000455632445
Duration	TDuration	bvTDuration	BV None Or BV (Duration {durDays = 2, durHours = 11, durMins = 58, durSecs = 50, durMsecs = 90})
Float	TFloat	bvTFloat	BV 12.8999
Integer	TInteger	bvTInteger	BV 2147483640
Long	TLong	bvTLong	BV 21474836400
Short	TShort	bvTShort	BV 130
String	TString	bvTString	BV “Product A”
Time	TTime	bvTTime	BV (Time {hours = 10, mins = 2, secs = 90})

Table A. 1: Basic Types

The names of the basic types in our language are the same as those in WSBM with the only difference that a ‘`T`’ has been added in front to indicate that these are the basic types of the tool and not those of the host language Haskell. When the type of some element needs to be specified as an input argument to a function then the values in the third column should be used. The naming convention used in this case is ‘`bv`’ to indicate that a basic value of a basic type is expected, followed by the name of the type in our language. When an instance of the type needs to be specified, the constructor `BV` needs to be included as illustrated above to indicate that it is a basic (and not a complex) value (Complex Types are discussed in the next sub-section). The specifications of the types are equivalent to those of WSBM and thus for more details see the user manual of the tool. In certain cases, where values allowed by the types overlap, the user might want to specify the required type by adding a type cast, in the following manner, `(BV 0.8768) :: TDouble`.

Complex Types

To define a new complex type, such as a business item named `Order`, the definitions in Listing A.2 are required.

```
newtype Order  = Order () deriving Typeable
type TOrder    = BI Order

instance ComplexType (TOrder)

biTOrder  = dType :: TOrder
```

Listing A. 2: Defining the business item `Order` as a new complex type

`Order` is the name of the new business item we want to define. We use the same naming convention as that in basic types and thus, we define a type synonym `TOrder` *(type TOrder = …)* to be able to refer to the newly introduced type. We also specify that `Order` is a business item by using the constructor `BI` *(type TOrder = BI Order)* and we specify that it is a complex and not a basic type *(instance ComplexType (TOrder)*). We also define `biTOrder` as the object which shall be passed on as an input argument (where necessary) to indicate this type. Once again, we decided to keep with the naming convention adopted for basic types, such that the first two characters `bi` indicate that this is a business item, followed by the name of the type.

These types can be declared once in a module and used when necessary by importing that module. For example, if the user places them in a module named `MyTypes`

```
module MyTypes where
       .........
```

and saves it as `MyTypes.hs`, then by importing it in the module where the process is defined (as illustrated below), the modeller would have access to these types.

```
module MyProcess where
       import MyTypes
       ......
```

A.4. Basic Modelling Functions

The language provides the same basic modelling elements as those provided in WSBM. Some new constructors were included in our language to explicitly define certain properties of the elements. To ensure the production of good quality models, various checks are carried out during the composition of models. If the combined modelling elements are not compatible, the compiler would generate an error as soon as the model is compiled. Thus, it is important for the user to be aware of the restrictions and checks that are carried out when defining and using these components.

A.4.1. Task

Tasks are the most essential elements within a process since they illustrate the activities that need to be carried out. They are usually defined globally and used in various processes within the project and thus it is important for these components to be easily defined and re-used. In our language a task is a function which given some input produces some output. When a task is defined, it is essential for the user to focus on the data flowing through the task rather than where and how it shall be used. Thus, focusing on just the flow of data, the following tasks in Figures A.2 and A.3 created in WSBM, can be defined, in our language, as illustrated on in Listing A.3 and A.4.

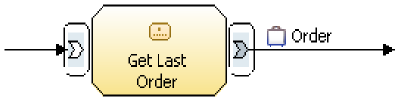


Figure A. 2: Task ‘Get Last Order’ in WSBM

tGetLastOrder = task “Get Last Order” (noData :-> biTOrder)

Listing A. 3: Task ‘Get Last Order’ in our language

Figure A. 3: Task ‘Increase Price’ in WSBM

tIncPrice = task “Increase Price” ((biTProduct, bvTFloat) :-> biTProduct)

Listing A. 4: Task ‘Increase Price’ in our language

In our language, a task is defined by using the function `task`. As input arguments the user must specify the name of the task and the types of the input and output data. The types of all the inputs and outputs must be explicitly defined, by using the arguments representing the types. If the task does not require any data as input or does not produce any data as output, `noData` should be used. The input argument types must be separated from the output argument types with the operator `:->`. Multiple inputs or outputs must be defined as a tuple as illustrated in the second example (Listing A.4). The previous definitions (Listing A.3, A.4) are actually representing tasks as functions with the following inputs and outputs (Figure A.4, A.5):

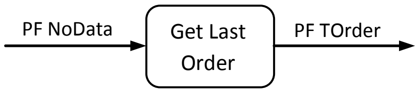


Figure A. 4: Actual type of task 'Get Last Order' in our language

tGetLastOrder = task “Get Last Order” (noData :-> biTOrder)

Listing A. 3: Task ‘Get Last Order’ in our language

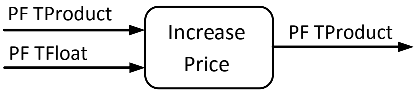


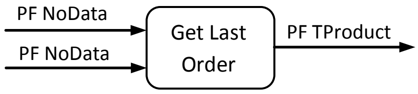
Figure A. 5 : Actual type of task 'Increase Price' in our language

tIncPrice = task “Increase Price” ((biTProduct, bvTFloat) :-> biTProduct)

Listing A. 4: Task ‘Increase Price’ in our language

The actual input and output types are defined in terms of `PF` (short for Process Fragment). This indicates that the function must obtain its inputs and pass its outputs to other process fragments, which would be attached to their inputs and outputs.

Although multiple inputs and outputs can be defined, our language would not allow the user to have multiple inputs or outputs specified as `noData`. This avoids redundancy. Thus the following (Figure A.6) is not allowed:

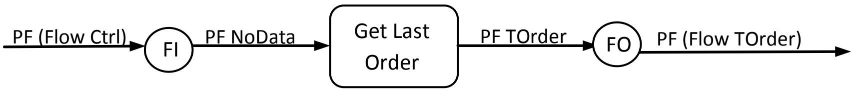


tGetLastOrder = task “Get Last Order” ((noData, noData) :-> noData)

Figure A. 6: A task with multiple inputs of type PF NoData

Defining the Input Source and Output Target Types

For a task to be used in a process, according to WSBM, the *input source* and *output target* must be specified. Thus, once the tasks are defined, possibly in a separate module, the modeller would select one of these tasks, specify the input source and output target type and use it in a process. For instance, let us assume that the modeller wants to use task `Get Last Order`. Since it does not require any data as input, the input source can only be defined as a `Flow`. The output target, on the other hand, can be set to `Flow` or `Repository`. Assume that the user wants a `Flow` as input and a `Flow` as output; in our language, new components must be used, to specify these input source and output target types:



ttGLO = flowCI <| tGetLastOrder |> flowCO

Figure A. 7: Specifying an input source and output target type for task 'Get Last Order'

The new components `flowCI` (Flow Connector In) and `flowCO` (Flow Connector Out) are attached to the input and output of the task by using the operator `<|` to indicate the type of input source and `|>` to indicate the output target type. In this way, the modeller would create a particular typed instance of the task `tGetLastOrder` which he can specifically use in his process, to guarantee that a process fragment which provides a control flow would be attached to the input of the task and only a process fragment which requires an input flow of type `TOrder`, would be attached to its output.

Now let's assume that the modeller also wants to include the task `Increase Price` in his process. He wants to specify that the product is obtained from a repository containing the details of the products and he wants the price of the product to be increased by a constant value, which he specifically wants to define. Once the price is increased, the product must be placed in another repository. Thus, the following components must be attached to the inputs and outputs to specify the input source and output target types he requires for the task in this particular process:

ttIncPrice = (repCI, constCI) <| tIncPrice |> repCO

Figure A. 8: Specifying input source and output target types for the task 'Increase Price'

For this reason, only components such as the following would be allowed to be attached to this typed task:

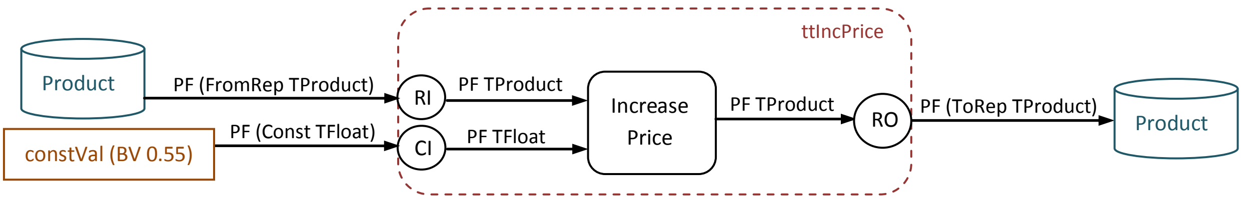


Figure A. 9: Possible valid components that can be attached to typed instance of task 'Increase Price' (ttIncPrice)

The previous process fragment (Figure A.9) is valid and correctly typed since all of its components are compatible in terms of data and connection type. In WSBM it would be defined as follows (Figure A.10):

Figure A. 10: Process fragment in Figure A.9 modelled in WSBM

If on the other hand, the user wants to use a task without restricting the input source and output target types, and would like them to be *inferred* automatically when these are connected to other modelling elements, then specific functions (such *lazy serial composition* `->>-` instead of conventional *serial composition* `->-`) defined in connection patterns section can be used to reduce the amount of specifications that the user needs to define when creating the model.

Adding Additional Control Flow Inputs and Outputs

The only problem with `ttIncPrice` in Figure A.8 is that it does not have an inflow. Thus, if this task had to be defined in a process it would never execute since it never gets the control to do so. For this reason, the function `hasInFlow` can be used to check whether the process fragment has an input data or control flow (in this case, `hasInFlow ttIncPrice = False`). If not, the user can use the function `addInFlow` for an additional control flow to be added to the process fragment. This function adds a default control flow by using another component known as `ctrl_dataFlowCombiner` (to combine control flows with a data flow), such that the new process fragment would become (see Figure A.11):

ttIncPrice_withInCtrl = addDefFlow ttIncPrice

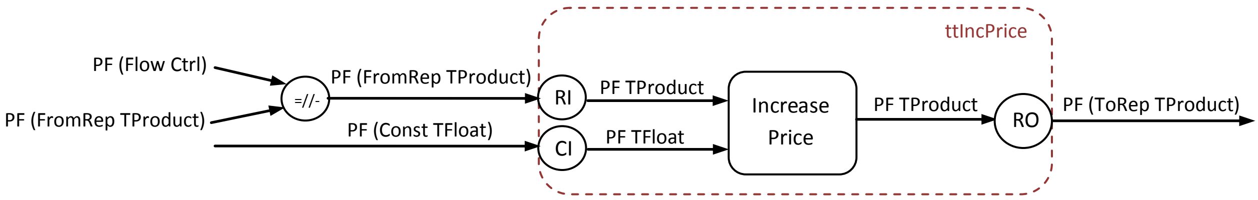


Figure A. 11: Adding a default control flow by using the function `addInFlow`

Thus, when a default flow is added using `addInFlow`, a `ctrl_dataFlowCombiner` =//- is attached to the first input such that the new process fragment would have an additional input. In this case, `ttIPcI` has three inputs of type `((PF (Flow Ctrl), PF (FromRep TProduct)), PF (Const TFloat))` and one output `(PF (ToRep TProduct))`.

Similar to `ctrl_dataFlowCombiner`, the language also provides a `ctrl_dataFlowSplitter` (to split up a data flow into a data flow and the required number of control flows). Thus, using these two components it is possible to add additional control flows to the inputs and outputs of a process fragment, in a similar way as in WSBM. In this way, although `ctrl_dataFlowCombiner` in the above example guarantees that the task gets control, once it executes, the control is not passed out to any other task, and thus the flow would terminate at that task. If the modeller wants to pass on the control to another modelling element in the process, a `ctrl_dataFlowSplitter` can be attached to the output of `ttIncPrice` as in the following Figure A.12.

ttIncPrice_withIOCtrl = ttIncPrice_withInCtrl ->- ctrl_dataFlowSplitter

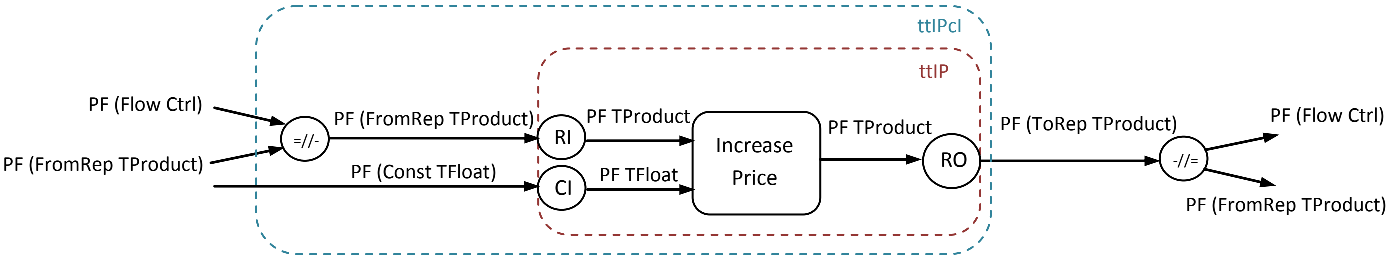


Figure A. 12: Adding outgoing control flow

The operator `->-` (serial composition) is actually a connection pattern (which shall be discussed later on) which allows the modeller to combine components, in this case, `ttIncPrice_withInCtrl` with `ctrl_dataFlowSplitter`. Although in the above diagram, `ttIncPrice_withIOCtrl` is depicted as having two outputs that is the pair `(PF (Flow Ctrl), PF (FromRep TProduct))`, the output of `ttIncPrice_withIOCtrl` is only determined when it is combined with another process fragment, depending upon the inputs of that process fragment. If the modeller wants to specify the specific output, then he can do so by the following definition:

```
ttIncPrice_withIOCtrl = (ttIncPrice_withInCtrl ->- ctrl_dataFlowSplitter)
                         :: (PF (Flow Ctrl), PF (FromRep TProduct))
```

`ctrl_dataFlowCombiner` and `ctrl_dataFlowSplitter` can flexibly be used to combine and split any number of control flows with data flows. The order of the control flows and data flow is irrelevant. The following (Figure A.13) illustrates some examples how these components can be used (`a` and `b` represent a data type):

Figure A. 13: Different ways how `cdCombine` and `cdSplit` can be used (a and b represent data types)

Similar components, `ctrlFlowsCombiner` and `ctrlFlowsSplitter` are provided to allow the modeller to add combine a number of control flows into one flow and split a control flow into multiple control flows, as illustrated below:

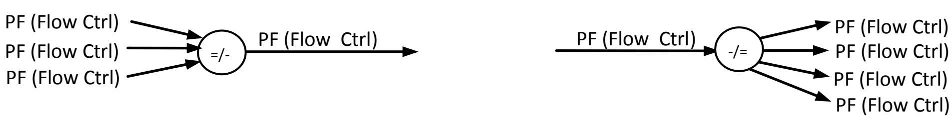


Figure A. 14: Different ways how `ccCombine` and `ccSplit` can be used (a and b represent data types)

These components are also important to define the following process fragments (Figure A.15):

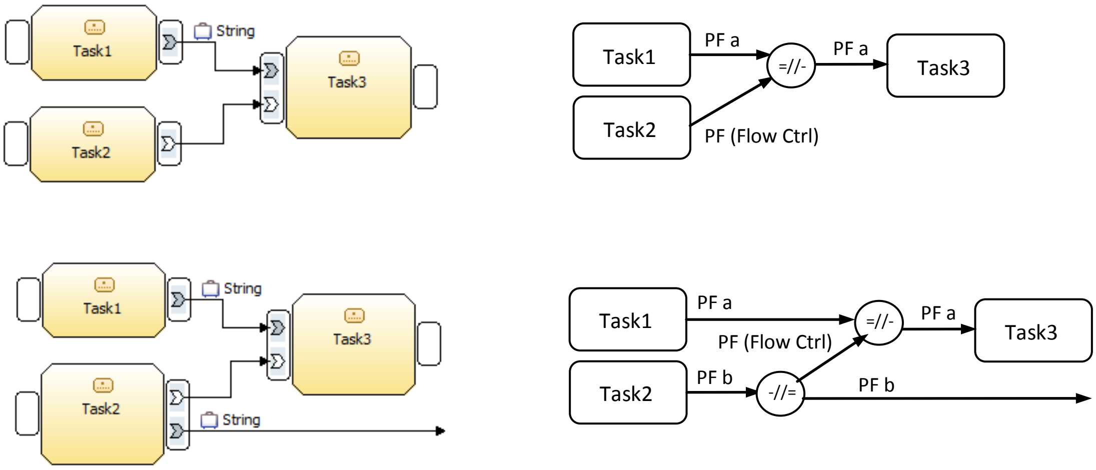


Figure A. 15: Models where the use of `cdCombine` and `cdSplit` is essential

The control flow from *Task2* to *Task3* is an additional control flow whose aim is to ensure that *Task3* does not start before both *Task1* and *Task2* terminate. *Task3* requires only the data from *Task1*. Thus in the second example (in Figure A.15) the output of *Task2* is not required and thus, a `ctrl_dataFlowSplitter` is used to split the flow into control and data and allow the control to pass to `ctrl_dataFlowCombiner` attached to *Task3*.

Final Remarks

In this way, similar to WSBM, globally defined tasks can be reused in various manners and adapted according to the required application within a process.

Different from WSBM, the language does not allow users to specify *minimum* and *maximum* amounts of values for inputs and outputs. Instead all the inputs and outputs have to be explicitly defined, even though they are of the same type. Moreover, there is no *activity* and *gateway form*. Rather than defining input and output criteria, the user is expected to define the models in terms of all the required modelling elements.

Input source types and *output target* types are used in the same way as defined in WSBM. Our language also guarantees that only the appropriate types are attached to the inputs and outputs of the modelling elements. Moreover, the same task can be used to create different instances with different input source and output target types.

When defining tasks, the name assigned to the task must be different from those used for other processes within the process fragment. To check whether the name has been used for any other component in the process fragment, the following function can be used, `isNameUsed`.

To specify that a connection type can be anything, `noConType` can be used. If on the other hand, the connection type should be specified automatically then rather then using example `repCI`, `infConType` can be used. This would, where possible, infer the connection type automatically.

A.4.2. Constants

In WSBM constants can be used as inputs to modelling elements. These are usually defined as attributes of the element. In our language, these are defined as other components in the model. They can only be attached to those inputs whose input source type is set to `Constant` (i.e. have input type `PF (Const a)` where `a` is a built-in or a user-defined complex type). Constants are defined as in the following examples (Listing A.5):

```
constVal (BV True)
constVal (BI ord1)
```
(assuming ord1 is an instance of the user-defined type `Order`)

Listing A. 5: Defining constants in our language

A.4.3. Repository

Similar to a task, a repository in our language must first be defined and then used within the process. Let us define the repository which we used previously to store the items of type `Product` (Listing A.6).

```
rProds = repository "Products" biTProduct
```

Listing A. 6: Defining a repository in our language

The name given to this repository is "`Products`" and in the definition of the process (that is in the script) it shall be referred to as `rProds`. The type of the items it shall store is defined as an input argument, in this case, `biTProduct` for `Product`. The type of the repository must be a basic or a complex user-defined type. Once defined, the repository needs to be used that is, it might be filled up with data by various activities and its output might be used by other activities. Example (refer to Figure A.16 and Listing A.7):

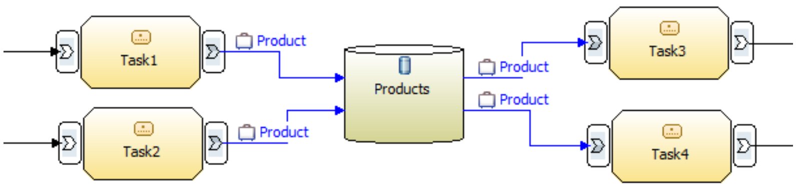


Figure A. 16: A process fragment (created in WSBM) illustrating how a repository can be filled in and used

```
pf1 x y = let
              orProd = rProd (ot1, ot2)
              ot1 = t1 x
              ot2 = t2 y
              ot3 = t3 orProd
              ot4 = t4 orProd
          in (ot3, ot4, orProd)
```

Listing A. 7: Defining the process fragment in Figure A.16 using our language

In Listing A.7, `ot1, ot2, ot3, ot4` are the outputs of tasks `t1, t2, t3, t4; x` and `y` are respectively the inputs to the process fragment and the inputs to `t1` and `t2`. As output, this process fragment produces a tuple of three elements. These include the output of `t3` and the output of `t4` and the output of the repository. If the repository was used locally in that process fragment then its contents would not have been passed out as output. However, since the repository in this fragment is meant to be global, then the content of the repository is generated as output so that it can be used as an input to other process fragments. If within the new process fragment the user wants to store other additional data, then a new repository would have to be defined as shown in Listing A.8. To indicate that the data is the current content of the repository, then the function `previousRepContent` should be used (Listing A.8).

```
rProds2 = repository "Products2" biTProduct

pf2 x y z = let
                (o1, o2, orProd1) = pf1 x y
                ot4 = t4 z
                orProd2 = rProd2 (previousRepContent orProd1, z)
             in (ot4, orProd2)
```

Listing A. 8: Adding data to a global repository in our language

The new repository named "`Products2`" contains the content of repository named "`Products`" and the data generated as output by task `t4`. The content of the repository named "`Products2`" is then generated as the output of process fragment `pf2` for other process fragments to have access to the content of this repository.

Thus, for a process to have access to a global repository, it should consume the output of the process which lately updated this repository and use it as a normal data input. This means that process fragments such as in Figure A.17, do not need to be defined in processes defined using our language:

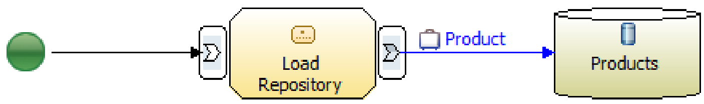


Figure A. 17: Process fragment to load the content of a global repository before this is used within a process

The modeller should be aware that once a repository is defined, then it must be filled in with some input and its output must be used either as an input to other modelling element/s in the process fragment (as in process fragment `pf1`, Listing A.11) or simply passed out as output (as in process fragment `pf2`, Listing A.11). This would thus prevent the production of redundant repositories which are later not used in process fragments or which are not filled up before being used.

A.4.4. Start, Stop, End Node

These nodes are handled in the same way as WSBM. The provided functions include `start, stop` and `end` and can be used as illustrated in Listing A.9.

```
pf1 = start ->>- t1
pf2 = t2 ->>- end
pf3 = t2 ->>- stop
```

Listing A.9: Using start, end, stop nodes in our language

In Listing A.9, `t1` and `t2` represent tasks and `pf1, pf2` and `pf3` represent the process fragments. The `start` node passes its input control to the element attached to its output (in this case `t1`). The `end` and the `stop` nodes take either a control or a data flow as input. In the case of a data flow, the type of the data must be either a basic or a user-defined complex type.

A.4.5. Decisions, Merges, Forks, Joins

Gateways in our language are handled in the same way as in WSBM. However, it is important for the user to know the checks that are carried out to ensure the appropriate combination of other components to these gateways.

Merges, Forks, Joins

Primarily, although the data types of all the inputs branches must be equivalent to that of the outgoing branches, as in WSBM, the input source and output target types can be different. Moreover, multiple data flows can flow through the branches of the gateway. However, if just a control flow is required for merges, forks or joins, then only one control flow is allowed. If the user wants to add additional control flows to the inputs or outputs of any of the gateways then `ctrl_dataFlowCombiner /ctrlFlowsCombiner` and `ctrl_dataFlowSplitter / ctrlFlowsSplitter` should be used, in a similar way as illustrated with tasks. The required number of incoming or outgoing branches would automatically be inferred depending upon the elements that are connected to the gateway. For example (Listing A.10):

```
merge, fork, join          e.g.:      om = merge (ot1, ot2, ot3)
                                  or
                                      (of1, of2) = fork ot1
                                      ot2 = t2 of1
                                      ot3 = t3 of2
```

Listing A. 10: Defining merge, fork and join in our language

To help the modeller compose models quicker and easier, the following functions are also provided (Listing A.11):

```
merge_withInPFs, fork_withOutPFs, join_withInPFs
                               e.g.:    merge_withInPFs (t1, t2, t3)
                                   or
                                         fork_withOutPFs (t2,t3)
```

Listing A. 11: Using functions `merge_withInPFs`, `fork_withOutPFs` and `join_withInPFs` in our language

While `merge_withInPFs` (*merge with incoming process fragments*) and `join_withInPFs` (*join with incoming process fragments)* take the process fragments, that need to be connected to the incoming branches of the gateway, as input, `fork_withOutPFs` (*fork with outgoing process fragments*) takes as input the process fragments that are attached to the outgoing branches of the fork.

Other operators which make model definitions easier to comprehend and reason about include:

`=>-` e.g.: `(pf1, pf2, pf3) =>- pf4`

i.e. a merge which has process fragments `pf1, pf2, pf3` connected to its incoming branches and process fragment `pf4` connected to its outgoing branch

`=|-` e.g.: `(pf1, pf2) =|-  pf3`

i.e. a join which has process fragments `pf1, pf2` connected to its incoming branches and process fragment `pf3` connected to its outgoing branch

`-|=` e.g.: `pf1 -|= (pf2, pf3)`

i.e. a fork which has process fragment `pf1` connected to its incoming branch and process fragments `pf2, pf3` connected to its outgoing branches

If the modeller wants to represent a fork-join then he can do so by using

`fork_join` e.g.: `fork_join (pf1, pf2, pf3, pf4)`

i.e. a fork which spawns its output to the four process fragments that execute in parallel and which are then connected to a join

If the modeller would like to user operators instead, then the above fork-join process fragment can be defined as follows

```
-|= (pf1, pf2, pf3, pf4) =|-
```

These are actually known as connection patterns because they take functions as inputs to produce a more complex function.

Inclusive and Exclusive Decisions

Different from the above gateways, for a *decision* to be defined, additional input arguments are required. Primarily the user must define whether he wants an *exclusive* or an *inclusive* decision. Both decisions take the same input arguments. To define an exclusive decision, the function `exclDecision` should be used and to define an inclusive decision, the function `inclDecision` is provided. In the following example (Listing A.12), an exclusive decision which is named "`Order Complete`?" is defined. It is assumed that, attached to its input the decision has process fragment `pf1` (thus, `opf1` is the output of `pf1`) and attached to its outgoing branches are process fragments `pf2` and `pf3`. The data that flows out of process fragment `pf1` is of type `(PF TInteger, PF (Flow TOrder))`.

```
eYes, eNo :: (PF TInteger, PF (Flow TOrder)) -> Bool
eYes = const True
eNo  = const False

(oOrderCompl, oOrderNotCompl) =
    exclDecision "Order Complete?" (branchProp eYes 0.4, branchProp eNo 0.6) opf1
opf2 = pf2 oOrderCompl
opf3 = pf3 oOrderNotCompl
```

Listing A. 12: Defining a decision in our language

Since the main purpose of a decision is to divert the input flow to one of the branches, an expression for each of the outgoing branches should be defined. The second input argument should be a tuple the size of the number of required outgoing branches, which contains a definition for each of the branches. The details for each branch are defined using the function `branchProp` which takes two inputs; the *expression* which determines whether that branch should be executed and the *probability* that the branch would execute. A run-time (not compile-time) a check ensures that the sum of the probabilities of the branches add up to 1. If not, then an error is generated. The expressions must be expressed as lambda functions (as illustrated in Listing A.16, for branch expressions `eYes` and `eNo`), which as input take the same type of data that flows as input into the decision and produces an output of type boolean. Since these expressions are defined as lambda expressions then it is important for the user to specify the type signature, in this case, `(PF TInteger, PF (Flow TOrder)) -> Bool`. Similar to other gateways, the input types of the process fragments attached to the outgoing branches of the decision must have the same data type as that of the input. The input source and output target type might be different. As an input, the decision cannot have just a control flow. It must have at least one data, since some data is required for the appropriate branch to be chosen. In this case, as input it has two types of data `(PF TInteger, PF (Flow TOrder))`. From the definition of `eYes` and `eNo` in Listing A.16, it can also be noted that the input source of one of the data flows is not defined (`PF TInteger`). In fact, gateways allow users to connect, for instances tasks, whose input source or output target type is not specifically defined. There is no limit on the number of outgoing branches.

As provided for other gateways, a number of functions that help users create simpler and more readable models, are provided for decisions. These include:

`exclDecision_withOutPFs,inclDecision_withOutPFs`

e.g.:
```
exclDecision_withOutPFs "Order Complete?"
                 (branchProp eYes 0.4, branchProp eNo 0.6) (pf2, pf3)
```

This is another way how to define the exclusive decision expressed earlier, with the only difference that in this case, this definition represents a process fragment (or function) rather than a value. In fact as input, `exclDecision_withOutPFs` (exclusive decision fragment with outgoing process fragments) (or the equivalent function for an inclusive decision), take the process fragments (or functions) that are connected to the branches of the decision rather than the output of these fragments. Although the input data type of the decision is known from the definition of the expressions assigned to each branch, the actual process fragment that is attached to its input is not defined. To connect `pf1` to the input of the decision, then the serial composition operator `->-` would have to be used, as follows:

```
pf1 ->- (exclDecision_withOutPFs "Order Complete?"
                 (branchProp eYes 0.4, branchProp eNo 0.6) (pf2, pf3))
```

If the modeller prefers to use operators to define such a decision, then the decision can be defined as follows:

```
pf1 -<| "Order Complete?" (branchProp eYes 0.4, branchProp eNo 0.6)|>= (pf2, pf3)
```

The new operators in this case are `-<|` and `|>=`. If an inclusive decision is required then the operator `-<||` can be used instead of `-<|`. Thus the previous definition can be expressed in terms of an inclusive decision in the following manner:

```
pf1 -<|| "Order Complete?" (branchProp eYes 0.4, branchProp eNo 0.6)|>= (pf2, pf3)
```

Similar to the fork-join, if the user wants to model an exclusive/inclusive decision followed by a merge, the following function can be used:

```
exclDecision_merge, inclDecision_merge
```

e.g.: `exclDecision_merge "Order Complete?"`
`(branchProp eYes 0.4, branchProp eNo 0.6) (pf2, pf3)`

i.e. an exclusion decision whose branches (which have the same properties as in the previous definitions) are attached to process fragments `pf2` and `pf3`. These fragments are then connected to a merge. The same applies for `inclDecision_merge`.

This definition is actually equivalent to

```
-<|| "Order Complete?" (branchProp eYes 0.4, branchProp eNo 0.6) |>= (pf2, pf3) =>-
```

Although these examples depict two branch decisions, any decision with any number of branches (minimum two) can be defined.

The modeller should make use of the above mentioned functions depending on what and how he wants to define elements in his model.

A.4.6. Packaging Process Fragments into a Sub-Process

In our language, processes or models are defined in the form of functions. Decomposing a complex process into fragments, which are essentially functions, would help the modeller to abstract away from the internal details of the process and to easily reason about and construct a complex model. In WSBM this is possible through the definition of processes or sub-processes. Although sub-processes in WSBM and process fragments, such as the ones defined in the previous examples, are both used to help the modeller decompose models and abstract away details of the process, a sub-process is essential different in that it is considered as one modelling element, which internally defines a process which has a start node and a stop node. For this reason, process fragments and sub-processes do not represent the same modelling structure.

To allow users to package a number of process fragments into one sub-process, we provide the function `pkSp`. Thus assuming that a modeller has defined process fragments `pf1` and `pf2` whereby `pf1` takes an input of type `(PF TString, PF (Rep TOrder))` and `pf2` takes a control flow as input, he can package them into a sub-process named "`Verify Order`" in this manner (see Listing A.13):

```
spVerifyOrder = packageSubProcess "Verify Order" (pf1, pf2)
```

Listing A. 13: Defining a sub-process in our language

One of the most important checks that is carried out, is to ensure that there is at least one of the process fragments that has a stop node attached to one of its outputs. This would ensure that once the process terminates, the outputs are always returned back to the parent process. The function `packageSubProcess` adds a start node to all the control flow inputs of process fragments (in this case, a start node is attached to the input of process fragment `pf2`) and identifies the data input types of the sub-process. Thus, in this case, the input type of the sub-process is `(PF TString, PF (Rep TOrder))` since the input of the other process fragment (`pf2`) is a control flow (which is inherently obtained from the data flow). If all of the process fragments had a control flow as input, then the input type of the sub-process would be `(PF (Flow Control))`. Similarly, the output type of the sub-process is identified, depending upon the output types of the internal process fragments. Although in this example, the sub-process contains two process fragments, any number of process fragments can be packaged together into one sub-process.

In this way, the sub-process in our language would be considered as one modelling element. Additional control inputs or outputs can be added in the same way as done with tasks. Moreover, if the user wants to specifically specify the input source and output target types of certain inputs and outputs of the sub-process then he can do so by using the operators `<| |>` as illustrated with tasks.

A.5. Connection Patterns

Since in our language models and process fragments are essentially function which given an input produce some other, as noted in some of the previous examples, it is possible to link one component or process fragment to another by either using the output of one as the input of the other or by using connection patterns. On these connection patterns which has been introduced in the previous examples is the serial composition operator `->-`, which combines two functions or process fragments into one function or fragment. Thus, connection patterns are essentially functions which take some functions as input and produce a more complex function. Various connection patterns have been defined in our language to help modellers create models quicker and in a concise way, thus reducing the chance of introducing errors and producing models which are easier to read and understand.

The following connections patterns are provided:

- ***To define gateways***

    ```
    exclDecision_withOutPFs, inclDecision_withOutPFs,
    fork_withOutPFs, merge_withInPFs, join_withInPFs
    exclDecision_merge,inclDecision_merge, fork_join
    -<|…|>=, -<||…|>=, =>-, -|=, =|-
    ```

    All to these have be defined in section A.4.5 5

- ***To terminating a Flow or a Process using End and Stop Nodes***

    If the user wants to attach an end node or a stop node to any one of the outputs of a task or a sub-process independent of the output and whether the output type of the activity is specified, the following function can be used:

    `|><|` e.g.: `t1 |><| stop` or `t1 |><| end`

    `t1` represents a particular task whose output is attached to a `stop` or `end` node. Thus the operator `|><|` expects an activity as the first input and stop or end as the second input.

    If for instance, one of the flows flowing out of a gateway should be terminated, then one of the following functions can be used

    `endBranch, stopBranch`

    e.g.: `-<| "Order Complete?" (branchProp eYes 0.4, branchProp eNo 0.6) |>=`
    `(endBr biTOrder, stopBr biTOrder)`

These functions take as input an argument that represents the type of the data flow that should be terminated. Thus in this case, the input argument `biTOrder` indicates that the type of the data that flows through the decision is `TOrder`. The above definition actually represents the process fragment in Figure A.18.

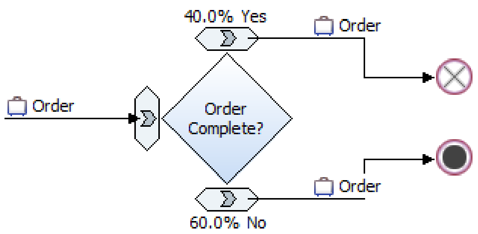


Figure A. 18: Exclusive decision with branches terminated with an end and a stop node

- ***To define flow splitters and combinators***

```
=/-, -/=        e.g.:  (pf1, pf2) =/- pf3         pf1 -/= (pf2, pf3, pf4)
-//=, =//-      e.g.:  (pf1, pf2, pf3) =/- pf4    pf1 -//= (pf2, pf3)
```

Operators with a single forward slash (`/`) indicate a control flow splitter or combinator, whereas a double forward slash (`//`) indicate a data flow splitter (which splits a data flow into some control flows and a data flow) or a control and data flow combinator (which combines multiple control flows and a data flow into a single data flow). Thus, `-/=` and `-//=` are used for `ctrlFlowsCombiner` and `ctrl_dataFlowCombiner` respectively, whereas `=/-` and `=//-` are used for `ctrlFlowsSplitter` and `ctrl_dataFlowSplitter` respectively. Using these operators instead of the usual functions, the user can specify the process fragments that are attached to its input and outputs more easily and produce models which are more readable.

If a data flow needs to be split up into a data and a control flow and a stop or an end node needs to be attached to the control flow, the following function can be used

```
-/=.        e.g.:  t1 -/=. stop          t1 -/=. end
```

`t1` represents the task whose data output is split up into a data and a control flow. The control flow is then terminated with a `stop` or an `end` node. The process fragment in Figure A.19 can be defined as `t1 -/=. stop`, and it could easily represent the last task in a process. If so, then the data flow would be connected to the output of the sub-process.

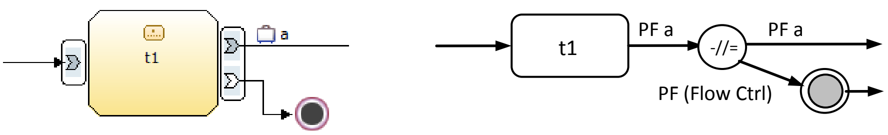


Figure A. 19: A task whose data output of type `a` is split up into a data and a control flow. The control flow is then terminated with a stop node. The fragment on the left was constructed using WSBM, whereas that on the right, depicts the same fragment using constructs defined in our language

- ***Serial and Parallel Composition***

```
->-   or  serial       e.g.:  pf1 ->- pf2    or    serial pf1 pf2
-|-   or  par          e.g.:  pf1 -|- pf2    or    par pf1 pf2
```

These are two of the most commonly used connection patterns, when constructing models. `->-` and -|- are infix operators which are respectively equivalent to the functions `serial` and `par`. When process fragments are put in sequence, the output type of first fragment must be strictly equivalent to the input type of the second fragment that is both the data type and the connection type must be equivalent. Thus if the output type of the first fragment is (`PF TString)` and the input type of the second fragment is (`PF (Flow TString))` then they cannot be connected together. The input and output types of fragments are not really relevant when par is used. These fragments would be grouped into one, such that if `pf1` has type (`(PF a, PFb) -> PF d)` and `pf2` has type (`PF c -> PF e)` then the new fragment would have type `((PF a, PF b), PF c) -> (PF d, PF e).`

If the same operations need to be carried out on more than two input fragments, then the following functions are provided:

```
serialC      e.g.: serial (pf1,pf2,pf3,pf4)
parC         e.g.: parC (pf1,pf2,pf3)
```

These functions (serial Composition and parallel Composition) can also be used to easily identify certain specific blocks in a process.

- ***Serial Lazy Composition***

If the user does not want to explicitly define the input source and output target types of tasks or activities, then it is possible to use the following lazy serial composition operator for the connection type to be automatically inferred depending upon the component it is connected to:

```
->>- or serialL      e.g.: pf1 ->>- pf2     or     serialL pf1 pf2
```

The modeller must be aware that although the connection type is inferred, the type of the data must be equivalent. Thus, if a task outputs a business item of type `TOrder`, then this task cannot be connected to a repository which stores business items of type `TProduct`.

If on the other hand, the output type of the first fragment and the input type of the second match perfectly (in terms of data and connection type) then ->>- acts as ->-. Thus if in a process the modeller wants to use activities, whose input source or output target types are defined, and others whose connection types are not defined and thus need to be automatically inferred on composition, then the user can uniformly use ->>- instead of the conventional ->-. In this way, unless the connection type of an activity really needs to be specified, users do not necessarily need to define the input source and output target types of activities. Thus, definitions would be more concise.

If more than one process fragment should be composed in sequence, the following function can be used

```
serialLC       e.g.: serialLC (pf1, pf2, pf3, pf4)
```

Else they can be defined in the following manner: `pf1 ->>- pf2 ->>- pf3 ->>- pf4`

If the user wants to pass on a value to a function and there is a possibility that the source type of the value or that of the fragment is not specified but the data type of two is the same, then the following function can be used:

`*>>-`   e.g.:   `x *>>- pf`

where `x` is a value which could be the output of another modelling element or fragment and `pf` is the process fragment which should take `x` as an input. Thus if the type of `x` is `PF TString` and the input type of `pf` is `PF (Flow TString)`, the operator would automatically deduce that the data types are equivalent and thus accept the composition.

Other Functions

Other functions which might also be useful when constructing models include:

```
swap     e.g.: swap (x,y) = (y,x)              -- to swap the elements in a pair
rev      e.g.: rev (w,x,y,z) = (z,y,x,w)  -- to reverse all the elements in a tuple of any size
swapIO, revIO                             -- to swap/reverse both the inputs and outputs of a function
```

The main difference between `swap` (and `swapIO`) and `rev` (and `revIO`) is that while the former operates on pairs, the latter operate on tuples of any size. The difference between `swap` (and `rev`) and `swapIO` (and `revIO`) is that while the former expects a pair (or tuple) as input, the latter expects a function which as input has a pair (or tuple) and another pair (or tuple) as output. For this reason, `swapIO` and `revIO` are connection patterns, whereas `swap` and `rev` are functions which can be useful when handling the actual output value of functions. Figure A.20 illustrates `swapIO` function given process fragment `pf` as input:

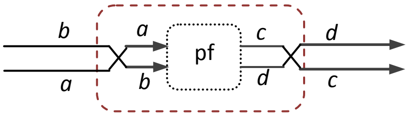


Figure A. 20: The function `swapIO`

Another function which is not a connection pattern but which might be useful especially when connection patterns are used is

`noActivity`   e.g.:   `noActivity bvTString`   or   `noActivity biTOrder`

This function is similar to the function **`id`** in Haskell. Thus, its main aim is to allow some data to flow through it without modifying it (hence the name `noActivity`, that is no Activity). The only difference is that it enforces the type of the data flow. For instance, in the first example (`noActivity bvTString`) a value of type `TString` is allowed to pass through. In the second example (`noActivity biTOrder`), a business item of type `TOrder` is expected. Any data type which is built-in or user-defined can be specified. This function is especially useful when the modeller needs a component that represents a plain connector, as in the following case:

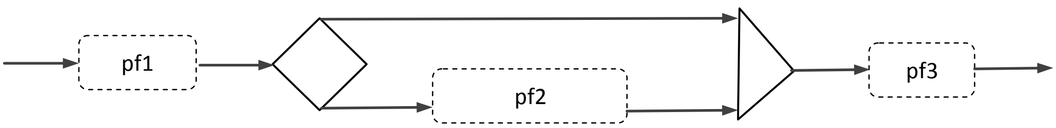


Figure A. 21: A decision with a 'do-nothing' branch which, assuming that the data flowing through the decision is of type `TOrder`, then in our language, it can be defined as:

```
pf1 -<|| "Order Complete?" (branchProp eYes 0.4, branchProp eNo 0.6)|>=
     (noActivity biTOrder, pf2) =>- pf3
```

The first decision branch does not carry out any activity (it just passes the data to merge) and thus, the function `noActivity` would be ideal to represent no activity on this branch.

If a tuple might contain other tuples as elements (e.g. `((a,b),c,(d,(e,f)))`) and all these need ungrouped into one single tuple (e.g. `(a,b,c,d,e,f)`) then the `ungrp` can be used. Similar to other functions, `ungrp` can can be connected in sequence with other components.

```
ungrp       e.g.:    ungrp ((a,b),c,(d,(e,f))) = (a,b,c,d,e,f)
```

Since there might be cases where components need to be attached together but one of the inputs should be obtained from the previous fragment, the rest should be obtained for instance from repository, to still be able to connect the components by using the provided serial connection patterns, `getSingledInputsFragment` or `curryFragment` is provided. This is essential the curry function provide in Haskell, with the only difference that the functions which we provide support functions which as input take any sized tuple. The two functions we provide are the same. The only difference is the name. Thus someone who is familiar with Haskell might be more intuitive for him to use `curryFragment` whereas a business modeller might understand better the term `getSingledInputsFragment`.

```
getSingledInputsFragment, curryFragment

      e.g.:   (getSingledInputsFragment tIncPrice) irProds
```

A.6. Built-In Models

Our language also provides a number of built-in models that can be used during construction of complex models. Besides, producing models which are much more concise and easier to read and understand these built-in models also prevent errors and the construction of unstructured models.

- ***A Sound Cycle***

  A sound cycle, that is free from deadlocks and lack of synchronization, can only be constructed in the following manner (Figure A.22):

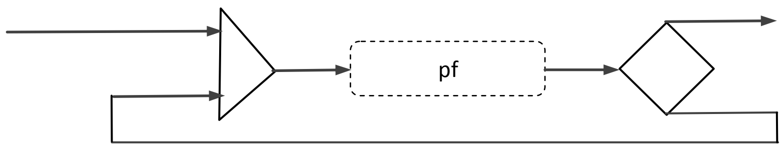


Figure A.22: A structured sound cycle

To define such a cycle, the modeller can use the built-in function `soundCycle` (merge exclusive decision Cycle):

```
soundCycle
    e.g.: soundCycle pf ("Dec 1", (branchProp eYes 0.4, branchProp eNo 0.6))
```

where the output data type of `soundCycle` (i.e. the data type of the first outgoing branch of the decision) is equivalent to the data output type (not necessarily the connection type) of `pf`. `ebr1` and `ebr2` are the expressions associated to the first and the second branches of the decision and must be defined as in conventional decisions.

- ***A Sound Inclusive Decision - Merge***

When a process contains an inclusive decision followed by a merge, then the process would be unsound as it might lack synchronisation. It is not possible to determine whether one or more branches of the decision shall be executed and thus, the activities following the merge, might be evaluated more than once. For this reason, it would be best to explicitly modelling every possible combination of branches by using exclusion decisions and forks. However, these definitions are rather long and tedious for a modeller to define. For instance, a two branch inclusive decision followed by a merge would be modelled in the following manner (refer to Figure A.23):

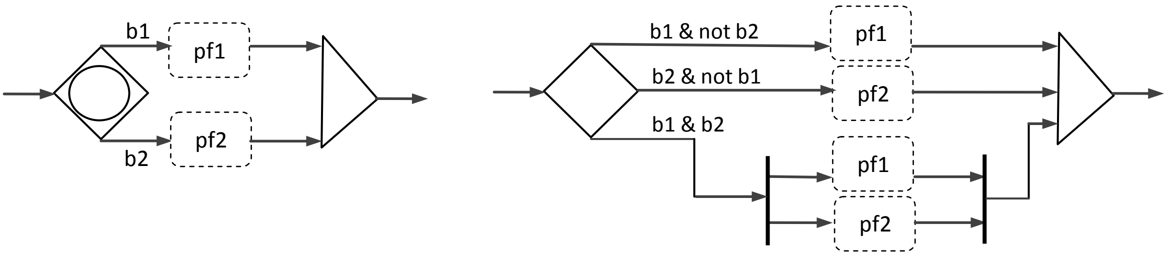


Figure A. 23: The fragment on the right, models the same behaviour as that on the left but in a sound manner. The left model uses an inclusive decision and a merge, whereas the model on the right uses an exclusive decision, a merge, a fork and a join.

These can be defined automatically by using the built-in function, `sound_inclDecision_Merge`

```
sound_inclDecision_Merge

    e.g.:  sound_inclDecision_Merge "dec1"
                              (branchProp e1 0.4, branchProp e2 0.6) (pf1, pf2)

           sound_inclDecision_Merge "dec2" (branchProp e1 0.6, branchProp e2 0.2,
                              branchProp e3 0.2) (pf1, pf2, pf3)
```

The first definition represents a two branch inclusive decision, whereas the second represents a three branch inclusive decision

Similarly, any inclusive decision, having any number of decision branches can be defined using the same function.

- ***Terminating Unnecessary Data Flows on Decision Branches***

Assuming that a decision has three branches and each branch is connected to a process fragment which requires inputs of type `a, b, c` respectively. Thus the input type of the decision should be `(a,b,c)`. All the three data flows would be passed on to all of the outgoing decision branches. However, data flows of type `b` and `c` are not really required by the first branch. Similarly for data flows of type `a` and `c` and `a` and `b`, are not required by the second and third decision branches respectively. For this reason, the unnecessary flows are usually terminated using an `end` node as illustrated below (Figure A.24):

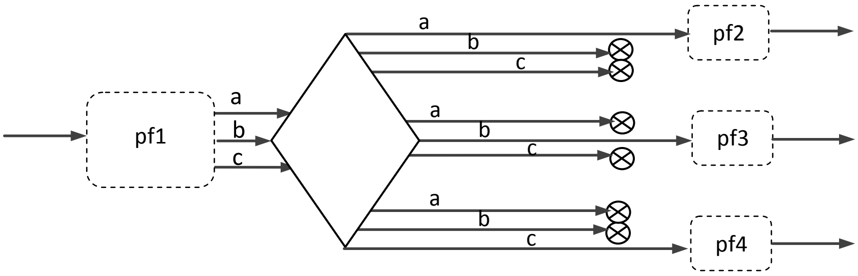


Figure A. 24: An 3-branch exclusive decision with a flow of type (a,b,c)
Only one of the data flows is maintained as soon as the branch is chosen

When such behaviour is required, the following functions can be used:

```
exclDecision_with2AltDataBranches, exclDecision_with3AltDataBranches,
exclDecision_with4AltDataBranches
```

-- *exclusive Decision with 2,3 or 4 Alternative Data Branches*

```
inclDecision_with2AltDataBranches, inclDecision_with3AltDataBranches,
inclDecision_with4AltDataBranches
```

-- *inclusive Decision with 2,3 or 4 Alternative Data Branches*

```
e.g.:  exclDecision_with3AltDataBranches pf1 "Decision 1"
        (branchProp e1 0.3, branchProp e2 0.4, branchProp e3 0.3)(pf2, pf3, pf4)
```

where `pf1` is the process fragment attached to the input of the decision and `e1`, `e2`, `e3` are the expressions associated to every single decision branch (in a similar way as illustrated in Figure A.24)

Similarly, inclusive and exclusive decisions with 2, 3 or 4 decision branches can be defined.

- ***Triangular Models***

To avoid producing cluttered models, modellers are encouraged to use the same merge for multiple decisions in the same process. Similarly, the same join can be used for multiple forks. However, users would have to be careful when the process has both forks and decisions; if a decision is closed with a join or a fork with a merge, then the process would become unsound. Thus the model would either have a deadlock or lack synchronisation. Models, such as the ones in Figure A.25, are sound:

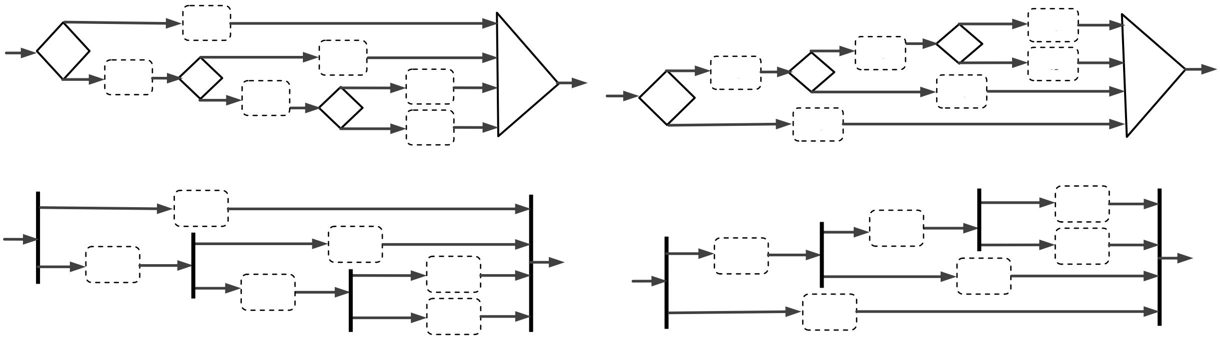


Figure A. 25: Some process fragments having either multiple decisions closed with one merge (as the models in the first row) or multiple forks closed with one join (as the models in the second row). The consecutive decisions/forks can be attached either to the bottom branch of the decisions (as the models in the first column) or to the top branch of the decisions (as the models in the second column)

With our language, the modeller can define similar models containing any number of decisions or forks by using any one of the functions:

`tri_exclDecisions_onLowerBranch_Merge`, `tri_exclDecisions_onUpperBranch_Merge`

-- *triangular model with multiple Decisions and one Merge (decisions on the lower or upper branches)*

`tri_Forks_onLowerBranch_Merge`, `tri_Forks_onUpperBranch_Merge`

-- *triangular model with multiple Forks and one Join (forks on the lower or upper branches)*

```
E.g.:  tri_exclDecisions_onUpperBranch_Merge ["d1","d2","d3"]
                    [bs1, bs2, bs3]  [(pf1a,pf1b),(pf1a,pf2b), (pf3a,pf3b)]
```

where "`d1`", "`d2`", "`d3`" are the names of the three decisions in this triangular model; `bs1`, `bs2`, `bs3` represent the tuples with the properties of the branches for each of the decisions, in the form of `(branchProp e1 0.3, branchProp e2 0.7)`; the third input is a list of tuples containing the process fragments that should be attached to each decision branch. Thus the length of the three input lists must be equivalent. The size of the tuples defined as the third input list must be equivalent to the size of the tuples in the second input list (since every branch of every decision must have an expression and a probability of occurrence, and a process fragment that would be attached to it). If the modeller does not want any process fragment attached to a decision branch, then `noActivity` (with the input data type specified as an input argument) should be used.

The only restriction imposed is that every process fragment and decision must have the same data type for all of its inputs and outputs.

A.7. Ways How to Construct a Model

To create complex models, users must decompose their model into process-fragments. In our language these fragments are essentially functions which given a certain input, a particular output is produced. Thus, even though the user does not want to package that particular process fragment into a sub-process, if the fragment is defined with a suitable name, then it would be easy for the modeller (or reader of the model) to understand the purpose of that fragment without having to understand the internal elements. Thus, it would be easier also for the modeller to reuse these fragments.

Working with functions means that for one component to be linked to another, then either the output of one is passed on as an input to the other or connection patterns (discussed in the section A.5) are used. Most of the time, the modeller is free to choose any of the two construction methods. Models constructed using connection patterns are usually more concise and easier to understand. However, in certain cases it is not always possible to group modelling elements into fragments and combine them using such patterns. For instance, let us assume we want to define the following process fragment using our language (Figure A.26):

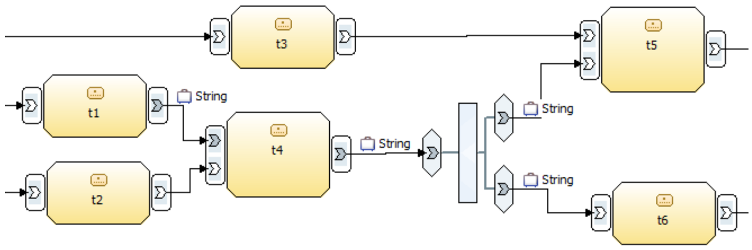


Figure A.26: A process fragment constructed using WSBM

It should be noted that the first part (the fragment that is not shaded in Figure A.27)

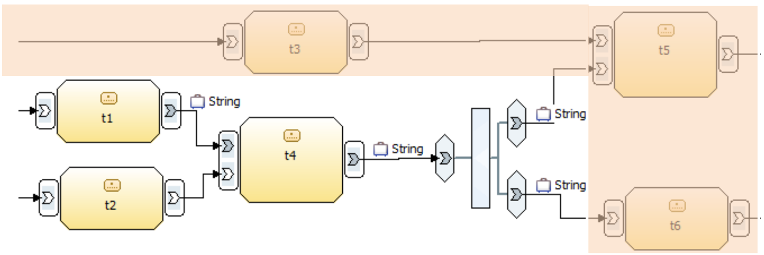


Figure A. 27

can be defined as `(t1-|- t2) ->- t4 -|=`

Thus tasks `t1` and `t2` are first composed in parallel and then the new fragment is composed in sequence to task `t4`. The next component is a fork which is defined by using the operator -|=. However, how shall the other tasks be connected to the outgoing branches of the fork? If we had the following model (Figure A.28)

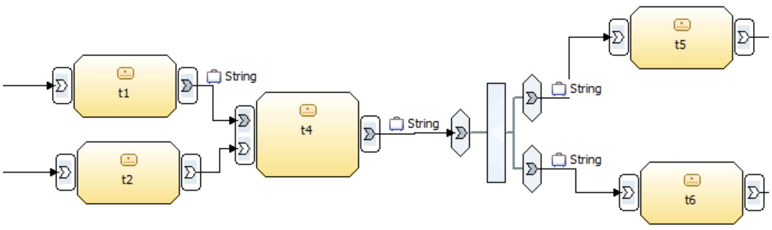


Figure A. 28

It would be easily defined in one line, that is `pf = (t1-|- t2) ->- t4 -|= (t5, t6)`

However, in Figure A.26, task `t5` is connected to the fork and to another task `t3`. Thus we cannot really use the connection pattern -|=. For this reason, it would be best to define it as:

```
pf (x,y,z) = let
                 (of1, of2) = ((t1-|- t2) ->- t4 ->- fork)(x,y)
                 ot5 = t5 (z,of1)
                 ot6 = t6 of2
             in (ot5, ot6)
```

Listing A. 14: Defining model in Figure A.26 in our language

where `x, y, z` are respectively the inputs to tasks `t1, t2, t3` and `ot5`, `ot6` are respectively the outputs of tasks `t5` and `t6` and thus the outputs of process fragment `pf`.

This model illustrates that users are always encouraged to identify all those fragments which can easily be defined using connection patterns. If this is not possible or not really feasible, then the other approach would have to be adopted.

For more examples and sample models, refer to Appendix B. In this appendix all the processes defined in the three sample projects that come along with IBM's WSBM tool are defined using different approaches and constructs in our language.

A.8. Parameterized Models

In our language it is possible for the user to define parameterized models such that depending upon the input argument, the required model is produced. If there are similar fragments which are used within models, then it would be wise to define such parameterized models to construct them by invoking just one function. For example, consider the model in Figure A.29.

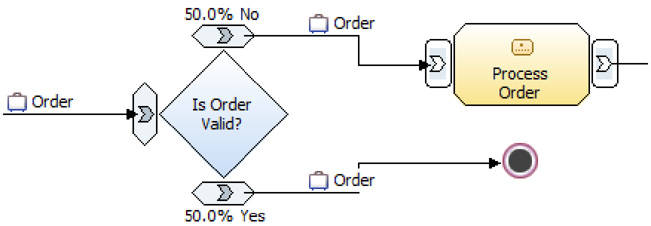


Figure A. 29: A process fragment handling an order

This can be defined as in Listing A.15.

```
pfOrder = (exclDecision "Is Order Valid? "(branchProp eYes 0.5, branchProp eNo 0.5))
           ->- (tProcessOrder -|- stopBranch biTOrder)
```

Listing A.15: Defining in our language the process fragment in Figure A.29

If the a parameterized model such as that in Listing A.16, is defined, then the fragment in Figure A.29 would be defined as illustrated in Listing A.17.

```
exclDec_withLowerStop nm brs typ pf  = (exclDecision nm brs)->-(pf-|-stopBranch typ)
```

Listing A.16: A parametrized model to define models similar to Figure A.29

`nm` is the name of the decision; `brs` is the tuple defining the properties of the decision branches; `typ` is the type of the data flow; `pf` is the process fragment attached to the upper branch

```
pfOrder = exclDec_withStopBranch  "Is Order Valid?"
                                  (branchProp eYes 0.5, branchProp eNo 0.5)
                                   biTOrder
                                   tProcessOrder
```

Listing A.17: Defining the process fragment in Figure A.29 by using the parameterized model in Listing A.16

In this way, fragments, such as Figure A.30, which have a similar structure, can rapidly be defined by simply using one function (as in Listing A.18).

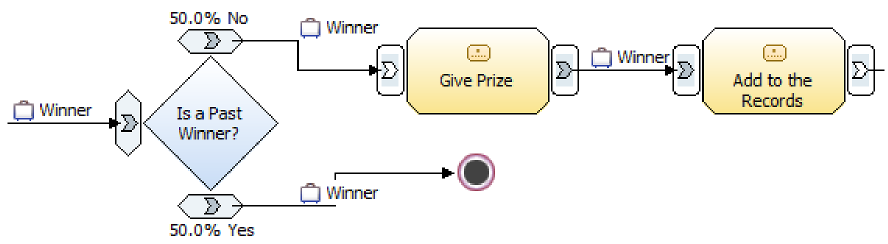


Figure A.30: A process fragment handling decide to how prizes are awared

```
pfWinners = exclDec_withStopBranch "Is a Past Winner? "
                                    (branchProp eNo 0.5, branchProp eYes 0.5)
                                     biTWinner
                                    (tGivePrize ->- tAddToRecords)
```

Listing A.18: Defining the process fragment in Figure A.30 by using the parameterized model in Listing A.16

A.9. Basic Checks

The language provides some in-built basic checks that can be carried out on the model. These are provided in the *two modes*: 1) as a function starting with `show`, to be able to display results on the screen, 2) as a function starting with `get` or `is`, to retrieve result and handle them in scripts

- `isToken, isConstValue, isFlowCon, isTask, isSubProcess, isFork, …`

  These checks are provided for all the modelling elements. It takes a process fragment as input and returns a pair containing a Boolean and details of the process fragment as a string

- `showProcSpec, getProcSpec`

  Get a detailed specification report of the process, including the details of all the elements in the fragment. It takes a process fragment as input and, in the second function, a record with lists for each kind of modelling element. Details from this list can be obtained by invoking functions such as `getTasksFromSpec` *which take the process specification record as input. The details of the each modelling element are defined a '#' separated string.* `getModellingElemType` *(to identify the type of the modelling element that is if Task, Start etc.)* or `getNthPropStr` *(to get the nth attribute in the property string)* or `displayPropertyStr` *(to get a string which can be displayed on the screen to view the properties is user-friendly manner) are provided*

- `showModellingElems, getModellingElems`

  To get just the type of the modelling elements without the properties

- `isNameUsed, showIsNameUsed`

  To check whether a particular name is already used with a process. It takes a process fragment and a name as as input and returns a pair containing a Boolean and details of the element found

- `containsTask, containsSubProcess, …`

  To check whether a particular named modelling element is contained within a process fragment. It takes a name and a process fragment and returns a pair containing a Boolean and details of the element found

- `inputTypeMatch, showInputTypeMatch, outputTypeMatch, showOutputTypeMatch, ioTypeMatch, showIOTypeMatch, …`

  By specifying the required input and/or output type and a process, the details of the elements which match with such types are returned

- `eqPFs`, `showEqPFs`, `eqModellingElems`, `showEqModellingElems`

  Given two process fragments, the two are compared and differences are pointed out and returned. The second two functions just compare the modelling element types and not the properties

Composite complex checks can be defined for pre and post conditions of, example, model transformations. These can be composed in a similar way as functions

A.10. Generating a Directed Graph and Quality Assuring Models

In our language it is possible to generate a directed graph representing the model to facilitate the analysis of the modelling elements when new quality assurance checks are developed. For this reason the following functions are provided:

- `getGraph`

  Given the process and a Boolean to indicate where sub-processes should be handled as a single element or opened up to consider their internal structure, a graph of type `Graph a` *is returned. This Graph is made up of a list of vertices of type Vertex (containing a unique numeric identifier and details of the element as a string), a list of edges (an edge represented as a pair of vertices), a pair containing a list of start and end vertices (in this case the list contains just the number of the vertex)*

- `displayGraphDetailsForPF`

  Similar to get Graph – in this case the details of the generated graph are displayed

- `getNextVerticesOnLeft, getNextVerticesOnRight, getVertexInList, getVerticesInList, …`

  These are some of the provided functions which can be used to handle the graph and carry out operations. While the first two (in the above list) returns the next vertices on the left and right of the element, the last two get either one or more vertices from a list

To check the soundness of model, `inSequence` and `isSound` are provided. The first checks whether elements in the fragment are connected in sequence whereas the second, returns a pair containing a Boolean (to indicate if sound or not) and a string to indicate whether it is a *structured* or *unstructured sequential* or *concurrent branching* fragment

A.11. Creating Composite Model Transformations

Basic primitive model transformations such as `renameRep` and `subRep` are handle different modelling elements and processes. Such transformations can be composed and pre and post conditions can be defined to ensure the quality of the transformed model, as illustrated in the following example:

```
tApplySpecialTerms = task “Apply Special Terms to Order” (biTOrder :-> biTOrder)

transOrderProcessing pf x =

   let
       (hasSPOrderVerif, _) = containsSubProcess “Order Verification” pf
       (hasTaskRejectOrder, _) = containsTask “Reject Order“ pf

        transf1@(wasTransDone, transMsg, transPF)  =

              if (hasSPOrderVerif)
                 then (renameRepQA “Order Verification” “Certify Order” pf x)
                 else if (hasTaskRejectOrder)
                      then (substituteTaskQA “Reject Order“ tApplySpecialTerms [1] pf x)
                      else (Succeeded, “”, pf x)

        transf2 =  renameDecisionQA  “Is Order Valid?“  “Is Order Certified?” pf x

   in transf2
```

Listing A.19: Defining the quality assured composite transformation `transOrderProcessing`

A.12. Conclusion

After reading this tutorial, the reader should be able to understand how to use the language and should be capable of defining simple models. At this stage, the reader should be able to understand the scripts defined for the sample projects that come along with IBM's tool and which are available on the attached CD.

Appendix B

Sample Models and Case Studies

B.1. Introduction

This appendix contains a number of sample models which have been constructed using IBM WebSphere Business Modeler Advanced v6.0.2[1] and defined using various approaches and constructs in our functional modelling language. These samples can be used as case studies to compare and contrast the models defined using IBM's modelling tool and our language. The different constructions defined in our language for the same model can be analysed. The main difference between these constructions is usually the use of specific connection patterns. These sample models are also useful for modellers to learn how to define business processes in our language.

The sample models that are considered in this appendix have been obtained from the sample projects that come along with the tool. These models are very realistic and they were purposely created to help modellers learn how to use IBM's tool. The sample projects include: the *ABC project*, the *External Claims Assessor Management (ECAM) project* and the *Quickstart Finance project.* For each project, a section has been dedicated whereby all the business items, global tasks and repositories and business processes are defined. Other sample processes, which have been intentionally constructed to illustrate specific features of our language are available in the following section.

It should be noted that, the business process *'Customer Order Handling'* defined in *ABC project*, '*Auto Claims Handling'* in *External Claims Assessor Management (ECAM) project* and '*Order Handling'* in the *Other Sample Models* section, have been analysed and used as case studies, in Chapter 6 ('*Evaluation and Case Studies'*), to evaluate this first version of our functional modelling language.

Each project shall be handled in the following manner: the business items, global tasks and repositories are first defined in separate modules for the entire project, and then referenced and used in the definitions of the processes. The process fragments are first constructed using basic modelling elements only (no connection patterns) and then using connection patterns. Once defined, the process fragments are packaged into a sub-process. The scripts, the figures illustrating the models and the IBM WebSphere Business Modeler sample projects are available on the attached CD.

B.2. ABC Project

This is one of the sample projects that come along with IBM WebSphere Business Modeler Advanced v6.0.2[1]. It contains business processes that handle customer orders within a company that sells some product ABC. It has its own user-defined complex types (mainly business items), global tasks and repositories and three processes, as illustrated in Figure B.1.

[1] http://www.ibm.com/developerworks/websphere/zones/businessintegration/roadmaps/modeler/roadmap_advanced.html

Business Items:

- ABC Product
- Customer Order
- Customer Record
- Order Request

Global Repositories:

- Customer Orders
- Customer Records
- Products

Business Processes:

- Customer Order Handling
- Order Verification
- Payment Handling

Figure B. 1: Business Items, Global Repositories and Business Processes defined in *ABC Project*

B.2.1. User-Defined Types - Business Items

The required business items are defined in Listing B.1. Note that module `FuncBPML` must be imported to start using our language.

```
module Types where

   import FuncBPML

   --  Unclassified Types
   --  (i.e. not specified as business item and complex type)

   newtype ABCProd    = ABCProd    ()  deriving Typeable
   newtype CustOrder = CustOrder  ()  deriving Typeable
   newtype CustRec    = CustRec    ()  deriving Typeable
   newtype OrderReq  = OrderReq   ()  deriving Typeable

   -- Type Synonyms for Business Items

   type TABCProd   = TABCProd
   type TCustOrder = TCustOrder
   type TCustRec   = TCustRec
   type TOrderReq  = TOrderReq

   -- Defining Types as Complex

   instance ComplexType (TABCProd)
   instance ComplexType (TCustOrder)
   instance ComplexType (TCustRec)
   instance ComplexType (TOrderReq)

   -- First Class Objects to represent the Type

   biTABCProd   = dType :: TABCProd
   biTCustOrder = dType :: TCustOrder
   biTCustRec   = dType :: TCustRec
   biTOrderReq  = dType :: TOrderReq
```

Listing B.1: ABC project - User-defined Types (Business Items)

B.2.2. Global Repositories

The required global repositories are defined in Listing B.2.

```
rProd       =  rep "Products"          biTABCProd
rCustOrder  =  rep "Customer Orders"   biTCustOrder
rCustRec    =  rep "Customer Records"  biTCustRec
```

Listing B.2: ABC project - Global Repositories

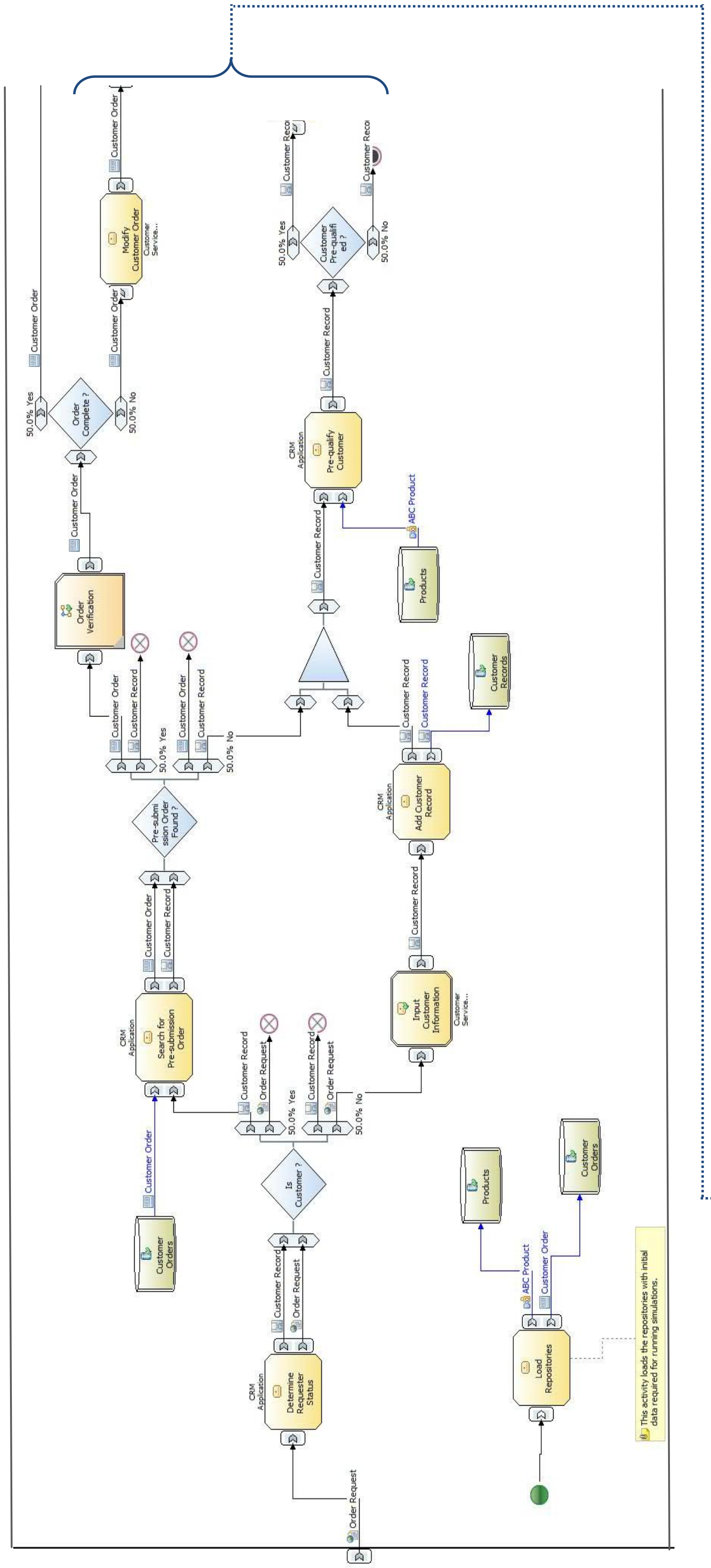
B.2.3. Tasks

The required tasks are defined in Listing B.3 and can be reused by any process in the project.

```
module Tasks where

  import Types

  -- used in Customer Order Handling

  tDetermineReqrStatus = task  "Determine Requester Status"      (biTOrderReq :-> (biTCustRec,biTOrderReq))
  tSearchPreSubmOrder  = task  "Search for Pre-submission Order" ((biTCustOrder,biTCustRec) :-> (biTCustOrder,biTCustRec))
  tInpCustInfo         = task  "Input Customer Information"      (biTOrderReq :-> biTCustRec)
  tAddCustRec          = task  "Add Customer Record"             (biTCustRec :-> (biTCustRec,biTCustRec))
  tPreQualifyCust      = task  "Pre-Qualify Customer"            ((biTCustRec, biTABCProd) :-> biTCustRec)
  tModifyCustOrder     = task  "Modify Customer Order"           (biTCustOrder :-> biTCustOrder)
  tUpdateCustOrder     = task  "Update Customer Order"           (biTCustOrder :-> biTCustOrder)
  tInpCustOrder        = task  "Input Customer Order"            (biTCustRec :-> biTCustOrder)
  tAddCustOrder        = task  "Add Customer Order"              (biTCustOrder :-> (biTCustOrder,biTCustOrder))
  tApproveCustOrder    = task  "Approve Customer Order"          (biTCustOrder :-> biTCustOrder)

  -- used in Order Verification

  tChangeOrderToActiveState   = task  "Change Order to Active State"  (biTCustOrder :-> biTCustOrder)
  tCreditVerfication          = task  "Credit Verification"           (biTCustOrder :-> biTCustOrder)
  tObtainProdAvailability     = task  "Obtain Product Availability"   (biTCustOrder :-> biTCustOrder)
  tConsolidateOrder           = task  "Consolidate Order"             (biTCustOrder :-> biTCustOrder)

  -- used in Payment Handling

  tIdentifyPaymentMethod = task "Identify Payment Method"        (biTCustOrder :-> biTCustOrder)
  tTakeCashOrCheck       = task "Task Cash or Check"             (biTCustOrder :-> biTCustOrder)
  tSwipeCardSign         = task "Swipe Card and Sign"            (biTCustOrder :-> biTCustOrder)
  tPreparePkgForCust     = task "Prepare Package for Customer"   (biTCustOrder :-> biTCustOrder)
```

Listing B.3: ABC project - Tasks

B.2.4. **Business Process 1:** *Customer Order Handling*



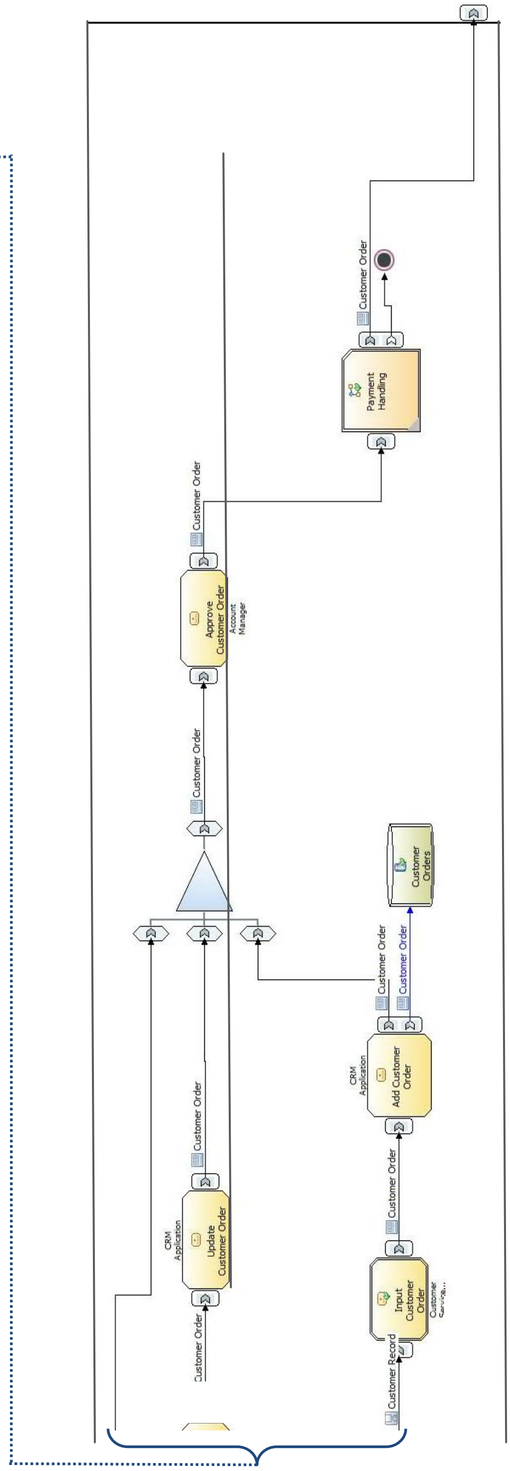


Figure B.2: Business process *'Customer Order Handling' in ABC project*

Figure B. 2

Construction 1: *Constructing the Model using only Basic Modelling Elements*

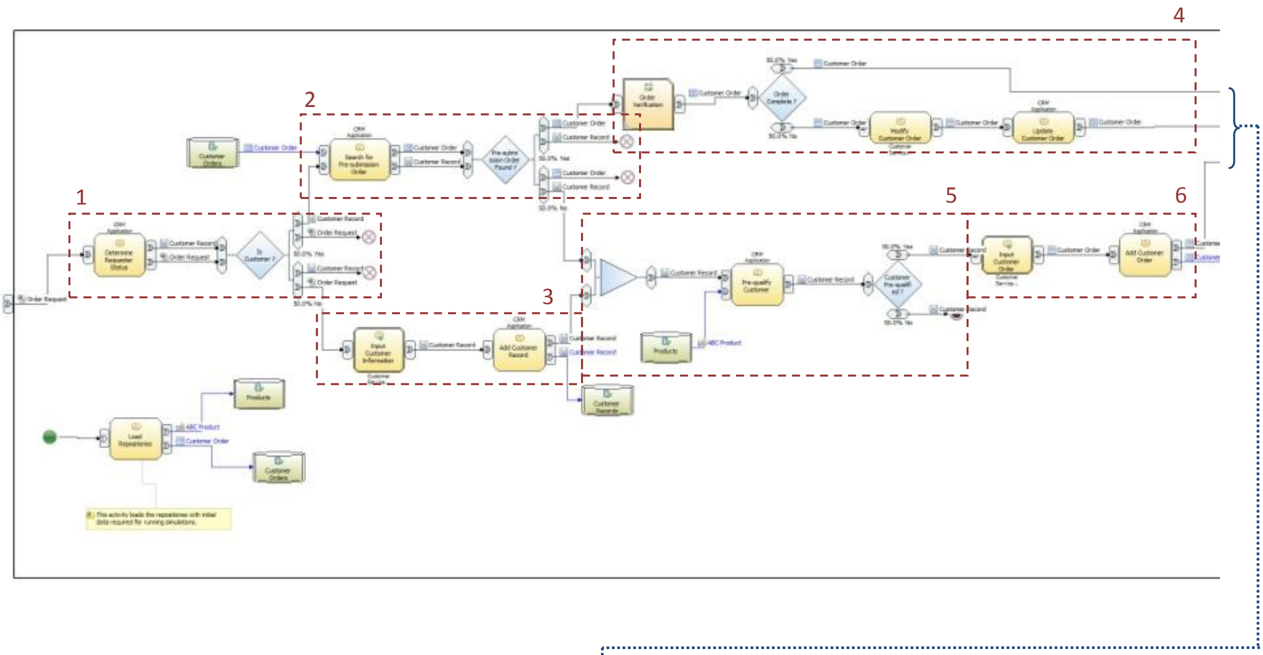
Listing B.4 defines the typed tasks and sub-processes that are used in the process and the expressions that are assigned to the decision branches within the process.

```
- Typed Tasks

ttDetermineReqrStatus    =  flowCI          <|tDetermineReqrStatus|>  (flowCO,flowCO)
ttSearchPreSubmOrder     =  (repCI,flowCI)  <|tSearchPreSubmOrder|>   (flowCO,flowCO)
ttInpCustInfo            =  flowCI          <|tInpCustInfo|>           flowCO
ttAddCustRec             =  flowCI          <|tAddCustRec|>           (flowCO, repCO)
ttPreQualifyCust         =  (flowCI,repCI)  <|tPreQualifyCust|>        flowCO
ttModifyCustOrder        =  flowCI          <|tModifyCustOrder|>       flowCO
ttUpdateCustOrder        =  flowCI          <|tUpdateCustOrder |>      flowCO
ttInpCustOrder           =  flowCI          <|tInpCustOrder|>          flowCO
ttAddCustOrder           =  flowCI          <|tAddCustOrder|>         (flowCO,repCO)
ttApproveCustOrder       =  flowCI          <|tApproveCustOrder|>      flowCO


-- Typed Sub-Processes

tspOrderVerification  =  flowCI  <|spOrderVerification|> flowCO
tspPaymentHandling    =  flowCI  <|spPaymentHandling|>   flowCO


-- Expressions assigned to Decision Branches

eYes_IsCust, eNo_IsCust :: (PF (Flow TCustRec), PF (Flow TOrderReq)) -> Bool
eYes_IsCust = const True
eNo_IsCust  = const True

eYes_OrderFound, eNo_OrderFound :: (PF (Flow TCustOrder), PF (Flow TCustRec)) -> Bool
eYes_OrderFound = const True
eNo_OrderFound  = const True

eYes_OrderComplete, eNo_OrderComplete :: (PF (Flow TCustOrder)) -> Bool
eYes_OrderComplete = const True
eNo_OrderComplete  = const True

eYes_CustPreQual, eNo_CustPreQual :: (PF (Flow TCustRec)) -> Bool
eYes_CustPreQual = const True
eNo_CustPreQual  = const True
```

Listing B. 4: The typed tasks and expressions defined for the first construction of `pfCustOrderHandling1`, defined by using only *Basic Modelling Elements* (no connection patterns)

To be able to handle and reference the different elements in the process, the model is divided into the logical fragments as illustrated in Figure B.3.

The definition of the model is decomposed into the sub-fragments as illustrated in Figure B.3 and is available in Listing B.5a and B.5b.



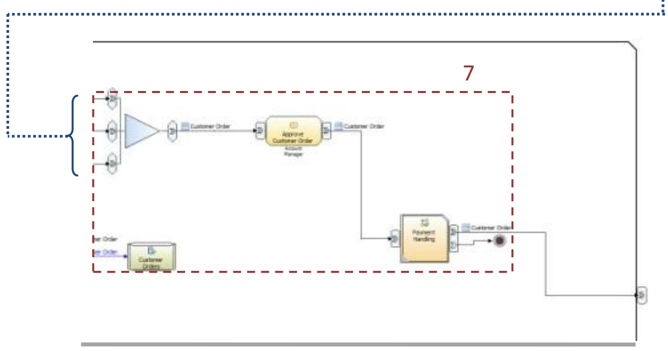


Figure B.7: This figure illustrates how the process has been decomposed into logical fragments while defining the first construction for the model in Figure B.2

Listing B. 5a

```
pfCustOrderHandling (irProd, irCustOrder, x) = let

        -- Passing data to Repositories
        orCustOrder = rCustOrder1 (o2ttAddCustOrder, previousRepContent biTCustOrder irCustOrder)
        orCustRec   = rCustRec1   (o2ttAddCustRec)

        -- Logical Fragment 1
        ((oIsCust_CustRec,oIsCust_OrderReq),(oNotCust_CustRec,oNotCust_OrderReq)) =
                  exclDecision "Is Customer?" (branchProp eYes_IsCust 0.5, branchProp eNo_IsCust 0.5) (ttDetermineReqrStatus x)
        oend_IsCust_OrderReq   = end (oIsCust_OrderReq   :: PF (Flow TOrderReq))
        oend_NotCust_CustRec   = end (oNotCust_CustRec :: PF (Flow TCustRec))

        -- Logical Fragment 2
        ((oOrderFound_CustOrder,oOrderFound_CustRec),(oOrderNotFound_CustOrder,oOrderNotFound_CustRec))=
                  exclDecision "Pre-Submission Order Found?" (branchProp eYes_OrderFound 0.5,branchProp eNo_OrderFound 0.5)
                 (ttSearchPreSubmOrder (orCustOrder,oIsCust_CustRec))
        oend_OrderFound_CustRec      = end (oOrderFound_CustRec      :: PF (Flow TCustRec))
        oend_OrderNotFound_CustOrder = end (oOrderNotFound_CustOrder :: PF (Flow TCustOrder))

        -- Logical Fragment 3
        (o1ttAddCustRec, o2ttAddCustRec) = (ttAddCustRec . ttInpCustInfo) oNotCust_OrderReq

        -- Logical Fragment 4
        (oOrderComplete,oOrderNotComplete)=
                exclDecision "Order Complete?" (branchProp eYes_OrderComplete 0.5, branchProp eNo_OrderComplete 0.5)
               (tspOrderVerification oOrderFound_CustOrder)
         ottUpdateCustOrder = (ttUpdateCustOrder . ttModifyCustOrder)  oOrderNotComplete
```

Listing B.5a: The first part of the required definition to model the process in Figure B.2 using only basic modelling elements. The logical fragments referenced in the definition have been marked on the model in Figure B.3. The typed tasks and expressions assigned to decision branches are defined in Listing B.4. Sub-processes spOerderVerificaiton and spPaymentHandling are other processes in the project which are defined in sections B.2.5 and B.2.6

Listing B.5b

```
 -- Logical Fragment 5
 omerge_CustRec = merge (oOrderNotFound_CustRec :: PF (Flow TCustRec), o1ttAddCustRec)
(oCustPreQual_CustRec,oCustNotPreQual_CustRec)=
            exclDecision "Customer Pre-Qualified?" (branchProp eYes_CustPreQual 0.5, branchProp eNo_CustPreQual 0.5)
           (ttPreQualifyCust (omerge_CustRec,irProd))
 ostop_CustNotPreQual_CustRec  = stop (oCustNotPreQual_CustRec :: PF (Flow TCustRec))

 -- Logical Fragment 6
(o1ttAddCustOrder, o2ttAddCustOrder) = (ttAddCustOrder . ttInpCustOrder) oCustPreQual_CustRec

 -- Sub-Process Fragment 7
 omerge_CustOrder = merge (oOrderComplete :: PF (Flow TCustOrder), ottUpdateCustOrder, o1ttAddCustOrder)
 otspPaymentHandling = (tspPaymentHandling . ttApproveCustOrder) omerge_CustOrder
(opfCustOrderHandling, oCtrl) = (ctrl_dataFlowSplitter otspPaymentHandling) :: (PF (Flow TCustOrder), PF (Flow  Ctrl))
 ostop_Ctrl = stop oCtrl

in

 opfCustOrderHandling, orCustOrder,orCustRec, oend_IsCust_OrderReq, oend_NotCust_CustRec, oend_OrderFound_CustRec,
 oend_OrderNotFound_CustOrder, ostop_CustNotPreQual_CustRec, ostop_Ctrl)
```

Listing B.5a: The second part of the required definition to model the process in Figure B.2 using only basic modelling elements.
The logical fragments referenced in the definition have been marked on the model in Figure B.3. The typed tasks and expressions assigned to decision branches are defined in Listing B.4. Sub-processes spOrderVerificaiton and spPaymentHandling are other processes in the project which are defined in sections B.2.5 and B.2.6

Construction 2: Constructing the Model using Connection Patterns

Listing B.6 defines the expressions that are assigned to the decision branches within the process. Since the lazy serial composition connection pattern is used in the definition in Listing B.7, typed tasks are not required. Instead, the tasks defined in the module `Tasks.hs` are used within the process definition.

```
-- Expressions assigned to Decision Branches

eYes_IsCust, eNo_IsCust :: (PF TCustRec, PF TOrderReq) -> Bool
eYes_IsCust = const True
eNo_IsCust  = const True

eYes_OrderFound, eNo_OrderFound :: (PF TCustOrder, PF TCustRec) -> Bool
eYes_OrderFound = const True
eNo_OrderFound  = const True

eYes_OrderComplete, eNo_OrderComplete :: (PF TCustOrder) -> Bool
eYes_OrderComplete = const True
eNo_OrderComplete  = const True

eYes_CustPreQual, eNo_CustPreQual :: (PF TCustRec) -> Bool
eYes_CustPreQual = const True
eNo_CustPreQual  = const True
```

Listing B. 6: The expressions defined for the second construction of the process, defined using *Connection Patterns*

To be able to handle and reference the different elements in the process, the model is divided into the sub-process fragments as illustrated in Figure B.4. The definition of the model is decomposed into sub-fragments as illustrated in Figure B.4 and is available in Listing B.7.

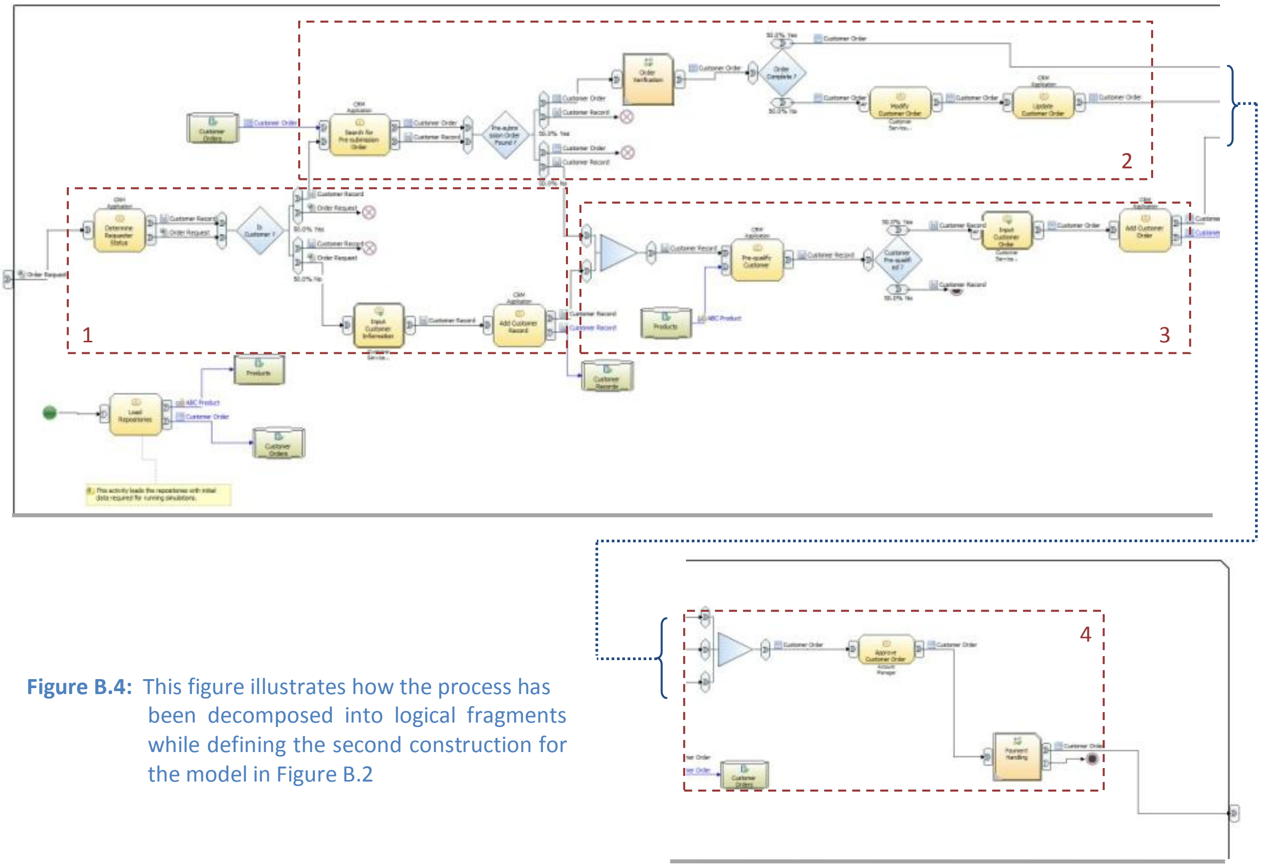


Figure B.4: This figure illustrates how the process has been decomposed into logical fragments while defining the second construction for the model in Figure B.2

```
pfCustOrderHandling (irProd, irCustOrder, x) = let
      -- Passing data to Repositories
      orCustOrder = rCustOrder2 (o2tAddCustOrder, previousRepContent biTCustOrder irCustOrder)
      orCustRec   = rCustRec2   (o2tAddCustRec)

      -- Logical Fragment 1
      spf1 = exclDecision_with2AltDataBranches  tDetermineReqrStatus
                      ("Is Customer?", (branchProp eYes_IsCust 0.5, branchProp eNo_IsCust 0.5))
                      ((noActivity biTCustRec), (tInpCustInfo ->>- tAddCustRec))
      ospf1 @ ( (oIsCust_CustRec,oend_IsCust_OrderReq), (oend_NotCust_CustRec, (o1tAddCustRec, o2tAddCustRec)) ) = spf1 x

      -- Logical Fragment 2
      spf2  =  exclDecision_with2AltDataBranches  ((repCI, noConType) <| tSearchPreSubmOrder)
                      ("Pre-Submission Order Found?", (branchProp eYes_OrderFound 0.5, branchProp eNo_OrderFound 0.5))
                      (dec2UpperBranch, (noActivity biTCustRec))
       dec2UpperBranch = spOrderVerification -<|("Order Complete?", (branchProp eYes_OrderComplete 0.5, branchProp eNo_OrderComplete 0.5) )|>=
                      (noActivity biTCustOrder, (tModifyCustOrder ->>- tUpdateCustOrder))
       ospf2 @ ( ((oOrderComplete_CustOrder, tUpdateCustOrder),end_OrderFound_CustRec),(oend_OrderNotFound_CustOrder,oOrderNotFound_CustRec))
                     = spf2 (orCustOrder, oIsCust_CustRec)

       -- Logical Fragment 3
       spf3  = (merge -|- id) ->- ((flowCI, repCI) <| tPreQualifyCust)
               -<|("Customer Pre-Qualified?",(branchProp eYes_CustPreQual 0.5, branchProp eNo_CustPreQual 0.5))|>=
               ((tInpCustOrder ->>- tAddCustOrder), stopBranch biTCustRec)
       ospf3 @ ((o1tAddCustOrder, o2tAddCustOrder), ostop_CustNotPreQual_CustRec) = spf3 ((oOrderNotFound_CustRec, o1tAddCustRec), irProd)

       -- Logical Fragment 4
       spf4 = merge ->- tApproveCustOrder ->>- spPaymentHandling -//=. stop
       ospf4 @ (ospPaymentHandling, ostop_Ctrl) = spf4 (oOrderComplete_CustOrder, otUpdateCustOrder, o1tAddCustOrder)
 in
      (ospPaymentHandling, orCustOrder,orCustRec, oend_IsCust_OrderReq, oend_NotCust_CustRec, oend_OrderFound_CustRec,
       oend_OrderNotFound_CustOrder, ostop_CustNotPreQual_CustRec, ostop_Ctrl)
```

Listing B.7: The required definition to model the process in Figure B.2 using connection patterns
The logical fragments referenced in the definition have been marked on the model in Figure B.4. The typed tasks and expressions assigned to decision branches are defined in Listing B.6. Sub-processes spOrderVerificaiton and spPaymentHandling are other processes in the project which are defined in sections B.2.5 and B.2.6

Packaging the Process Fragment into a Sub-Process

Once the model is defined as a process fragment, it is packaged into a sub-process, as illustrated in Listing B.8. Any one of the previous definitions can be used.

```
spCustOrderHandling =  packageSubProcess  "Customer Order Handling"  pfCustOrderHandling
```

Listing B.8: The process fragment defined in Listing B.5 or B.7 is packaged into the sub-process `spCustOrderHandling`

B.2.5. **Business Process 2:** *Order Verification*

Figure B.5 illustrates the business process model constructed in IBM WebSphere Business Modeler.

Figure B.5

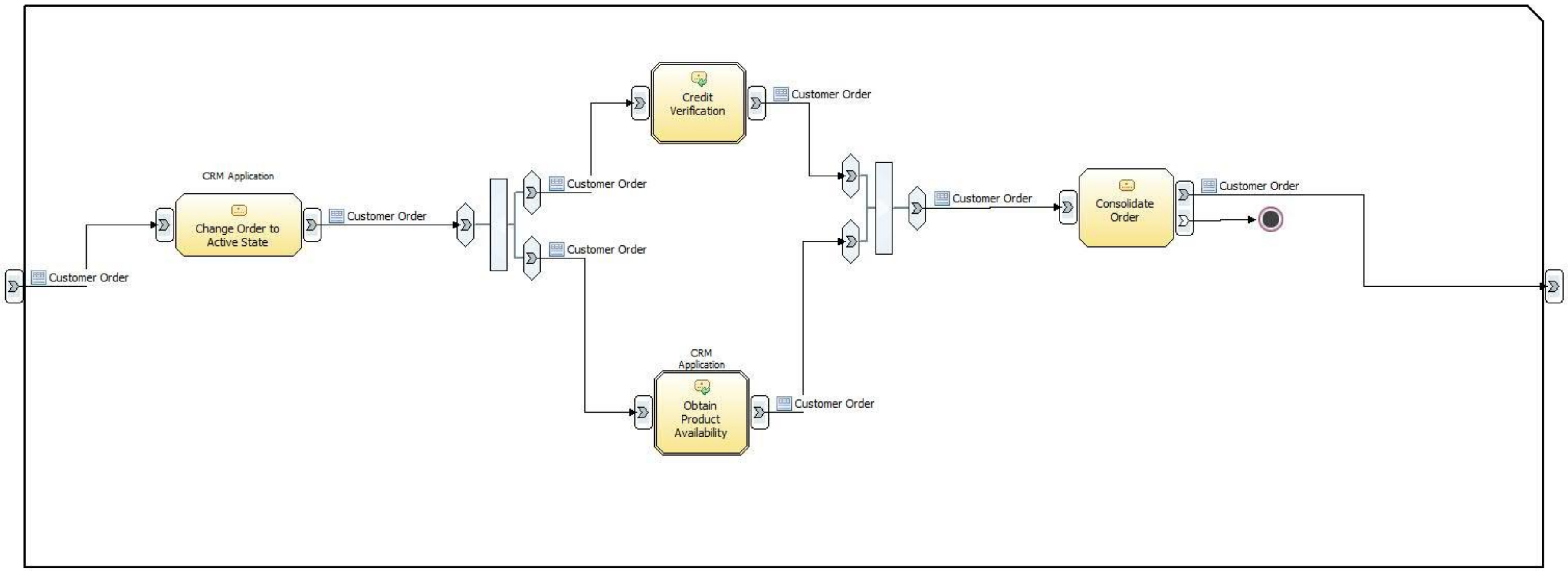


Figure B.5: Business process '*Order Verification*' in *ABC project*

Construction 1: Constructing the Model using only Basic Modelling Elements

The definition of the model using only basic modelling elements is available in Listing B.9. This listing also defines the typed tasks which are required for the definition.

Since in this process, besides tasks, only gateways are used, and since gateways simply allow the data from any compatible connection to flow through, it is possible to use the initial definitions of the tasks (i.e. without the connection types specified). If so, then such untyped versions should be used for all the tasks in the definition. Moreover, the typing used for `ctrl_dataFlowSplitter`, would be set to `::(PF TCustOrder,PF (Flow Ctrl))` rather than `::(PF (Flow TCustOrder),PF (Flow Ctrl))`.

```
-- Typed Tasks

ttChangeOrderToActiveState   =  flowCI  <|tChangeOrderToActiveState|> flowCO
ttCreditVerfication          =  flowCI  <|tCreditVerfication|>        flowCO
ttObtainProdAvailability     =  flowCI  <|tObtainProdAvailability|>   flowCO
ttConsolidateOrder           =  flowCI  <|tConsolidateOrder |>        flowCO


-- Process Fragment

pfOrderVerification x = let

       -- Task before fork
       ottChangeOrderToActiveState = ttChangeOrderToActiveState x

       -- Fork
       (o1fork,o2fork) = fork ottChangeOrderToActiveState

       -- Tasks attached to outgoing branches of fork
       ottCreditVerfication = ttCreditVerfication o1fork
       ottObtainProdAvailability = ttObtainProdAvailability o2fork

       -- Join
       ojoin = join (ottCreditVerfication,ottObtainProdAvailability)

       -- Task after join
       ottConsolidateOrder = ttConsolidateOrder ojoin

       -- Data flow Splitter & Stop Node
       (pfOrderVerification, oCtrl) = (ctrl_dataFlowSplitter ottConsolidateOrder)
                                   :: (PF (Flow TCustOrder), PF (Flow  Ctrl))
        ostop_Ctrl = stop oCtrl

    in (pfOrderVerification, ostop_Ctrl)
```

Listing B.5: Process fragment `pfOrderVerification` defined by using *only Basic Modelling Elements* (no connection patterns). The typed tasks used in this definition are defined above the process definition

Construction 2: *Constructing the Model using Connection Patterns*

Listing B.10 and B.11 illustrate two different ways how the models can be defined in our language. The difference between the two is the type of connection patterns that are used.

```
-- using connection pattern fork_join, -//=. & initially defined tasks (not typed)

pfOrderVerification  =  tChangeOrderToActiveState ->-
                         (fork_join (tCreditVerfication,tObtainProdAvailability)) ->-
                          tConsolidateOrder -//=. stop
```

Listing B.10: Process fragment `pfOrderVerification`, defined by using connection pattern `fork_join`

```
-- using connection patterns -|= & =|-, -//=. & initially defined tasks (not typed)

pfOrderVerification = tChangeOrderToActiveState -|=
                      (tCreditVerfication,tObtainProdAvailability) =|-
                       tConsolidateOrder -//=. stop
```

Listing B.11: Process fragment `pfOrderVerification`, defined by using connection pattern – | = (for fork) and =|- (for join)

Note that different from *Construction 1*, instead of using a `ctrl_dataFlowSplitter` and typing its output before attaching it to a stop node, the connection pattern `-//=.` is very conveniently used. As indicated with the operator, the flow is split and the second branch is terminated (that is why `-//=` with a `.`) with the succeeding event node (in this case, a `stop` node).

Packaging the Process Fragment into a Sub-Process

Once the model is defined as a process fragment, it is packaged into a sub-process, as illustrated in Listing B.12. Any one of the previous definitions can be used.

```
spOrderVerification =  packageSubProcess  "Order Verification"  pfOrderVerification
```

Listing B.12: The process fragment defined in Listing B.9, B.10 or B.11 is packaged into the sub-process `spOrderVerification`

B.2.6. **Business Process 3:** *Payment Handling*

Figure B.6 illustrates the business process model constructed in IBM WebSphere Business Modeler. Note that this process is similar to the '`Order Verification`' process. The only difference is that, instead of a `fork` and a `join`, an `exclusive decision` and a `merge` are used.

Construction 1: *Constructing the Model using only Basic Modelling Elements*

The definition of the model using only basic modelling elements is available in Listing B.13. This listing also defines the typed tasks which are required for the definition and the expressions that are assigned to the decision branches.

Similar to the '`Order Verification`' process, since in this process, besides tasks, only gateways are used, and since gateways simply allow the data from any compatible connection to flow through, it is possible to use the initial definitions of the tasks (i.e. without the connection types specified). If so, then such untyped versions should be used for all the tasks in the definition. Moreover, the typing used for `ctrl_dataFlowSplitter`, would be set to `::(PF TCustOrder,PF (Flow Ctrl))` rather than `::(PF (Flow TCustOrder),PF (Flow Ctrl))`.

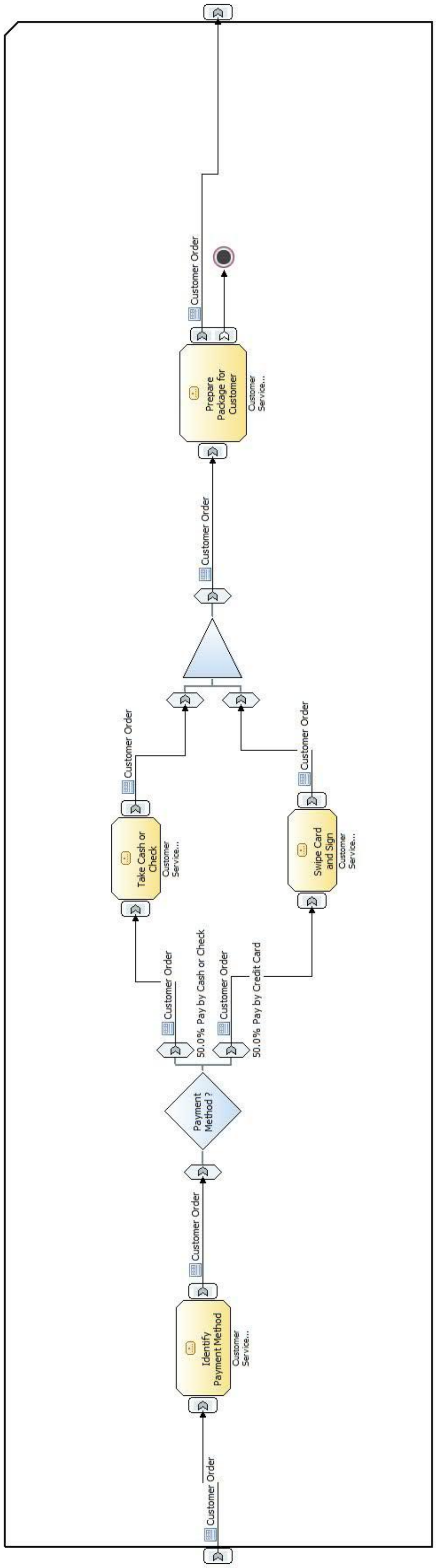


Figure B.6: Business process '*Payment Handling*' in *ABC project*

Figure B.6

```
-- Typed Tasks

ttIdentifyPaymentMethod   =  flowCI  <|tIdentifyPaymentMethod|>  flowCO
ttTakeCashOrCheck         =  flowCI  <|tTakeCashOrCheck|>        flowCO
ttSwipeCardSign           =  flowCI  <|tSwipeCardSign|>          flowCO
ttPreparePkgForCust       =  flowCI  <|tPreparePkgForCust|>      flowCO


-- Expressions assigned to Decision Branches

eCashCheckFlow, eCreditFlow  :: (PF (Flow TCustOrder)) -> Bool
eCashCheckFlow = const True
eCreditFlow    = const True


-- Process Fragment

pfPaymentHandling x = let
                     -- Task before decision
                     ottIdentifyPaymentMethod = ttIdentifyPaymentMethod x

                     -- Exclusive Decision
                     (odecCashCheck,odecCredit) = exclDecision "Payment Method?"
                              (branchProp eCashCheckFlow 0.5,branchProp eCreditFlow 0.5)
                              ottIdentifyPaymentMethod

                     -- Tasks attached to decision branches
                     ottTakeCashOrCheck = ttTakeCashOrCheck odecCashCheck
                     ottSwipeCardSign   = ttSwipeCardSign   odecCredit

                     -- Merge
                     omerge = merge (ottTakeCashOrCheck, ottSwipeCardSign)

                     -- Task after decision
                     ottPreparePkgForCust = ttPreparePkgForCust omerge

                     -- Data flow Splitter & Stop Node
                     (opfPaymentHandling,oCtrl)=(ctrl_dataFlowSplitter ottPreparePkgForCust)
                                         ::(PF (Flow TCustOrder), PF (Flow  Ctrl))
                     ostop_Ctrl = stop oCtrl

                 in  (opfPaymentHandling, ostop_Ctrl)
```

Listing B.13: The process fragment, `pfPaymentHandling`, defined by using *only Basic Modelling Elements*. The typed tasks used in this definition and the expressions assigned to the decision branaches are defined above the process definition

Construction 2: *Constructing the Model using Connection Patterns*

Listing B.14 and B.15 illustrate two different ways how the models can be defined in our language. The difference between the two is the type of connection patterns that are used.

```
-- using connection pattern exclDecision_merge, -//=. & initially defined tasks (not typed)

pfPaymentHandling = tIdentifyPaymentMethod ->- (exclDecision_merge "Payment Method?"
                    (branchProp eCashCheck 0.5,branchProp eCredit 0.5)
                    (tTakeCashOrCheck,tSwipeCardSign)) ->- tPreparePkgForCust -//=. stop
```

Listing B.14: Process fragment `pfPaymentHandling`, defined by using connection pattern `exclDecision_merge`

```
-- using connection patterns -<|..|>= & =>-, -//=. & initially defined tasks (not typed)

pfPaymentHandling = tIdentifyPaymentMethod
                -<|("Payment Method?", (branchProp eCashCheck 0.5,branchProp eCredit 0.5))|>=
                (tTakeCashOrCheck,tSwipeCardSign) =>- tPreparePkgForCust -//=. stop
```

Listing B.15: Process fragment `pfPaymentHandling`, defined by using connection pattern `−<|..|>=` (for exclusive decision) and `=>−` (for merge)

Packaging the Process Fragment into a Sub-Process

Once the model is defined as a process fragment, it is packaged into a sub-process, as illustrated in Listing B.16. Any one of the previous definitions can be used.

```
spPaymentHandling =  packageSubProcess  "Payment Handling"  pfPaymentHandling
```

Listing B.16: The process fragment defined in Listing B.13, B.14 or B.15 is packaged into the sub-process `spPaymentHandling`

B.3. External Claims Assessor Management (ECAM) Project

This is one of the sample projects that come along with IBM WebSphere Business Modeler Advanced v6.0.2[1]. As the name suggests, this project contains business processes of an automobile insurance company. It has its own user-defined complex types (mainly business items), global tasks and repositories and four processes, as illustrated in Figure B.7.

Business Items:

- Assessment Report
- Assessment Request
- Assessor List
- Assessor QOS History
- Assessor Record
- Assessor Selection Rule
- Auto Claim
- Notification of Loss
- Policy

Global Repositories:

- Assessor Records
- Assessor Selection Rules
- Assessors QOS History
- Auto Claims
- Policies

Business Processes:

- Assessor Determination
- Auto Claims Handling
- Auto Claims Process
- Auto Claims Submission

Figure B.7: Business Items, Global Repositories and Business Processes defined in *ECAM Project*

B.3.1. User-Defined Types - Business Items

The required business items are defined in Listing B.17.

[1] http://www.ibm.com/developerworks/websphere/zones/businessintegration/roadmaps/modeler/roadmap_advanced.html

```
module Types where
   import FuncBPML

   --  Unclassified Types
   --  (i.e. not specified as business item and complex type)

   newtype AssRep      = AssRep ()     deriving Typeable
   newtype AssReq      = AssReq ()     deriving Typeable
   newtype AssList     = AssList ()    deriving Typeable
   newtype AssQOSHist  = AssQOSHist () deriving Typeable
   newtype AssRec      = AssRec ()     deriving Typeable
   newtype AssSelRec   = AssSelRec ()  deriving Typeable
   newtype AutoClaim   = AutoClaim ()  deriving Typeable
   newtype NotifLoss   = NotifLoss ()  deriving Typeable
   newtype Policy      = Policy ()     deriving Typeable

   -- Type Synonyms for Business Items

   type TAssRep      = BI AssRep
   type TAssReq      = BI AssReq
   type TAssList     = BI AssList
   type TAssQOSHist  = BI AssQOSHist
   type TAssRec       = BI AssRec
   type TAssSelRec   = BI AssSelRec
   type TAutoClaim   = BI AutoClaim
   type TNotifLoss   = BI NotifLoss
   type TPolicy      = BI Policy

   -- Defining Types as Complex

   instance ComplexType (TAssRep)
   instance ComplexType (TAssReq)
   instance ComplexType (TAssList)
   instance ComplexType (TAssQOSHist)
   instance ComplexType (TAssRec)
   instance ComplexType (TAssSelRec)
   instance ComplexType (TAutoClaim)
   instance ComplexType (TNotifLoss)
   instance ComplexType (TPolicy)

   -- First Class Objects to represent the Type

   biTAssRep       = dType  :: TAssRep
   biTAssReq       = dType  :: TAssReq
   biTAssList      = dType  :: TAssList
   biTAssQOSHist   = dType  :: TAssQOSHist
   biTAssRec       = dType  :: TAssRec
   biTAssSelRec    = dType  :: TAssSelRec
   biTAutoClaim    = dType  :: TAutoClaim
   biTNotifLoss    = dType  :: TNotifLoss
   biTPolicy       = dType  :: TPolicy
```

Listing B.17: ECAM project - User-defined Types (Business Items)

B.3.2. Global Repositories

The required global repositories are defined in Listing B.18.

```
rAssRec      =  repository  "Assessor Records"          biTAssRec
rAssSel      =  repository  "Assessor Selection Rules"  biTAssSelRec
rAssQOSHist  =  repository  "Assessor QOS History"      biTAssQOSHist
rAutoClaims  =  repository  "Auto Claims"               biTAutoClaim
rPolicies    =  repository  "Policies"                  biTPolicy
```

Listing B.18: ECAM project - Global Repositories

B.3.3. Tasks

The required tasks are defined in Listing B.19 and can be reused by any process in the project.

```
module Tasks where

   import Types

   -- used in Assessor Determination

   tIdentifySuitableAssr   = task "Identify Suitable Assessors"   ((biTAssReq,   biTAssRec)    :-> (biTAssList, biTAssSelRec))
   tIdentifyAvailableAssr  = task "Identify Available Assessors"  ((biTAssList, biTAssSelRec) :-> (biTAssList, biTAssSelRec))
   tSelectAssr             = task "Select Assessor"      ((biTAssList, biTAssSelRec, biTAssSelRec, biTAssQOSHist) :-> biTAssReq)

   -- used in Auto Claims Handling

   tAssignNewAC               = task "Assign New Auto Claim"          (biTAutoClaim :->biTAutoClaim)
   tVerifyAC                           = task "Verify Auto Claim"               ((biTPolicy, biTAutoClaim) :->biTAutoClaim)
   tInvestigateAC                 = task "Investigate Auto Claim"         (biTAutoClaim :-> biTAutoClaim)
   tReqAutoAss                    = task "Request Auto Assessment"            (biTAutoClaim :-> (biTAutoClaim, biTAssReq))
   tIdentifyClaimHotSpots     = task "Identify Claim Hot Spots"       (biTAssReq :->biTAssReq)
   tReqAssFollowUp            = task "Request Assessment Follow-Up"   (biTAssReq :->biTAssReq)
   tClaimAssr                     = task "Claim Assessor"                     ((biTAssReq,biTAssReq) :-> biTAssRep)
   tCreateAssRep              = task "Create AssessmentReport"        ((biTAutoClaim,biTAssRep) :-> biTAutoClaim)
   tNegotiateSettlement       = task "Negotiate Settlement"           (biTAutoClaim :->biTAutoClaim)
   tReceiveRegisterAcceptance     = task "Receive and Register Acceptance"(biTAutoClaim :->biTAutoClaim)
   tInitiatePayAndRepair          = task "Initiate Payment and Repair"    (biTAutoClaim :-> biTAutoClaim)
   tCloseAC                       = task "Close Auto Claim"                   (biTAutoClaim :-> biTAutoClaim)

   -- used in Auto Claims Process
   --      none -> uses other processes

   -- used in Auto Claims Submission

   tCreateNewAC      = task "Create New Auto Claim"   (biTNotifLoss :-> biTAutoClaim)
   tVerifyAutoPolicy = task "Verify Auto Policy"      ((biTAutoClaim, biTPolicy) :-> biTAutoClaim)
   tRegisterNewAC    = task "Register New Auto Claim" (biTAutoClaim :-> biTAutoClaim)
   tCancelNewAC      = task "Cancel New Auto Claim"   (biTAutoClaim :-> noData)
```

Listing B. 19: ECAM project - Tasks

B.3.4. Business Process 1: *Assessor Determination*

Figure B.8 illustrates the business process model constructed in IBM WebSphere Business Modeler.

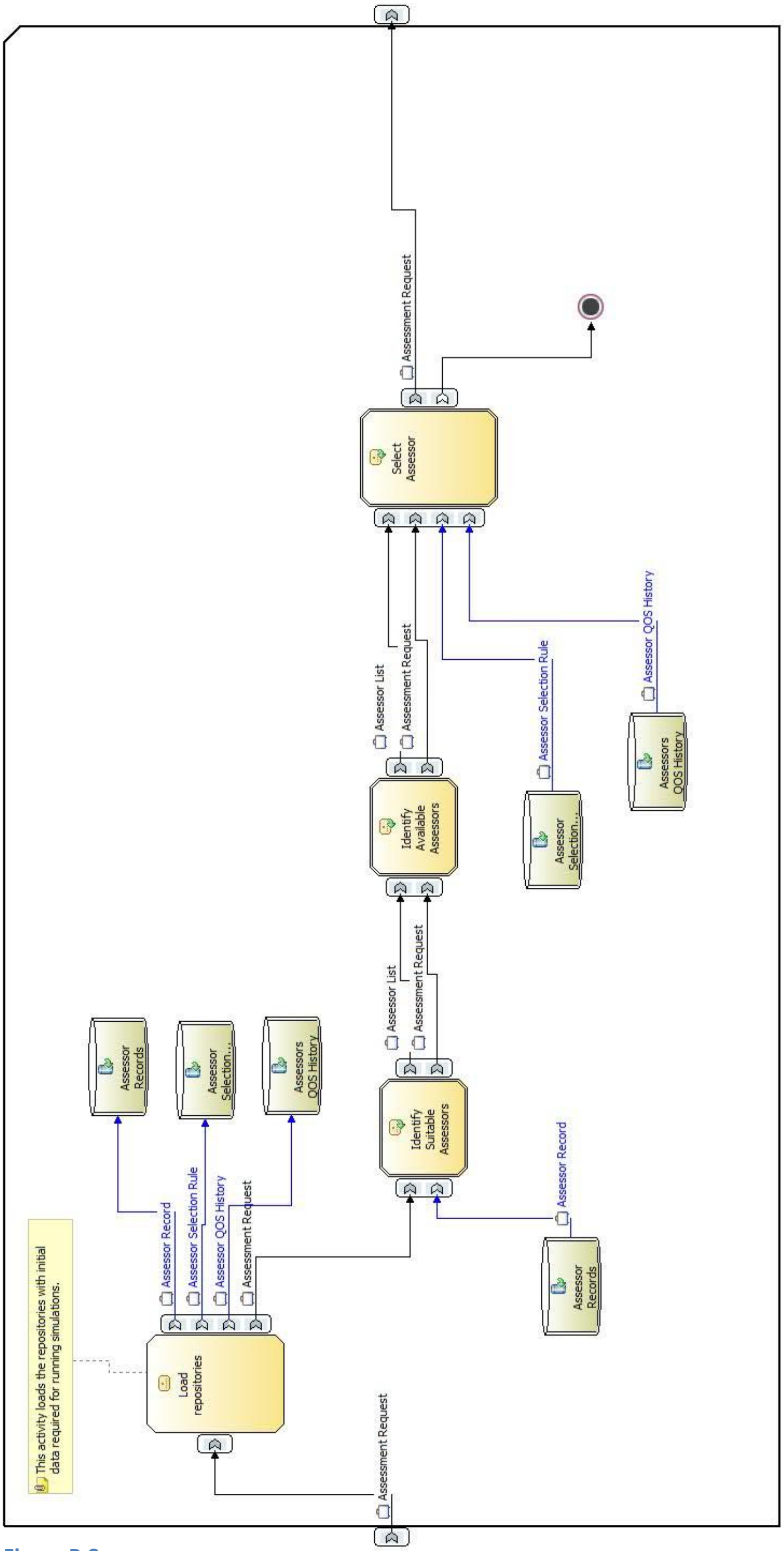


Figure B.8

Figure B.8: Business process *'Assessor Determination'* in *ECAM project*

Construction 1: *Constructing the Model using only Basic Modelling Elements*

The definition of the model using only basic modelling elements is available in Listing B.20. This listing also defines the typed tasks which are required for the definition. The global repositories are not defined updated within this process and thus it is assumed that they are filled up by others and their content is passed on as input arguments to this process. These input arguments are `irAssRec`, `irAssSel`, `irAssQOSHist`.

```
-- Typed Tasks

ttIdentifySuitableAssr   =  (flowCI, repCI)   <| tIdentifySuitableAssr |>   (flowCO, flowCO)
ttIdentifyAvailableAssr  =  (flowCI, flowCI)  <| tIdentifyAvailableAssr |>  (flowCO, flowCO)
ttSelectAssr             =  (flowCI, flowCI, repCI, repCI)  <| tSelectAssr |>  flowCO

-- Process Fragment

pfAssessorDetermination (irAssRec,irAssSel,irAssQOSHist,x) = let

    -- Tasks in sequence
   (otIdentifySuitableAssr_AssList,otIdentifySuitableAssr_AssReq) =
                                                        ttIdentifySuitableAssr (x, irAssRec)
   (otIdentifyAvailableAssr_AssList,otIdentifyAvailableAssr_AssReq) =
                              ttIdentifyAvailableAssr
                              (otIdentifySuitableAssr_AssList,otIdentifySuitableAssr_AssReq)
    ottSelectAssr = ttSelectAssr (otIdentifyAvailableAssr_AssList,
                              otIdentifyAvailableAssr_AssReq, irAssSel, irAssQOSHist)

    -- Data flow Splitter & Stop Node
   (opfAssessorDetermination, oCtrl) = (ctrl_dataFlowSplitter ottSelectAssr)
                                                     :: (PF (Flow TAssReq), PF (Flow  Ctrl))
    ostop_Ctrl = stop oCtrl

 in (opfAssessorDetermination, ostop_Ctrl)
```

Listing B. 20: The process fragment, `pfAssessorDetermination`, defined by using *only Basic Modelling Elements* The typed tasks used in this definition are defined above the process definition

Construction 2: *Constructing the Model using Connection Patterns*

Since different tasks require as input data from different repositories, it is not possible to define the models as one single function. Instead, connections are partly defined using connection patterns and partly defined by passing output arguments as input to other functions (see Listing B.21).

```
pfAssessorDetermination (irAssRec,irAssSel,irAssQOSHist,x) = let

     -- First 2 Tasks
    (otIdentifyAvailableAssr_AssList, otIdentifyAvailableAssr_AssReq) =
                         (tIdentifySuitableAssr ->>- tIdentifyAvailableAssr) (x,irAssRec)

     -- Third Task with Stop
     (opfAssessorDetermination, ostop_Ctrl) =  (tSelectAssr -//=. stop)
                         (otIdentifyAvailableAssr_AssList, otIdentifyAvailableAssr_AssReq,
                         irAssSel, irAssQOSHist)

in  (opfAssessorDetermination, ostop_Ctrl)
```

Listing B.21: Process fragment `pfAssessorDetermination`, defined by passing arguments & using connection patterns

Packaging the Process Fragment into a Sub-Process

Once the model is defined as a process fragment, it is packaged into a sub-process, as illustrated in Listing B.22. Any one of the previous definitions can be used.

```
spAssessorDetermination= packageSubProcess "Assessor Determination" pfAssessorDetermination
```

Listing B.22: The process fragment defined in Listing B.20 or B.21 is packaged into sub-process `spAssessorDetermination`

B.3.5. Business Process 2: *Auto Claims Handling*

Figure B.9 illustrates the business process model constructed in IBM WebSphere Business Modeler.

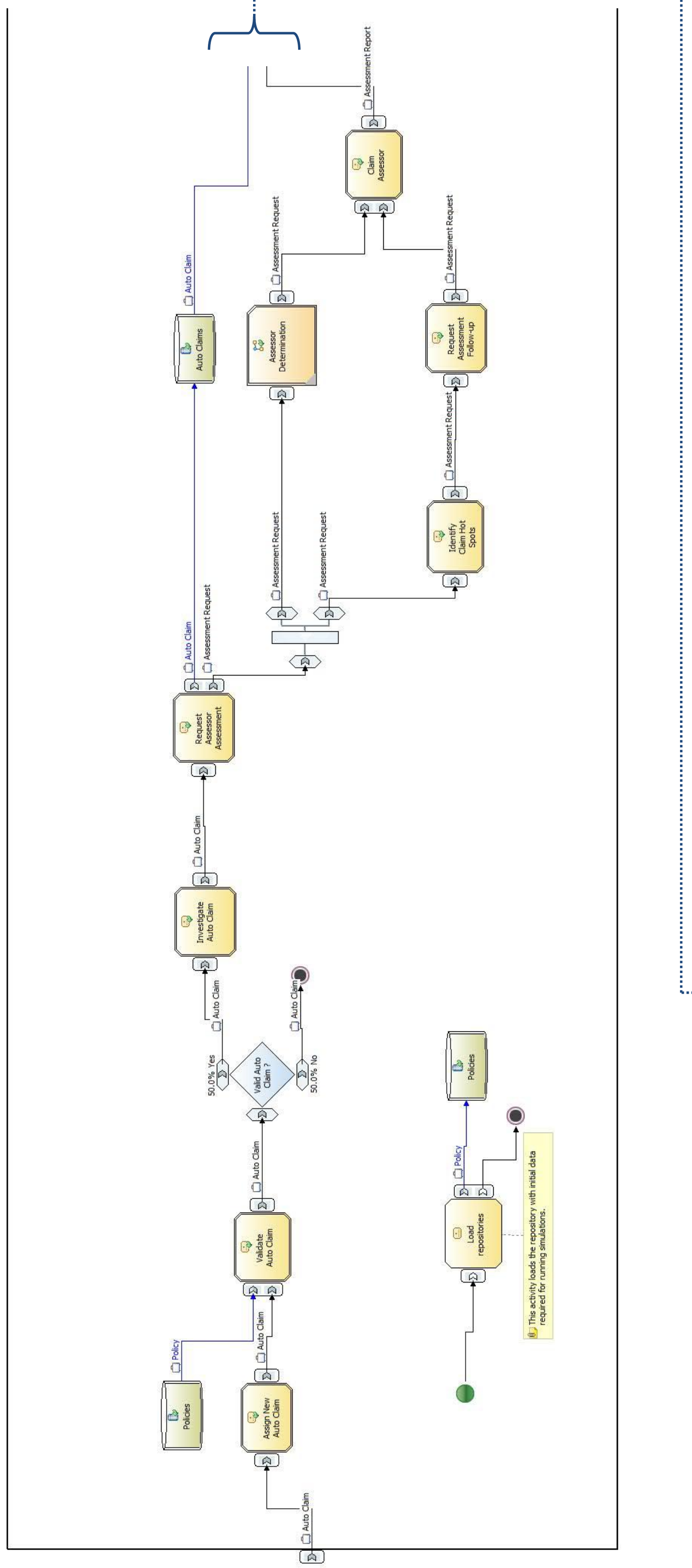
Construction 1: Constructing the Model using only Basic Modelling Elements

Listing B.23 defines the typed tasks and sub-process that are used in the process and the expressions that are assigned to the decision branches within the process.

```
-- Typed Tasks

ttAssignNewAC           = flowCI          <| tAssignNewAC |>                flowCO
ttVerifyAC              = (repCI, flowCI)  <| tVerifyAC |>                  flowCO
ttInvestigateAC         = flowCI          <| tInvestigateAC |>              flowCO
ttReqAutoAss            = flowCI          <| tReqAutoAss |>          (repCO,flowCO)
ttIdentifyClaimHotSpots = flowCI          <| tIdentifyClaimHotSpots |>      flowCO
ttReqAssFollowUp        = flowCI          <| tReqAssFollowUp |>             flowCO
ttClaimAssr             = (flowCI, flowCI) <| tClaimAssr |>                 flowCO
ttCreateAssRep          = (repCI,flowCI)   <| tCreateAssRep |>              flowCO
ttNegotiateSettlement   = flowCI          <| tNegotiateSettlement |>        flowCO
ttReceiveRegisterAcceptance = flowCI      <| tReceiveRegisterAcceptance |>  flowCO
ttInitiatePayAndRepair      = flowCI      <| tInitiatePayAndRepair |>       flowCO
ttCloseAC                   = flowCI      <| tCloseAC |>                    flowCO


-- Expressions assigned to Decision Branches

eYes_IsValidAC, eNo_IsValidAC :: (PF (Flow TAutoClaim)) -> Bool
eYes_IsValidAC = const True
eNo_IsValidAC  = const True
```

Listing B.23: The typed tasks and sub-process and expressions defined for the first construction of `pfAutoClaimsHandling`, defined by using only *Basic Modelling Elements Only* (no connection patterns)

The process fragment is defined in Listing B.24



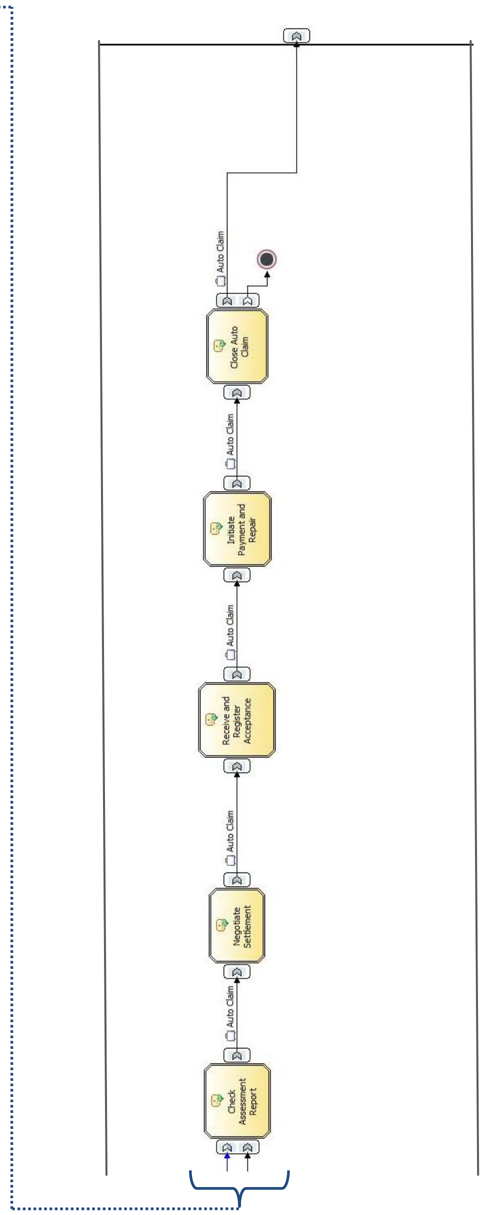


Figure B.9

Figure B.9: Business process *'Auto Claims Handling'* in *External Claims Assessor Management (ECAM) project*

Listing B. 24

```
pfAutoClaimsHandling (irAssRec,irAssSel,irAssQOSHist,irPolicy,x) =  let
                -- First 2 tasks
                ottAssignNewAC = ttAssignNewAC x
                ottVerifyAC  = ttVerifyAC (irPolicy, ottAssignNewAC)

                -- Decision with 2 sequential tasks and a stop
                (oValidAC, oNotValidAC) = exclDecision "Valid Auto Claim?"
                                          (branchProp eYes_FlowIsValidAC 0.5,branchProp eNo_FlowIsValidAC 0.5) ottVerify
                ostop_NotValidAC_AC = stop (oNotValidAC :: PF (Flow TAutoClaim))
                ottReqAutoAss @ (ottReqAutoAss_AutoClaim, ottReqAutoAss_AssReq)  = (ttReqAutoAss . ttInvestigateAC) oValidAC

                -- Auto Claims repository fill up
                orAutoClaim = rAutoClaims ottReqAutoAss_AutoClaim

                -- Fork fragment
                (o1fork, o2fork) = fork ottReqAutoAss_AssReq
                ospAssessorDetermination = spAssessorDetermination (irAssRec,irAssSel,irAssQOSHist,o1fork)
                ottReqAssFollowUp = (ttReqAssFollowUp . ttIdentifyClaimHotSpots) o2fork
                ottClaimAssr = ttClaimAssr (infConType ospAssessorDetermination, ottReqAssFollowUp)

                -- Final 4 sequential tasks
                ottCloseAC = (ttCloseAC . ttInitiatePayAndRepair . ttReceiveRegisterAcceptance . ttNegotiateSettlement .
                              ttCreateAssRep)(orAutoClaim,ottClaimAssr)

                -- Stop attached to final task
                (opfAutoClaimsHandling, oCtrl) = (ctrl_dataFlowSplitter ottCloseAC)::(PF(Flow TAutoClaim),PF (Flow Ctrl))
                ostop_ctrl = stop oCtrl

           in  (opfAutoClaimsHandling, orAutoClaim, ostop_NotValidAC_AC, ostop_ctrl)
```

Listing B.24: The process fragment, `pfAutoClaimsHandling`, defined by using *only Basic Modelling Elements* (no connection patterns). The sub fragments are illustrated in Figure B.9. The actual process constructed in IBM's tool is in Figure B.9. The typed tasks and expressions assigned to decision branches are defined in Listing B.23. The sub-process spAD is another process in the project which is defined in section B.3.4

Construction 2: *Constructing the Model using Connection Patterns*

The expressions that are assigned to the decision branches within the process are defined in Listing B.25. The actual process is defined in Listing B.26. Since the lazy serial composition connection pattern is used in the definition in Listing B.26, typed tasks are not required. Instead, the tasks defined in the module `Tasks.hs` are used within the process definition. Instead, to ensure that the task '`Verify Auto Claim`' gets its first input from the repository, `noActivity` is used. Its input connection type is specified by using the connection pattern `<|`. Doing so, the user is able to construct the model by using untyped tasks and then enforce the types of the inputs by using such an approach.

```
-- Expressions assigned to Decision Branches
eYes_IsValidAC, eNo_IsValidAC :: (PF TAutoClaim) -> Bool
eYes_IsValidAC = const True
eNo_IsValidAC  = const True
```

Listing B. 25: The expressions that are assigned to the decision branches of the process in Listing B.26

Listing B. 26

```
pfAutoClaimsHandling (irAssRec,irAssSel,irAssQOSHist,irPolicy,x) =  let
                  -- Main stream
                  spf1 = ((repCI <| noActivity biTPolicy) -|- tAssignNewAC) ->>- tVerifyAC
                          -<|("Valid Auto Claim?",(branchProp eYes_IsValidAC 0.5,branchProp eNo_IsValidAC 0.5))|>=
                         (tInvestigateAC ->>- tReqAutoAss, stopBranch biTAutoClaim)
                  ospf1 @ ((otReqAutoAss_AutoClaim, otReqAutoAss_AssReq), ostop_NotValidAC_AC) = spf1 (irPolicy,x)

                  orAutoClaim = rAutoClaims otReqAutoAss_AutoClaim

                  -- Upper decision branch
                  spf2 = (fork_withOutPFs ((getSingledInputsFragment spAssessorDetermination) irAssRec irAssSel irAssQOSHist,
                         tIdentifyClaimHotSpots ->>- tReqAssFollowUp) ->>- tClaimAssr) ->>-
                        (((getSingledInputsFragment tCreateAssRep) (infConType orAutoClaim)) ->>-
                        tNegotiateSettlement ->>- tReceiveRegisterAcceptance ->>- tInitiatePayAndRepair ->>- tCloseAC)
                        -//=. stop
                  ospf2 @ (opfAutoClaimsHandling, ostop_ctrl) = spf2 otReqAutoAss_AssReq

            in   (opfAutoClaimsHandling, orAutoClaim, ostop_NotValidAC_AC, ostop_ctrl)
```

Listing B.26: The required definition to model the process in Figure B.9 using lazy serial composition in our language
The sub-process spAssessorDetermination is another process in the project which is defined in section B.3.4

Packaging the Process Fragment into a Sub-Process

Once the model is defined as a process fragment, it is packaged into a sub-process, as illustrated in Listing B.27. Any one of the previous definitions can be used.

```
spAutoClaimsHandling =  packageSubProcess "Auto Claims Handling" pfAutoClaimsHandling
```

Listing B. 6: The process fragment defined in Listing B.25 or B.26 is packaged into the sub-process `spAutoClaimsHandling`

B.3.6. Business Process 3: *Auto Claims Process*

Figure B.10 illustrates the business process model constructed in IBM WebSphere Business Modeler.
By analysing the inputs of the internal sub-processes, it should be noted that as input both processes require the content of some of the global repositories. Thus, for this process to execute these sub-processes, the contents of these repositories must be passed on as input to this process.

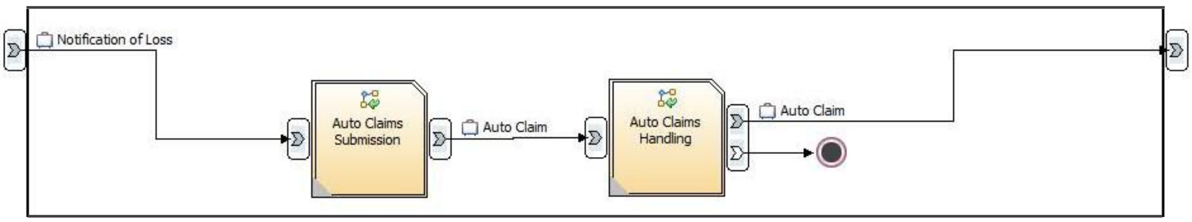


Figure B.10: Business process *'Auto Claims Handling'* in *External Claims Assessor Management (ECAM) project*

Construction 1: *Constructing the Model using only Basic Modelling Elements*

Listing B.28 defines the model and the typed sub-processes that are used to define the process.

```
-- Process Fragment
pfAutoClaimsProcess (irAssRec,irAssSel,irAssQOSHist,irPolicy,x) = let
     -- Sub-Processes in sequence
     ospAutoClaimsSubmission = spAutoClaimsSubmission (x, irPolicy)
     (o1spAutoClaimsHandling, o2spAutoClaimsHandling) = spAutoClaimsHandling
                       (irAssRec,irAssSel,irAssQOSHist,irPolicy,ospAutoClaimsSubmission)
      -- Data flow Splitter & Stop Node
     (opfAutoClaimsProcess, oCtrl) = (ctrl_dataFlowSplitter o1spAutoClaimsHandling)
                                      :: (PF TAutoClaim, PF (Flow  Ctrl))
     ostop_Ctrl = stop oCtrl
  in (opfAutoClaimsProcess, ostop_Ctrl)
```

Listing B.7: Process fragment `pfAutoClaimsProcess`, defined by using *only Basic Modelling Elements.*

Construction 2: *Constructing the Model using Connection Patterns*

Listing B.29 defines the model using connection patterns. Since the two sub-processes require additional input (that is the contents of the specific repositories, which are passed as inputs to this process), besides that produced by the previous fragment, the function provided in our language `getSingledInputsFragment` is used. This curries the input function such that it converts it from a function that accepts a tuple as an input to a function that accepts the inputs as single. In this way, by providing the first inputs, that is the contents of the repositories, we are still able to connect this fragment with others by using the provided connection patterns, such that the remaining input would be obtained from the fragment connect to it.

```
pfAutoClaimsProcess (irAssRec,irAssSel,irAssQOSHist,irPolicy,x) =
                  (spAutoClaimsSubmission ->-
                  ((getSingledInputsFragment spAutoClaimsHandling)
                          irAssRec irAssSel irAssQOSHist irPolicy) ->-
                  ((noActivity biTAutoClaim) -|- (id -//=.stop))) (x,irPolicy)
```

Listing B.29: Process fragment `pfAutoClaimsProcess`, defined by using *Connection Patterns*

Packaging the Process Fragment into a Sub-Process

Once the model is defined as a process fragment, it is packaged into a sub-process, as illustrated in Listing B.30. Any one of the previous definitions can be used.

```
spAutoClaimsProcess =  packageSubProcess  "Auto Claims Process"  pfAutoClaimsProcess
```

Listing B.30: The process fragment defined in Listing B.28 or B.29 is packaged into the sub-process `spAutoClaimsProcess`

B.3.7. Business Process 3: *Auto Claims Submission*

Figure B.11 illustrates the business process model constructed in IBM WebSphere Business Modeler.

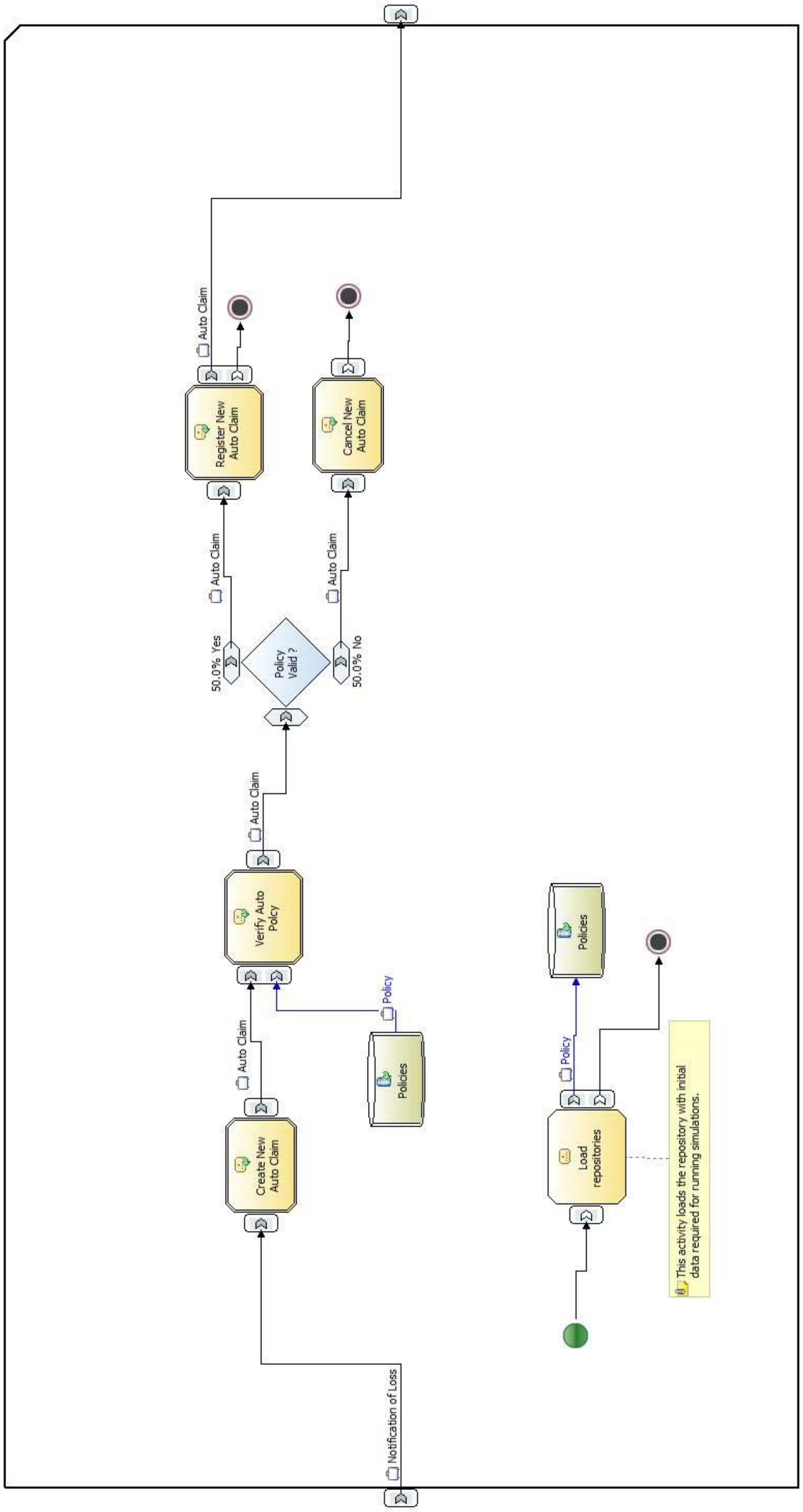


Figure B.11: Business process *'Auto Claims Submission'* in *External Claims Assessor Management (ECAM) project*

Figure B.11

Construction 1: *Constructing the Model using only Basic Modelling Elements*

Listing B.31 defines the model, the expressions assigned to the decision branches and the typed tasks that are used to define the process.

```
-- Typed Tasks

ttCreateNewAC        =  flowCI          <|tCreateNewAC|>      flowCO
ttVerifyAutoPolicy   = (flowCI,repCI)   <|tVerifyAutoPolicy|> flowCO
ttRegisterNewAC      =  flowCI          <|tRegisterNewAC|>    flowCO
ttCancelNewAC        =  flowCI          <|tCancelNewAC|>      flowCO


-- Expressions assigned to Decision Branches

eYes_FlowPolicyValid, eNo_FlowPolicyValid :: (PF (Flow TAutoClaim)) -> Bool
eYes_FlowPolicyValid = const True
eNo_FlowPolicyValid  = const True


-- Process Fragment

pfAutoClaimsSubmission (x,irPolicy) = let

      -- First 2 tasks in sequence
      ottCreateNewAC = ttCreateNewAC x
      ottVerifyAutoPolicy = ttVerifyAutoPolicy (ottCreateNewAC,irPolicy)

      -- Exclusive decision and the succeeding tasks on each of the decision branches
      (oPolicyValid, oPolicyNotValid) = exclDecision "Policy Valid?"
           ((branchProp eYes_FlowPolicyValid 0.5),(branchProp eNo_FlowPolicyValid 0.5))
           (ottVerifyAutoPolicy)
      ottRegisterNewAC  = ttRegisterNewAC oPolicyValid
      ottCancelNewAC    = ttCancelNewAC oPolicyNotValid

      -- Stop nodes attached to the final tasks (including a splitter for the topmost task)
     (opfAutoClaimsSubmission, oCtrl) = (ctrl_dataFlowSplitter ottRegisterNewAC)
                                   :: (PF (Flow TAutoClaim), PF (Flow  Ctrl))
      ostop_Ctrl      = stop oCtrl
      ostop_AutoClaim = stop ottCancelNewAC

  in (opfAutoClaimsSubmission, ostop_Ctrl, ostop_AutoClaim)
```

Listing B.31: Process fragment `pfAutoClaimsSubmission`, defined by using *only Basic Modelling Elements*. The typed tasks and the expressions assigned to the decision branches are defined above the process definition

Construction 2: *Constructing the Model using Connection Patterns*

Listing B.32 defines the model using connection patterns and the expressions that are assigned to the decision branches. The initial defined untyped tasks are used. To ensure that task `'Verify Auto Policy'` obtains its second input from a repository, the connection pattern `<|` is used. The Haskell's `id` function is used as a dummy component, which allows data to pass through, and thus be able to express the process fragment as a single function.

```
-- Expressions assigned to Decision Branches

eYes_PolicyValid, eNo_PolicyValid :: (PF TAutoClaim) -> Bool
eYes_PolicyValid = const True
eNo_PolicyValid  = const True

-- Process Fragment

pfAutoClaimsSubmission = (tCreateNewAC -|- id) ->-
                          ((noConType, repCI) <| tVerifyAutoPolicy)
                         -<|("Policy Valid?", (branchProp eYes_PolicyValid 0.5,
                              branchProp eNo_PolicyValid  0.5))|>=
                          (tRegisterNewAC -//=. stop, tCancelNewAC |><| stop)
```

Listing B.32: Process fragment `pfAutoClaimsSubmission`, defined by using *Connection Patterns*. The expressions assigned to the decision branches are defined above the process definition

Packaging the Process Fragment into a Sub-Process

Once the model is defined as a process fragment, it is packaged into a sub-process, as illustrated in Listing B.33. Any one of the previous definitions can be used.

```
spAutoClaimsSubmission = packageSubProcess "Auto Claims Submission"
                              pfAutoClaimsSubmission
```

Listing B.33: The process fragment defined in Listing B.31 or B.32 is packaged into the sub-process `spAutoClaimsSubmission`

B.4. Quickstart Finance Project

This is one of the sample projects that come along with IBM WebSphere Business Modeler Advanced v6.0.2[1]. This project contains business processes that handle loan applications within a company. It has its own user-defined complex types (mainly business items) and two processes, as illustrated in Figure B.12. It does not have any repositories.

Business Items:

- Application
- E-mail Notification
- Funds

Global Repositories:

- *None*

Business Processes:

- Loan Application (As Is)
- Loan Application (To Be)

Figure B.12: Business Items, Global Repositories and Business Processes defined in *Quickstart Finance Project*

B.3.1. User-Defined Types - Business Items

The required business items are defined in Listing B.34.

[1] http://www.ibm.com/developerworks/websphere/zones/businessintegration/roadmaps/modeler/roadmap_advanced.html

```
module Types where

   import FuncBPML

   --  Unclassified Types (i.e. not specified as business item and complex type)
   newtype Appl       = Appl ()       deriving Typeable
   newtype EmailNot   = EmailNot ()   deriving Typeable
   newtype Funds      = Funds ()      deriving Typeable

   -- Type Synonyms for Business Items
   type TAppl     = BI Appl
   type TEmailNot = BI EmailNot
   type TFunds    = BI Funds

   -- Defining Types as Complex
   instance ComplexType (TAppl)
   instance ComplexType (TEmailNot)
   instance ComplexType (TFunds)

   -- First Class Objects to represent the Type
   biTAppl      = dType :: TAppl
   biTEmailNot  = dType :: TEmailNot
   biTFunds     = dType :: TFunds
```

Listing B.34: Quickstart Finance project - User-defined Types (Business Items)

B.3.2. Global Repositories

The project does not have any repositories.

B.3.3. Tasks

The required tasks are defined in Listing B.35 and can be reused by any process in the project.

```
module Tasks where

   import Types

   -- used in Loan Application (To Be) but also in Loan Application (As Is)
   tReviewLoanApp        = task "Review Loan and Application"   (bvTString :-> biTAppl)
   tApplySpecialTerms    = task "Apply Special Terms"           (biTAppl :-> biTAppl)
   tRejectApp_NotifyCust = task "Reject Application and Notify Customer"
                                                                 (biTAppl :-> biTEmailNot)
   tDisburseFunds        = task "Disburse Funds"                (biTAppl :-> biTFunds)
```

Listing B.35: Quickstart Finance project – Tasks

B.3.4. Business Process 1: *Loan Application (As Is)*

Figure B.13 illustrates the business process model constructed in IBM WebSphere Business Modeler.

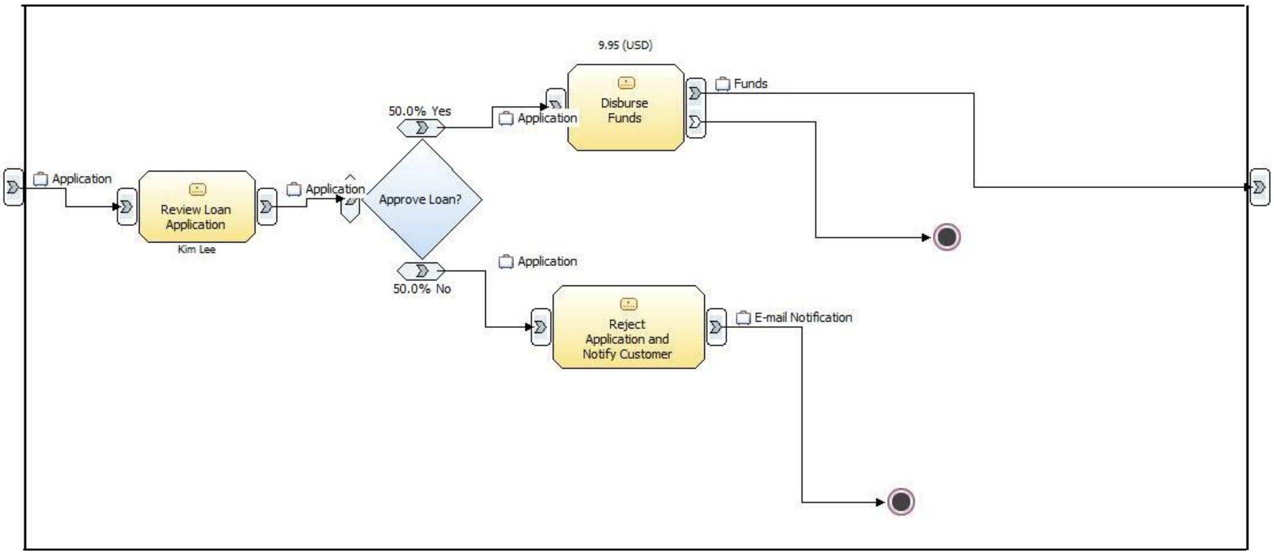


Figure B.13: Business process '*Loan Application (As Is)*' in *Quickstart Finance project*

Construction 1: *Constructing the Model using only Basic Modelling Elements*

Listing B.36 defines the model, the expressions assigned to the decision branches and the typed tasks that are used to define the process.

```
-- Typed Tasks

ttReviewLoanApp         = flowCI <|tReviewLoanApp|>        flowCO
ttRejectApp_NotifyCust = flowCI <|tRejectApp_NotifyCust|> flowCO
ttDisburseFunds         = flowCI <|tDisburseFunds|>        flowCO

-- Expressions assigned to Decision Branches

eYes_Flow, eNo_Flow:: (PF (Flow TAppl)) -> Bool
eYes_Flow = const True
eNo_Flow  = const True

-- Process Fragment

pfLoanApplAsIs x = let
           -- First task
           ottReviewLoanApp  = ttReviewLoanApp  x

           -- 2-branch Exclusive Decision & fragments attached to the branches
            (oApproved,oNotApproved) = exclDecision "Approve Loan?"
                      (branchProp eYes_Flow 0.5,branchProp eNo_Flow 0.5) ottReviewLoanApp
           ottDisburseFunds    = ttDisburseFunds  oApproved
           ottRejectApp_NotifyCust = ttRejectApp_NotifyCust oNotApproved
           ostop_EmailNotif = stop ottRejectApp_NotifyCust

           -- Data flow Splitter & Stop Node
           (opfLoanApplAsIs, oCtrl) = (ctrl_dataFlowSplitter ottDisburseFunds )
                                       :: (PF (Flow TFunds), PF (Flow  Ctrl))
           ostop_ctrl = stop oCtrl

      in  (opfLoanApplAsIs, ostop_EmailNotif, ostop_ctrl)
```

Listing B.36: Process fragment `pfLoanApplAsIs`, defined by using *only Basic Modelling Elements*. The typed tasks and the expressions assigned to the decision branches are defined above the process definition

Construction 2: Constructing the Model using Connection Patterns

Listing B.37 defines the model by using connection patterns and the expressions that are assigned to the decision branches. The untyped version of tasks is used to allow the system to automatically infer types.

```
-- Expressions assigned to Decision Branches
eYes, eNo:: (PF TAppl) -> Bool
eYes = const True
eNo  = const True

-- Process Fragment
pfLoanApplAsIs = tReviewLoanApp
                 -<|("Approve Loan?", (branchProp eYes 0.5,branchProp eNo 0.5))|>=
                 (tDisburseFunds -//=.stop,tRejectApp_NotifyCust |><| stop)
```

Listing B.37: Process fragment `pfLoanApplAsIs`, defined by using *Connection Patterns.* The expressions assigned to the decision branches are defined above the process definition

Packaging the Process Fragment into a Sub-Process

Once the model is defined as a process fragment, it is packaged into a sub-process, as illustrated in Listing B.38. Any one of the previous definitions can be used.

```
spLoanApplAsIs =  packageSubProcess "Loan Application (As Is)"  pfLoanApplAsIs
```

Listing B.38: The process fragment defined in Listing B.36 or B.37 is packaged into the sub-process `spLoanApplAsIs`

B.3.5. Business Process 2: *Loan Application (To Be)*

Figure B.14 illustrates the business process model constructed in IBM WebSphere Business Modeler.

Construction 1: Constructing the Model using only Basic Modelling Elements

Listing B.39 defines the model, the expressions assigned to the decision branches and the typed tasks that are used to define the process. Listing B.40 defines the actual process.

```
-- Typed Tasks
ttReviewLoanApp         = flowCI <|tReviewLoanApp|>         flowCO
ttApplySpecialTerms     = flowCI <|tApplySpecialTerms|>     flowCO
ttRejectApp_NotifyCust = flowCI <|tRejectApp_NotifyCust|> flowCO
ttDisburseFunds         = flowCI <|tDisburseFunds|>         flowCO

-- Expressions assigned to Decision Branches
eApprove_Flow, eApproveWithTerms_Flow, eReject_Flow :: (PF (Flow TAppl)) -> Bool
eApprove_Flow           = const True
eApproveWithTerms_Flow  = const True
eReject_Flow            = const True
```

Listing B.39: The typed tasks and the expressions assigned to the decision branches are defined above the process definition

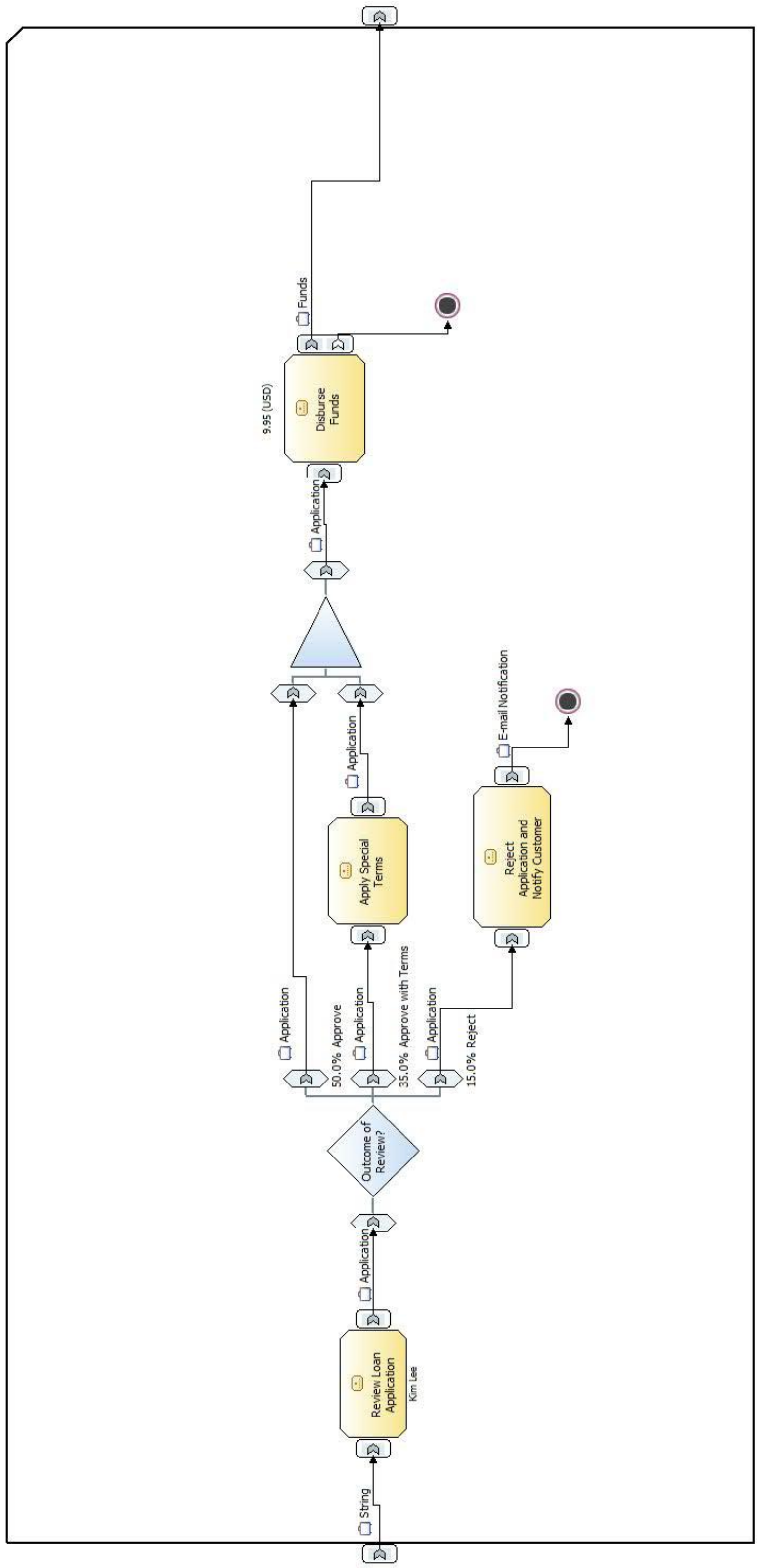


Figure B.14: Business process *'Loan Application (To Be)'* in *Quickstart Finance project*

Figure B. 14

```
-- Process Fragment

pfLoanApplToBe x = let
          -- First task
          ottReviewLoanApp = ttReviewLoanApp x

          -- 3-branch Exclusive Decision & fragments attached to branches 2 & 3
          (oApproved,oApprovedWithTerms,oRejected) =
                 exclDecision "Approve Loan?" (branchProp eApprove_Flow 0.5,
                 branchProp eApproveWithTerms_Flow 0.35, branchProp eReject_Flow 0.15)
                 ottReviewLoanApp
          ottApplySpecialTerms    = ttApplySpecialTerms  oApprovedWithTerms
          ottRejectApp_NotifyCust = ttRejectApp_NotifyCust oRejected
          ostop_EmailNotif = stop ottRejectApp_NotifyCust

          -- 3-input Merge
          omerge = merge (oApproved::(PF TAppl), ottApplySpecialTerms)

          -- Final task
          ottDisburseFunds = ttDisburseFunds omerge

          -- Data flow Splitter & Stop Node
          (opfLoanApplToBe, oCtrl) = (ctrl_dataFlowSplitter ottDisburseFunds)
                                     :: (PF (Flow TFunds), PF (Flow  Ctrl))
          ostop_ctrl = stop oCtrl
    in    (opfLoanApplToBe, ostop_EmailNotif, ostop_ctrl)
```

Listing B.40: Process fragment `pfLoanApplToBe`, defined by using *only Basic Model Elements.* The typed tasks and the expressions assigned to the decision branches are defined in Listing B.39

Construction 2: *Constructing the Model using Connection Patterns*

Listing B.41 defines the expressions that are assigned to the decision branches. The actual model is defined in Listing B.42. The untyped version of tasks is used to allow the system to automatically infer types.

Note that different from construction 1, instead of explicitly typing the output from the first decision branch before passing it as input to the merge, the `noActivity` function is used. This allows the flow to pass through without being modified and at the same time specifies its expected type, to allow the system to automatically infer the type.

```
-- Expressions assigned to Decision Branches

eApprove, eApproveWithTerms, eReject :: (PF TAppl) -> Bool
eApprove      = const True
eApproveWithTerms  = const True
eReject            = const True
```

Listing B.41: The expressions assigned to the decision branches of the process defined in Listing B.42.

```
-- Process Fragment
    pfLoanApplToBe x = let
              -- Sub-process fragment 1 (initial part just before merge)
              spf1 = tReviewLoanApp -<|("Approve Loan?", (branchProp eApprove 0.5,
                     branchProp eApproveWithTerms 0.35, branchProp eReject 0.15))|>=
                     (noActivity biTAppl,tApplySpecialTerms, tRejectApp_NotifyCust |><| stop)
              (oApproved, otApplySpecialTerms, ostop_EmailNotif) = spf1 x

              -- Sub-process fragment 2 (final part -> used spf1)
              spf2 = merge ->- tDisburseFunds -//=. stop
              (opfLoanApplToBe, ostop_ctrl)  = spf2 (oApproved, otApplySpecialTerms)

          in  (opfLoanApplToBe, ostop_EmailNotif, ostop_ctrl)
```

Listing B. 42: Process fragment `pfLoanApplToBe`, defined using *Connection Patterns.* The expressions assigned to the decision branches are defined in Listing B.41

Packaging the Process Fragment into a Sub-Process

Once the model is defined as a process fragment, it is packaged into a sub-process, as illustrated in Listing B.43. Any one of the previous definitions can be used.

```
spLoanApplToBe =  packageSubProcess "Loan Application (To Be)"  pfLoanApplToBe
```

Listing B.43: The process fragment defined in Listing B.41 or B.42 is packaged into the sub-process `pfLoanApplToBe`

B.5. Other Sample Models

This section contains other samples which were intentionally constructed to illustrate specific features of our language. Different from the other sections, these samples are simple process fragments and not entire projects. However, they still make use and define their own business items, tasks and repositories, where necessary. Various connection patterns are used at different levels of abstraction, to assist users to rapidly construct models and to prevent the introduction of new errors when such patterns are constructed manually.

B.5.1. Order Handling

Given an order, the process first decides whether the customer should be given a gift and/or a discount, reduces the item from the stock and finally handles customer's payment and prepares the package with the goods. The customer can pay either by cash or by credit card or with both (i.e. part by cash and part by credit card).
Figure B.15 illustrates such a business process model constructed in IBM WebSphere Business Modeler.

Figure B.15

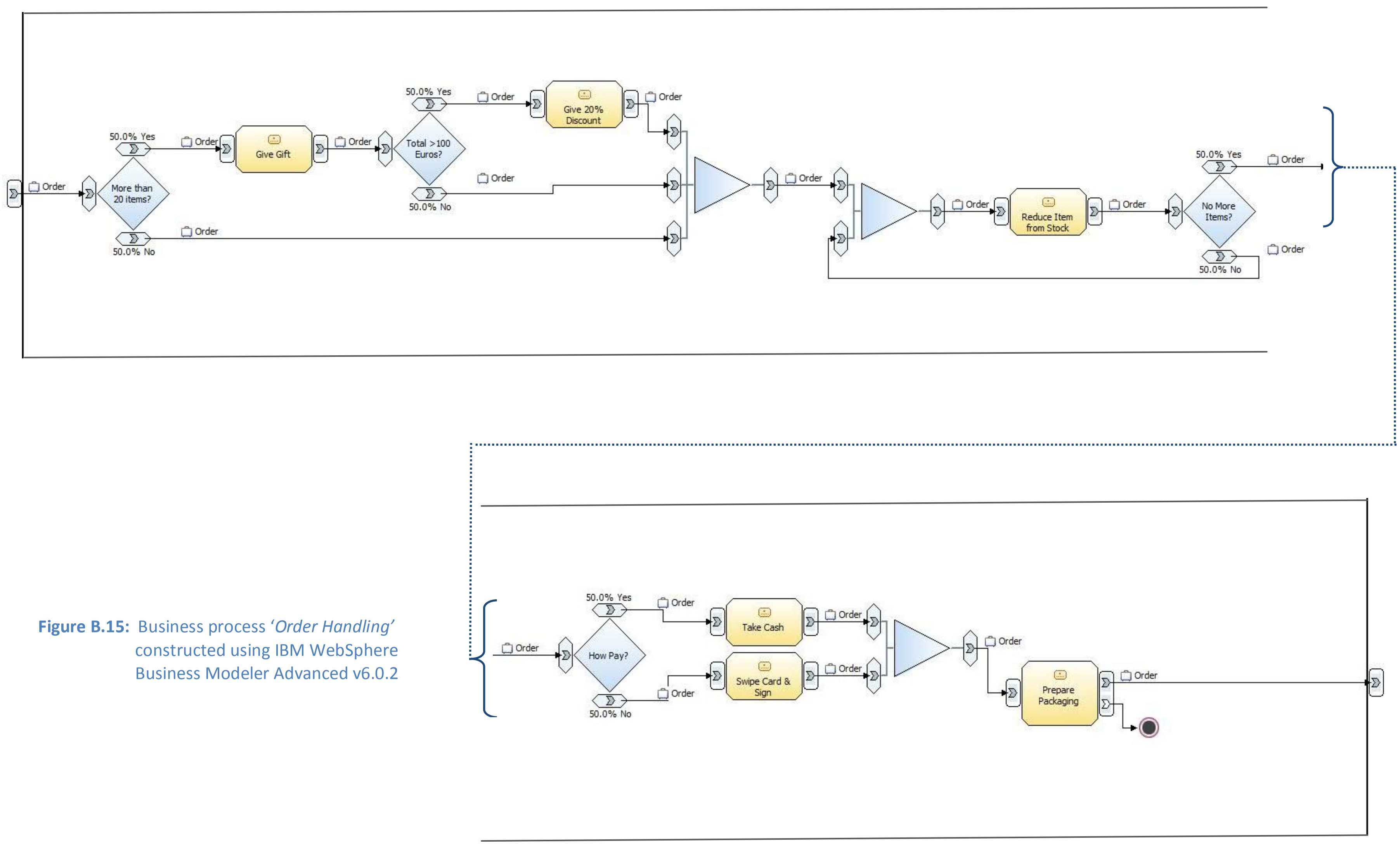


Figure B.15: Business process *'Order Handling'* constructed using IBM WebSphere Business Modeler Advanced v6.0.2

In Figure B.15, three main fragments and a final task can be identified, as illustrated in Figure B.16. These can be handled individually and later on connected in sequence.

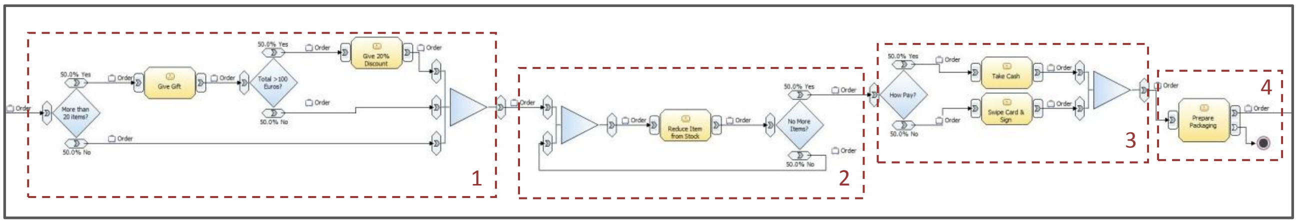


Figure B.16: Process fragment in Figure B.15 is decomposed into sub-process fragments. The decisions and merges are marked in green. To view the details of the process fragment, refer to Figure B.15

As illustrated in Figure B.16, the process fragment in Figure B.15 can be decomposed into sub-process fragments such that the first block is a *triangular-shaped model* for which the connection pattern `tri_exclDecisions_onUpperBranch_Merge` can be used, the second is a *merge - exclusive decision cycle* which can be constructed by using the connection pattern `soundCycle`, and the third fragment is an *inclusive decision followed by a merge*, which can be constructed in a sound manner (not to introduce lack of synchronization) by explicitly using exclusive decisions and forks (as shall be illustrated in *Construction 1*), in which case the connection pattern `sound_inclDecision_Merge` should be used. The latter decision in the third fragment is an inclusive decision; the customer might decide to pay part by cash and part by credit card, in which case, the fragment would lack synchronization. Finally, the process ends with a task and a stop node. (For more details about these connection patterns, refer to the tutorial, Appendix A)

User-Defined Types - Business Items

The required business items are defined in Listing B.44.

```
--  Unclassified Types (i.e. not specified as business item and complex type)
newtype Order = Order () deriving Typeable

-- Type Synonyms for Business Items
type TOrder = BI Order

-- Defining Types as Complex
instance ComplexType (TOrder)

-- First Class Objects to represent the Type
biTOrder = dType :: TOrder
```

Listing B.44: User-defined Type (Business Item) used within process '*Order Handling*'

Global Repositories

No repositories are required

Tasks

The required tasks are defined in Listing B.45.

```
tGiveGift             = task "Give Gift"              (biTOrder :-> biTOrder)
tGiveDiscount         = task "Give 20% Discount"      (biTOrder :-> biTOrder)
tReduceItemFromStock  = task "Reduce Item From Stock" (biTOrder :-> biTOrder)
tTakeCash             = task "Take Cash"              (biTOrder :-> biTOrder)
tSwipeCardSign        = task "Swipe Card & Sign"      (biTOrder :-> biTOrder)
tPreparePackaging     = task "Prepare Packaging"      (biTOrder :-> biTOrder)
```

Listing B.45: Tasks used within process '*Order Handling*'

Construction 1: *Constructing the Model using only Basic Modelling Elements*

Listing B.46 defines the expressions assigned to the decision branches and the typed tasks that are used to define the process. Listing B.47, defines the actual model.

Note that expression `eCashCred` is true, when payment is carried out partly by cash and partly by credit card, and is assigned to the third decision branch that is used to mimic an inclusive decision.

```
-- Typed Tasks

ttGiveGift             =  flowCI  <|tGiveGift|>            flowCO
ttGiveDiscount         =  flowCI  <|tGiveDiscount|>        flowCO
ttReduceItemFromStock  =  flowCI  <|tReduceItemFromStock|> flowCO
ttTakeCash             =  flowCI  <|tTakeCash|>            flowCO
ttSwipeCardSign        =  flowCI  <|tSwipeCardSign|>       flowCO
ttPreparePackaging     =  flowCI  <|tPreparePackaging|>    flowCO


-- Expressions assigned to Decision Branches

eYes, eNo :: (PF (Flow TOrder)) -> Bool
eYes = const True
eNo  = const True

eCash, eCredit, eCashCredit :: (PF (Flow TOrder)) -> Bool
eCash       = const True
eCredit     = const True
eCashCredit = \x -> (eCash x) && (eCredit x)
```

Listing B.46: The expressions assigned to the decision branches and the typed tasks used in the process, in Listing B.47

Listing B. 47

```
pfOrderHandling x     =  let

          -- Sub-Fragment 1 - Triangular Model
          (oMoreThan20Items,oLessThan20Items) = exclDecision ">20 Items?" (branchProp eYes_Flow 0.5, branchProp eNo_Flow 0.5)
                                                              (x :: PF (Flow TOrder))
          ottGiveGift = ttGiveGift oMoreThan20Items
          (oMoreThan500Euros,oLessThan500Euros) = exclDecision "Total >500Euros?" (branchProp eYes_Flow 0.5, branchProp eNo_Flow 0.5)
                                                              ottGiveGift
          ottGiveDiscount = ttGiveDiscount oMoreThan500Euros
          omerge1 = merge (ottGiveDiscount, oLessThan500Euros :: PF (Flow TOrder), oLessThan20Items :: PF (Flow TOrder))

          -- Sub-Fragment 2 - Sound Cycle
          omerge2 = merge (omerge1 :: PF (Flow TOrder), oMoreItems :: PF (Flow TOrder))
          ottReduceItemFromStock = ttReduceItemFromStock omerge2
         (oNoMoreItems,oMoreItems) = exclDecision "No More Items?" (branchProp eYes_Flow 0.5, branchProp eNo_Flow 0.5) ottReduceItemFromStock

          -- Sub-Process Fragment 3 - Sound Inclusive Decision Merge Fragment
          (oCash,oCredit,oCashCredit)= exclDecision "How Pay?"
                                                    (branchProp eCash_Flow 0.4, branchProp eCredit_Flow 0.5, branchProp eCashCredit_Flow 0.1)
                                                    (oNoMoreItems :: PF (Flow TOrder))
          ottTakeCash = ttTakeCash oCash
          ottSwipeCardSign = ttSwipeCardSign oCredit
         (o1fork,o2fork) = fork (oCashCredit :: PF (Flow TOrder))
          ottTakeCash2    = ttTakeCash o1fork
          ottSwipeCardSign2 = ttSwipeCardSign o2fork
          ojoin = join (ottTakeCash2, ottSwipeCardSign2)
          omerge3 = merge (ottTakeCash, ottSwipeCardSign, ojoin :: PF (Flow TOrder))

          -- Sub-Fragment 4 - Final task and stop node
          ottPreparePackaging = ttPreparePackaging omerge3
         (opfOrderHandling, oCtrl) = (ctrl_dataFlowSplitter ottPreparePackaging) :: (PF (Flow TOrder), PF (Flow  Ctrl))
          ostop_ctrl = stop oCtrl

      in  (opfOrderHandling, ostop_ctrl)
```

Listing B.47: The required definition to model the process in Figure B.15 using only basic modelling elements
The logical fragments referenced in the definition have been marked on the model in Figure B.16

Construction 2: *Constructing the Model using Connection Patterns*

Listing B.48 defines the expressions that are assigned to the decision branches. The actual model is defined in Listing B.49.

The untyped version of tasks is used to allow the system to automatically infer types. Different from *Construction 1*, input and output arguments do not need to be explicitly typed.

```
-- Expressions assigned to Decision Branches

eYes, eNo:: (PF TOrder) -> Bool
eYes = const True
eNo  = const True

eCash, eCredit:: (PF TOrder) -> Bool
eCash   = const True
eCredit = const True
```

Listing B.48: The expressions assigned to the decision branches in Listing B.49

```
pfOrderHandling x  =  let

        -- Sub-Fragment 1 - Triangular Model
        pfOrderHandling2=let

        -- Sub-Process Fragment 1 - Triangular Model
        spf1 = tri_exclDecisions_onUpperBranch_Merge ["&gt;20 Items?", "Total &gt;500 Euros?"]
                                                     [(branchProp eYes 0.5, branchProp eNo 0.5),(branchProp eYes 0.5, branchProp eNo 0.5)]
                                                     [(tGiveGift, noActivity biTOrder), (tGiveDiscount, noActivity biTOrder)]

        -- Sub-Process Fragment 2 - Sound Cycle
        spf2 = soundCycle tReduceItemFromStock ("No More Items?",(branchProp eYes 0.5, branchProp eNo 0.5))

        -- Sub-Process Fragment 3 - Sound Inclusive Decision Merge
        spf3 = sound_inclDecision_Merge "How Pay?" (branchProp eCash 0.5, branchProp eCredit 0.5) (tTakeCash,tSwipeCardSign)

   in
        spf1 ->>- spf2 ->>- spf3 ->>- tPreparePackaging -//=. stop
```

Listing B.49: The required definition to model the process in Figure B.15 using connection patterns
The logical fragments referenced in the definition have been marked on the model in Figure B.16

Packaging the Process Fragment into a Sub-Process

Once the model is defined as a process fragment, it is packaged into a sub-process, as illustrated in Listing B.50. Any one of the previous definitions can be used.

```
spOrderHandling =  packageSubProcess  "Order Handling"  pfOrderHandling
```

Listing B.50: The process fragment defined in Listing B.47 or B.49 is packaged into the sub-process `spOrderHandling`

B.6. Conclusion

These are just simple sample models which can be used and referenced by modellers who would like to start defining models in our language. It would be helpful for the user to take a look at these realistic models while reading the tutorial (Appendix A). In this way, the user would have a complete code example of where and how the provided features and components in the language can be used. These samples can also be used as case studies (as done in Chapter 6) to analyse the different ways how models can be constructed and thus decide upon new connection patterns that can be added to the language and upon a feasible approach to construct the models.