TECHNICAL REPORT

Report No. CS2009-02
Date: December 2009

Embedded Languages for Business Process Modelling, Transformation and Quality Assurance in Business-Driven Development

Luana Micallef
Gordon J. Pace

Department of Computer Science
University of Malta
Msida MSD 06
MALTA

Tel: +356-2340 2519
Fax: +356-2132 0539
http://www.cs.um.edu.mt

Embedded Languages for Business Process Modelling, Transformation and Quality Assurance in Business-Driven Development

Luana Micallef
School of Computing
University of Kent
Canterbury, UK
lm304@kent.ac.uk

Gordon J. Pace
Department of Computer Science
University of Malta
Msida, Malta
gordon.pace@um.edu.mt

Abstract: *In Business-Driven Development (BDD), process models are produced by business analysts. To ensure that the defined requirements are satisfied, the IT solution must ideally be derived directly from the specifications through a process of model refinement. However, if the original models contain errors or lack some technical detail, an incorrect implementation would be inferred and the entire BDD life-cycle would have to be revised. In this report, we investigate the use of embedded language techniques to enable more abstract model descriptions and enable quality assurance and transformation of models. We have embedded such a domain-specific language in the functional programming language Haskell and show how it enables: (i) the rapid development of models in a concise and abstract manner, focusing on the specifications rather than the implementation and ensuring that all the required details to generate the executable code are specified; (ii) quality assurance of the models through the use of Haskell's type checker, at construction-time and through soundness analysis; (iii) transformation, analysis and interpretation of the models; and (iv) definition of composite model transformations, including the use of quality assurance.*

Embedded Languages for Business Process Modelling, Transformation and Quality Assurance in Business-Driven Development

Luana Micallef
School of Computing
University of Kent
Canterbury, UK
lm304@kent.ac.uk

Gordon J. Pace
Department of Computer Science
University of Malta
Msida, Malta
gordon.pace@um.edu.mt

Abstract: *In Business-Driven Development (BDD), process models are produced by business analysts. To ensure that the defined requirements are satisfied, the IT solution must ideally be derived directly from the specifications through a process of model refinement. However, if the original models contain errors or lack some technical detail, an incorrect implementation would be inferred and the entire BDD life-cycle would have to be revised. In this report, we investigate the use of embedded language techniques to enable more abstract model descriptions and enable quality assurance and transformation of models. We have embedded such a domain-specific language in the functional programming language Haskell and show how it enables: (i) the rapid development of models in a concise and abstract manner, focusing on the specifications rather than the implementation and ensuring that all the required details to generate the executable code are specified; (ii) quality assurance of the models through the use of Haskell's type checker, at construction-time and through soundness analysis; (iii) transformation, analysis and interpretation of the models; and (iv) definition of composite model transformations, including the use of quality assurance.*

1 Introduction

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 organization. The models should then act as an abstract representation of the solution. It is essential to adapt to Business-Driven Development (BDD) (see [Mit05] and [KHK+08]) 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 (whereby a process waits indefinitely for some data or operation to complete). If models are not quality assured at the modelling phase, errors would be discovered later and the entire BDD life-cycle 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 [KGK+07].

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 organizations, 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 may be developed. This would capture precisely the semantics of the business process modelling domain and thus provide the right abstraction to define model descriptions which are easy to comprehend and reason about.

Language embedding is a technique from the programming language community in which a domain-specific language is built within a general-purpose one, often referred to as the host language. By giving means of constructing domain-specific programs as part of programs written in the host language, one can use abstraction techniques from the host language to build and structure descriptions of domain-specific programs. Furthermore, still within the host language, one has access to the domain-specific programs as data objects which can be manipulated, analysed and executed.

In this report, we propose the use of embedded languages to build a domain-specific language to model business processes embedded in Haskell. We show how such an approach enables us to model, transform and quality assure business processes in BDD. Through the use of abstraction mechanisms of the host language, one can build concise and compositional model descriptions, focusing on the required behaviour rather than the implementation, and then transform and analyse them in various ways. An important aspect is the quality assurance of these models by carrying out checks ranging from the host language's type system, to construction-time checks and run-time soundness analysis. The availability of the host language sitting above the model description language enables the user to define new composite model transformations, possibly ensuring quality assurance as part of the transformation. Other tools can be used in conjunction to the embedded language, such that models would be easily constructed, analysed and transformed in the embedded language itself. Such a tool, for

instance a model checker for more sophisticated analysis or a language for developers to write their own custom transformations and analysis, can be used as a plug-in in a standard business process modelling tool. Visual editing, on the other hand, could be done in an external tool and the definition of the model could be transformed from a visual to a textual one and vice versa. Such a language was embedded in Haskell to capture the domain semantics of IBM's WebSphere Business Modeler Advanced (WSBM).[1]

2 Embedded Languages

The use of domain-specific languages usually leads to more effective, targeted solutions. However, building the necessary infrastructure around a domain-specific language to enable abstraction, modularization, compiling, etc. can be a long and laborious task. A technique to circumvent this problem is the use of embedded languages — the building of the domain-specific language within a general-purpose language (usually referred to as the host language). By defining the syntactic constructs and the type system of the domain-specific language within the host language, domain-specific programs can be built as objects in the host language. Apart from constructs to build domain-specific programs, one typically also provides (from within the host language) different interpretations of domain-specific programs — for execution, visualization, analysis, etc.

This approach enables one to inherit the tools and features of the host language in the design, development and analysis of the domain-specific language [Hud98]. In this way, the language designers are able to reuse the infrastructure of the host and thus focus directly on the semantics of the new language. Since the limitations of the host are also inherited, it is important for the language designer to choose the appropriate host to embed the required language for that specific domain.

The degree of abstraction achievable in the host language and its syntax, impinge on the degree of assimilation of the embedded language within the host language and how sharply delineated the boundary between the domain-specific code and the rest of the program. Through various case studies, it has been shown that the functional language Haskell can be an excellent host language to provide the right modularity and abstraction to develop a language which is maintainable, extendible, easy to design and easy to use [HJ94]. Over the years, Haskell [Jon03] has been used for embedded languages for various diverse domains ranging from financial contracts [JES00] to hardware description [CSS03].

[1] `http://www-01.ibm.com/software/integration/wbimodeler/advanced/`

Using this approach, programs in the domain-specific language are simply data objects in the host language and therefore, once the end-user defines and generates the required domain-specific program, it is possible to analyse, interpret, manipulate, transform, test and verify it through other programs written in the host language. This gives a limited degree of meta-programming flexibility — programs in the embedded language are both code, which can be executed or interpreted, and data objects, which can be generated, manipulated and analysed. In this way, higher-order embedded programs can be seen as having two run-time phases: (i) the execution of the code in the host language producing a fixed embedded program, and (ii) the execution and analysis of these programs using the interpretation provided in the host language. For example, when a hardware description language is embedded, a program in the host language can be defined such that, given a number n, a bit-adder for words of size n is produced. The simulation of a 24-bit adder consists of a run with distinct phases: (i) running the general adder function to produce the 24-bit adder, and (ii) simulating the concrete 24-bit adder with the given input.

At face value, an embedded language may appear to be no more than a domain-specific library in the general-purpose language, using which, programmers are able to describe and manipulate domain-specific objects. However, careful design of domain-specific combinators creates the illusion that the domain-specific code fragments are 'programs' written in a domain-specific language that are part of the programs in the host language. The programmer may then generate, analyse and manipulate these 'programs' (written in the domain-specific language) as though they were part of the host language itself and thus, effectively make the host act as a meta-language for the embedded language. Clearly, the more flexible and high-level the host language and its syntax are, the more difficult it becomes to distinguish where the domain-specific program ends and where the rest of the code starts.

From the language designer's point of view, the main advantage of designing a domain-specific language and embedding it within a host language is that one needs not reinvent the wheel and create a new general-purpose language. From the end-user's perspective, the main advantage is that the meta-language is a standard language with which one may already be familiar and knowledge of which goes beyond the use of the domain-specific language.

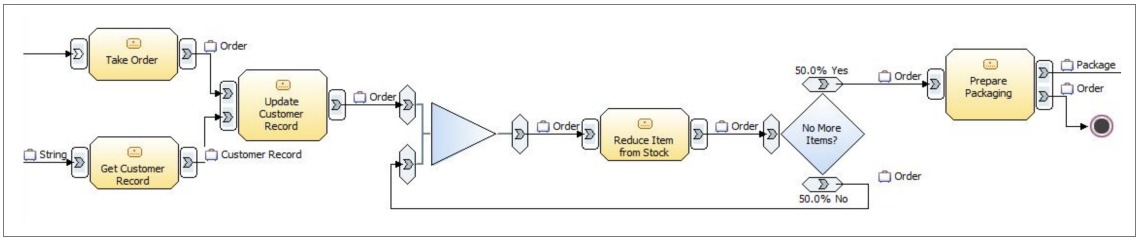

Figure 1: A *process* to handle orders

3 An Embedded Business Process Modelling Language

In process modelling, a sequence of business activities with clearly defined inputs and outputs is specified in a particular order with the aim of capturing the business' requirements and objectives. Such models can represent the current (*'as is'*) and the future (*'to be'*) processes of the organization. By analysing these models, the efficiency and the quality of the processes can be improved before they are implemented.

Simple shapes, similar to conventional flowchart components, are usually used to define the required process. These are normally classified as: *activities*, *gateways* and *events*. Although the actual behaviour of the process is defined by tasks (activities), gateways (such as a merge or a decision) handle the flow of control and data within the process, and events (such as a start or stop node) ensure the initialization or termination of the process. This can be seen in the order handling example shown in Figure 1 — Once an order is taken and the customer record is retrieved, the record is updated and the ordered items are reduced from the stock. The items are then packaged and returned to the customer, and the order is discarded. In this example there are: four tasks (represented as rectangles), two gateways (a decision named *No more items?* and a merge represented as a triangle) and one event (a stop node represented as a black circle). Connectors are then used to connect the components and to allow user-defined business items (e.g. *Order*, *Customer Record* or *Package*), basic typed items (e.g. *String* to represent the customer identification code) and control (e.g. the input to task *Take Order*) to flow between the components. All types have to be appropriately packaged to enable their use in models.

We proceed to embed these concepts in a domain-specific language for business process modelling in Haskell [Jon03]. This is possible by building constructs to describe the concepts (individual blocks) in the models and by providing means to compose them together to produce larger models. Abstraction and modularization techniques are inherited directly from the host language.

Listing 1: Defining the order handling process in Figure 1 using basic modelling elements in our language

```
orderHandling (x,y) =
  let

    updateCustRec_out= updateCustRec(takeOrder x,getCustRec y)
    merge_out = merge (updateCustRec_out, moreItems_out)

    (noMoreItems_out, moreItems_out) =
                 decision "No More Items?",
                (branch_NoMoreItems 0.5, branch_MoreItems 0.5)
                (reduceItemFromStock merge_out)

    (preparePackaging_Package, preparePackaging_Order) =
                              preparePackaging noMoreItems_out

    stop_out = stop preparePackaging_Order

  in (preparePackaging_Package, stop_out)
```

3.1 Embedding the Language

An important feature of embedded languages is that domain-specific programs typically appear as first class objects in the host language. To merge seamlessly as part of the host language, processes and modelling elements are embedded as functions, and through the use of function definition, one can easily define processes and reuse them to define more complex models using higher-order functions. This is possible through the use of functional application as illustrated in Listing 1, where the functions `takeOrder`, `getCustRec`, `updateCustRec`, `merge`, `reduceItemFromStock`, `decision`, `preparePackaging` and `stop`, which are also basic modelling elements, are applied by assigning the appropriate inputs. The elements are then connected by using the output of one function as the input of another.

The embedded language, supporting basic modelling constructs and combinators, is provided to the end-users as a Haskell library. The provided basic constructs in the language include modelling elements such as: `stop` (an event to terminate a process), `decision` (a gateway to determine how the flow should be diverted on the outgoing branches, depending on some boolean expressions), and `task` (to define specific activities such as the task `updateCustRec`, which given an *Order*, updates a *Customer Record* accordingly). The provided combinators in the library are then

Listing 2: Defining the task *Update Customer Record* for the order handling process in Figure 1

```
updateCustRec = task "Update Customer Record"
                     ((type_Order,type_CustRec):->type_Order)
```

used to combine the various tasks so that, similar to composite functions, process fragments carrying out complex operations can be produced.

Using the Haskell type system, one can ensure that the entire system (including process fragments and modelling elements) is strongly-typed. This means that the type of all the inputs and outputs of tasks are clearly defined and checked for conflicts at construction-time. Since these types explicitly define the actual operation and objective of a task, business items specific to that business domain are defined as domain-specific types. Typing of the language combinators ensure that the constructed fragments would in turn be strongly-typed.

This means that before tasks such as *Update Customer Record* (in Figure 1) are used to construct a process (as in Listing 1), they must first be defined as in Listing 2.

Multiple structured input and output types can be defined. These types are specified as objects and as in this case (Listing 2), they can be specific to the concerned business domain. For instance, `type_CustRec` refers to a user-defined business item of type *Customer Record*.

One of the advantages of using an embedded language approach is that one automatically inherits the features of the host language. Using the strong static typing of Haskell enables an embedding which ensures type-safety of the models at construction-time, thus trapping such errors as early as the modelling phase. When new models are defined, the types of the combined process fragments and basic modelling elements are checked, and other undefined types are inferred by the embedded type system. Thus, while the input and output types of tasks are explicitly defined by the modeller, the types of gateways such as a decision are inferred depending upon the output types of the fragments attached to its inputs, and the input types of the fragments attached to its outputs. The main constraint that should be enforced in the case of gateways is that the data types of all the incoming branches should be equivalent to the data types of its outgoing branches, for the simplest reason that the main objective of gateways is to handle the control and data-flow rather than to manipulate the data.

To infer the appropriate types and restrict constraints on the types of certain modelling elements, type classes [JJM97] provide ad-hoc polymorphism, enabling over-

loaded modelling elements to be implemented and thus carry out certain computations at the type level.

Hence, if for example, an element that expects as input some data item other than a *Customer Record* is attached to the output of the task *Get Customer Record*, the type checker would generate an error at construction-time and it would prohibit the user from carrying out other operations on that model. Once the model is defined and type checked, an internal abstract representation made up of primitive untyped constructors (with appropriate type annotations) is created. Such a deep embedded approach, allows interpretation and analysis of defined models.

Thus, when Haskell programs using the domain-specific language (such as the code shown in Listing 1) are executed, an object representing the model is generated. However, besides enabling the generation of typed models, the library also provides: operators to package sub-processes (thus supporting modularisation and compositionality), combinators to effectively combine models, and functions to transform and quality assure the processes.

3.2 Strongly-Typed Process Fragments

Due to Haskell's strong and static type system, at construction-time, all the process fragments and modelling elements must be typed. If the types of the various components are not explicitly defined, Haskell's inferencing type system tries to automatically infer the appropriate type. Haskell's type checker carries out type analysis of the process-model program and thus strictly prohibits all forms of type errors to be ignored. This is essentially what is required when defining business process models at the initial phase of the development life-cycle. To make the most out of the host, a domain-specific type system has to be defined over that of Haskell. In this way, even though users do not explicitly define the type signature (denoting the input and output types of the function) for the function representing their model, Haskell is still able to infer the correct type intended by the user and thus, always ensure a typed model. In cases where an untyped model is desirable, one can always use a generic `Data` type, which places no constraints at all, but which may be later refined as the model is specialized.

Consider for instance task *Update Customer Record* defined in Listing 2 for the model in Figure 1. The type that the compiler infers for this task is defined by the following type signature:

```
updateCustRec ::
        (PF Order, PF CustRec) -> PF Order
```

`PF Order` and `PF CustRec` represent the actual higher-order types for a process fragment outputting a user-defined business item of type *Order* and a process fragment outputting a user-defined business item of type *Customer Record*. The correct inferencing of the input and output types of this task is possible through the type signature defined for the basic construct in the embedded language, that is:

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

The first type (`String`) is used to represent the name assigned to the task. The second (`(PF a, PF b)`) and the third (`PF c`) types are used to respectively represent the inputs and outputs of the task. `PF a`, `PF b` and `PF c` are all parameterized polymorphic types, whereby Haskell's compiler would bind the generic types `a`, `b` and `c`, to any other defined types — in the case of task `updateCustRec`, `a` is bound to type *Order*, `b` to type *Customer Record* and `c` to type *Order*. These types might be specific to a business domain and thus, before used, the required higher-order types have to be defined for that domain. Since we want to ensure that the generic types can be manipulated and perceived as business process modelling types and thus used accordingly, type classes are used to provide the required ad-hoc polymorphism and to constrain the actual type of such generic types. For this reason, the actual type signature for the construct `task` in the embedded language is defined as:

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

The part of the type signature before the implication mark (`=>`) gives constraints on the polymorphism of the types, while what follows is the actual type. The constraint in this case is that generic types `a`, `b` and `c` are all instances of the type class `DataType`, which is defined in the embedded language. Since this type class collects all the business domain-specific types, when a new type is required, example *Order*, the user is expected to add this new datatype to `DataType` by simply defining it as an instance of the class as shown below:

```
newtype Order = Order ()
instance DataType (Order)
```

By embedding the domain-specific type system in that of the host as illustrated above, Haskell's inferencing system would be able to infer the correct and user-intended type, even though a type signature is not explicitly defined. It is then also possible to ensure typed models and use Haskell's type checker to ensure the appropriate application and use of constructs in the embedded language.

To allow for various interpretations and analysis of the defined models, a deep embedded approach is adopted such that, once the model is type checked, an internal abstract representation made up of primtive untyped constructors (with type annotations) is created and then used.

In the domain-specific library, this is possible through the definition of a polymorphic type `PF a` defined as

```
newtype PF a = PF PrimitivePF
```

and the definition of the abstract recursive data type `PrimitivePF` which is partly defined as

```
data PrimitivePF = ConstantValue Value
                 | Task String PrimitivePF
                 | ...
```

Thus, the abstract representation for the task `updateCustRec` would internal be defined as

```
updateCustRec = Task "Update Customer Record"
                     (x takeOrder,
                      y getCustomerRecord)
```

where x refers to the *internal definition of task* `takeOrder` and y refers to the *internal definition of task* `getCustomerRecord`.

The parameterized polymorphic data type `PF a` is essentially a phantom type [Rhi03] usually consisting of one constructor (as in this case), whose instances (in this case, `PF PrimitivePF`) are independent of the type variables (in this case, `a`). Thus the solely purpose of such type variables is to express a type constraint.

In this way, phantom types are used to construct an effective type system which acts as an additional layer on top of the primitive untyped constructors. This would ensure the production of type-safe models as early as construction-time and thus trap errors at the initial phases of the development life-cycle. Moreover, since the primitive untyped constructors are defined within an abstract data type in the host language, they act as first class objects in Haskell and thus they facilitate the interpretation and analysis of the defined model.

3.3 Packaging Models into Sub-Processes

The representation of models as functions automatically gives a modularization mechanism to modelling. Moreover, referential transparency in functional programming languages, such as Haskell, also ensures that processes are safely used within other models.

Processes defined in this way provide the right abstraction for a modeller to reason about and describe a complex model made up of a number of other process fragments. However, compilation of the Haskell code describing the model reduces it to an internal representation made up of primitive constructs and thus, the modularization in the original description is lost. This is a major issue in embedded languages and to mitigate, one typically has to use

explicit block marking mechanisms. In our case, we resolve this issue be adopting a model tagging approach.

If an activity in a business process does not have at least one data input, a control input (as in task `takeOrder`) is required to start off the execution of that activity. Similarily, if an activity does not produce any data as output, control should be returned and passed on to the next modelling element in the process. Thus, by representing the model in Figure 1 as a function as shown in Listing 1, the process `orderHandling` would have two inputs (a control and a data input of type *String*) and produces two outputs (a data output of type *Package* and a control from the stop node). However, noting that a data flow implicitly contains control, the inputs and outputs of a packaged sub-process can be reduced such that the model in Figure 1 would simply have one input (of type *String*) and one output (of type *Package*). When a model is tagged, the definition is modified such that the required input control flows are obtained from an input data flow and the returned control flows are merged into an output data flow. By eliminating unnecessary inputs and outputs, modellers are enabled to easily and more flexibly reuse sub-processes to create more complex models.

3.4 Connection Patterns and Parameterized Models

To compose complex domain-specific programs, connection patterns have been widely used in embedded languages for domains such as hardware description. Using such patterns, the end-user does not need to explicitly refer to the actual inputs and outputs, which act as connectors, as in Listing 1. Instead, through the combined use of higher-order functions and functional application, a connection pattern such as `->>-` enables the combination of two processes in sequence by connecting the output of the first to the input of the second. Such an approach enables model construction to be closer to a visual approach, in such a way that textual definitions depict precisely the flow and execution order of components as in visual definitions. For instance, `-|-` clearly depicts parallel processes and `->>-` serially composes processes such that the output of the first is connected to the input of the next. In this way, considering the model in Figure 1, its textual construction in Listing 3 would be more readable and easier to comprehend than the definition in Listing 1.

Frequently, when building large models, one identifies patterns which one could conceptually generate using a simple algorithm. Since the host language sits above the modelling language, an embedded language approach enables the user to define functions producing large, regular models. These functions represent a family of similar structured processes such that, given a specific input, a particular model in the family is automatically generated. Furthermore, the input parameters of these functions can include other models, such that various process fragments can be combined together in any specific and required manner. This is in fact one of the most effective features of embedded languages.

Consider the composition of multiple *fork-joins* as in Figure 2. This can easily be defined using a parameterized model as shown in Listing 4. To enclose `n` of these *fork-join* fragments between a *decision* and a *merge*, a parameterized model such as `decisionMerge_forkJoins`

Listing 3: Defining the order handling process in Figure 1 using connection patterns in our language

```
orderHandling = (takeOrder -|- getCustRec)
                ->>- updateCustRec
                ->>- soundCycle  reduceItemFromStock
                                ("No More Items?",
                                 (branch_NoMoreItems 0.5,
                                  branch_MoreItems 0.5))
                ->>- preparePackaging
                ->>- stop
```

Listing 4: A parameterized model to define models such as Figure 2

```
fork_joins [procsInFJ] = fork_join procsInFJ
fork_joins (procsInFJ:procInFJs) = (fork_join procsInFJ) -|-
                                   (fork_joins procsInFJs)

decisionMerge_forkJoins name branches procsInFJsList =
    decision_merge name branches (fork_joins procsInFJsList)
```

Listing 5: Defining the model in Figure 2 using `decisionMerge_forkJoins` (in Listing 4)

```
proc =
  decisionMerge_forkJoins
         "How Pay?"
         (branch_CreditCard 0.5, branch_Cash 0.5)
         [(swipeCard, signReceipt, recordCardHolderDetails),
          (countMoney, issueCardReceipt)]
```

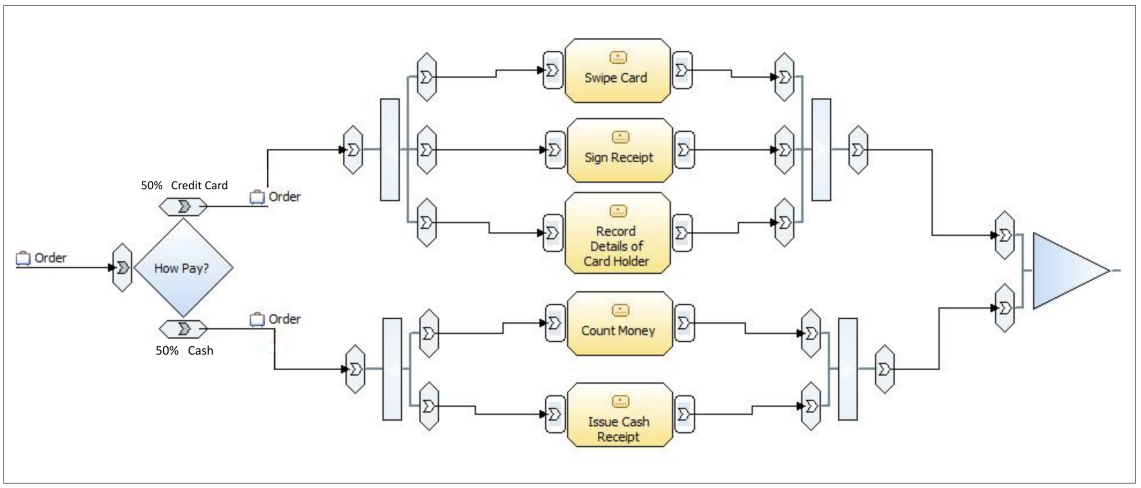

Figure 2: A *decision-merge* with internal *fork-joins*

(in Listing 4) can be defined. Doing so, then the complex model in Figure 2 can rapidly and safely be defined by the concise definition in Listing 5, where `decisionMerge_forkJoins` is invoked by specifying the name of the *decision*, the properties for the decision branches and the list of tasks that should be enclosed by the *fork-join* fragments.

4 Model Transformations and Quality Assurance

Besides generating the required domain-specific programs, embedded languages also enable users to manipulate, analyse and verify them. Since these generated programs are essentially first class data objects in the code of the host language, users can carry out operations on them by essentially defining functions to obtain the required behaviour.

In our case, transformation and quality assurance functions are particularly useful to assist modellers in the production of high quality business processes. While transformations help modellers to transform the current *'as is'* to the future *'to be'* models, quality assurance of processes at the modelling phase is essential in BDD to trap control and data-flow errors such as deadlocks (whereby a process waits indefinitely for some data or operation to complete) and lack of synchronization (whereby a process receives more than one control and thus executes more than once). These quality assurance checks are implemented as Haskell functions which analyse the model using standard functions.

For instance, some of the basic transformations that modellers might frequently use include *substitution* or *renaming* of modelling elements in the process. Modellers might also need to check some trivial properties about a model, for instance, whether the model *contains* a particular named activity, gateway or event and whether a particular *name has already been*

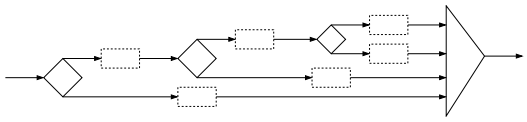

Figure 3: Triangular models made up of *decisions* and a *merge*

assigned to an element in the model. By providing such basic transformations and checks in the embedded language, the modeller is enabled to define one's own transformation as a function and then, through the use of higher-order functions and functional application, apply the transformation on the required generated model to produce a new one.

Different from the current visual modelling tools, using an embedded language, modellers are not limited to a set of pre-defined transformations. Instead, they are free to define their own sequential, branching and iterative transformations as first class objects in the host language, as illustrated in Listing 6. In this example, the branching transformation `trans1` and the simple transformation `trans2` are carried out in sequence, such that the required changes are carried out on the elements of the process. Thus, on execution of `trans1`, if the model contains a sub-process named *Order Verification*, then this is renamed to *Certify Order*. Else, if it contains a task named *Reject Order*, then this is substituted with another task called *Apply Special Terms to Order*. On completion of the first transformation, the second one is carried out and thus, the decision *Is Order Valid?* is renamed to *Is Order Certified?*.

Before and after transformations are applied, algorithms to check the soundness of models (to detect deadlocks and lack of synchronization) are applied, thus ensuring quality assured composite model transformations. By analysing different models and patterns identified in [KV07a, KV07b, RtHvdAM06, RtHEvdA04], functions such as `isSound` were provided in the library to try to identify patterns and anti-patterns and thus to detect soundness. If this is not possible, then soundness is not determined by such functions, and complete-state analysis techniques would have to be used. To help modellers with the definition of high quality processes, some of the most complex models, which modelling researchers (such as Koehler and Vanhatalo in [KV07a] and [KV07b]) identified as the most common modelling errors, are provided as pre-defined process fragments or parameterized models. Some of these include models to construct *sound cycles*, which in our language is provided as a pre-defined process fragment named `soundCycle` (as seen in Listing 3), and functions to construct triangular models made up of a number of `decision` constructs and one `merge` construct such as in Figure 3.

Using the embedded language approach, connectors are not explicitly defined as modelling elements and thus modellers can only traverse the generated model top-down. Carrying out analysis on such a model is not really feasible and intuitive. To mitigate this issue, a directed graph defined as a set of vertices and edges can be generated. This graph would flexibly and more intuitively enable the modeller to carry out the required analysis on the model. Knowing the initial and ending vertices, the modellers would also be allowed to

Listing 6: Defining the quality assured composite transformation `transOrderProcessing`

```
applySpecialTerms =
       task "Apply Special Terms to Order"
            (type_Order :-> type_Order)

transOrderProcessing proc x =
 let

   hasSubProcOrderVerif =
               containsSubProcess "Order Verification" proc
   hasTaskRejectOrder =
               containsTask "Reject Order" proc

   trans1@(wasTransDone, transProc) =
          if (hasSubProcOrderVerif)
             then (renameSubProc "Order Verification"
                                 "Certify Order" proc x)
          else if (hasTaskRejectOrder)
             then (substituteTask "Reject Order"
                                  applySpecialTerms proc x)
          else (Succeeded, proc x)

   trans2 = renameDecision "Is Order Valid?"
                           "Is Order Certified?" transProc

 in  trans2
```

traverse the model either top-down or bottom-up.

5 Evaluation and Case Studies

A number of models created with WSBM were used for our preliminary case studies. These models were constructed using different approaches and each one was analysed.

The first two case studies are based on two models obtained from the sample projects that are available with IBM's tool. These projects 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 evaluate our first prototype and to help modellers learn how to define real world processes in our language. In fact, these sample projects are also provided as samples in our domain-specific library.

The process in the first case study dealt with auto claims handling in a claims assessor management system of an automobile insurance company. Our main aim was to analyse the different ways how models and modelling elements can be defined using our language. We also tried to identify which of these methods would be most feasible and convenient to use for a modeller who is more acquainted to a visual modelling language. We managed to illustrate the ease at which complex processes can be defined, without the need for any knowledge of the host language Haskell and the underlying type system which carries out all the type-safety checks of the models at construction-time. The definitions using connection patterns proved to be more comprehensible and compact.

In the second case study, a sophisticated business process that handles customer orders, from the point the order is received upto the point when the items are reduced from the stock and dispatched, was considered. An interesting feature of this process is that it contained links to various data repositories and previously user-defined sub-processes. The main aim was to identify how easy a complex model can be defined with the least amount of effort, components and expertize, while still ensuring the correctness of the model. The importance and usefulness of connection patterns was particularly evident. Such patterns ensured the right abstraction to construct models which are more readable, comprehensible and simpler to reason about. Some of the process fragments which are commonly modelled incorrectly and which are pre-defined in our domain-specific library were also used and doing so, less effort and expertize was required to correctly define the model.

Noting the importance of connection patterns when handling complex models, a different order handling process was intentionally constructed to illustrate how most of the connection patterns, defined in our domain-specific library, could be used. Two semantically equivalent definitions for the same process were defined. In contrast to the first definition, the second made use of connection patterns and some of the provided pre-defined parameterized models. Comparing the two, it was evident that besides being more concise, the second definition was easier to construct, more comprehensible and required less effort to ensure its correctness.

Finally, two examples of parameterized models were investigated in the fourth case study. The main aim was to illustrate different ways how modellers can possibly define their own parameterized models and convey examples of situations where these would be useful when defining a business process. The model in Figure 2 was used as one of these examples. With this study, we illustrated that different from WSBM, modellers are not limited to a simple recording feature that carries out operations in sequence. More than this, modellers are free to identify their own commonly used fragments and define them as parameterized models. Doing so, then the definition of the processes that refer to such models would be more comprehensible and modular, abstracting away unnecessary details which can easily be defined in a parameterized model.

Even though these are just preliminary case studies, it is already evident that using an embedded language approach, various business process models can rapidly be constructed in a concise and readable manner. This is possible through the use of connection patterns and parameterized models which provide the right modularity and abstraction mechanisms to effectively define the required model. Moreover, the produced models are type-safe and most of the errors are identified at construction-time, such that, they are trapped at the modelling phase and are thus not allowed to propagate to the succeeding stages in BDD life-cycle. To effectively prove the benefits of our approach, we still need to carry out more elaborate empirical studies, possibly involving various domain experts from various areas and with various skills.

6 Related Work

To assist modellers, over the years, various languages and tools such as WSBM have been developed. The most recent is Business Process Modelling Notation (BPMN) [Obj08]. Although its main objective is to unify the features of all the other languages and promote the use of just one standard notation, still, similar to the other previous languages, it does not adopt a functional approach based on higher-order logic. Thus, our language is the first modelling language (to specifically model business processes) of this sort.

As argued in [KGK+07], a *declarative approach* would be appropriate to define composite transformations and pre and post conditions that assure the quality of the produced models. In [KHSW05], pre and post conditions of out-place transformations were represented in the Object Constraint Language and were successfully used to refine the models into the executable BPEL code. However, such an approach brings about other advantages. Noting how effectively certain features in Haskell [Jon03] were used to define circuits [She05] and other domains, we were inspired to use Haskell as our host and thus define models as functions. In this way, modellers can declaratively define composite transformations and pre and post conditions for quality assurance more effectively.

To analyse and interpret the model in an infinite variety of ways, we have adopted a *combinatorial approach* as in [JES00] whereby a combinator library in Haskell was produced

to compose financial contracts. By employing such a deep embedded approach, the basic modelling elements in our language act as combinators.

To extend WSBM, in [KGK+07], IBM presents a *model transformation framework*. Their main objective is to provide an abstract layer over the tool, such that specialized developers are able to easily define new transformations, quality assure them and integrate them into the tool. However, since it uses first-order logic, developers still need to consider the implementation of the required operations. Moreover, to carry out checks while the user is constructing or editing the model, linear-time algorithms that do not introduce any significant delay such as [VVL07] would have to be adopted. In contrast, with our approach, we are able to statically trap errors and ill-typed processes at construction-time through our embedded domain-specific type system and Haskell's type checker. These are identified before any further computation is carried out. Phantom types and type classes are used in a similar way as in [LM99] and [CSS03] to define our strongly-typed system. Besides this, specialized functions that operate on the abstract representation are provided to analyse the structural correctness of the models.

Over the years, various *quality assurance techniques* have been suggested. In [VVL07], the authors argue that if models are decomposed into Single-Entry-Single-Exit fragments, they can be quality assured more effectively by using linear-time control-flow heuristics or complete state analysis. Similarly, a set of patterns and anti-patterns have been identified in [KV07a] and [KV07b]. A number of evaluation criteria for workflow patterns were also presented in [RtHvdAM06] and [RtHEvdA04]. To help modellers rapidly and safely transform the current *'as is'* to the future *'to be'* models, in-place model transformations must be combined with quality assurance techniques. Even though IBM's framework enables programmers to define such transformations, it is still based on first-order logic and thus, it not possible for the modellers themselves to create composite branching and iterative transformations and to define pre and post conditions that quality assure them. Our language is based on higher-order logic and thus the users can declaratively define sequential, branching and iterative composite transformations and the required pre and post conditions.

7 Conclusion

Through the use of embedded language techniques, we have discussed an approach and developed a language which is able to capture the domain of business process modelling and allow the design, modelling, transformation and quality assurance of business processes in BDD. Connection patterns play an important role to ensure that the definitions of models are readable, easy to comprehend and type-safe. Unlike visual modelling tools, users are able to define their own parameterized models and transformations. 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. Quality assurance can be combined to model transformations and by using the generated directed graph for the model, users can

easily analyse the processes. Tools can be used in conjunction to the embedded language and added as plug-ins to standard business process modelling tools, to facilitate construcion, analysis and transformations of models in the language. We now plan to link our language to a model checker to carry out complete state analysis of the constructed models and thus, assure the quality and thorough evaluation of various models.

Since our language is embedded in Haskell, we were able to adopt a functional approach and inherit the infrastructure, tools and features of the language without necessarily having to re-implement them. Various models have been defined in our language to ensure that our objectives were achieved.

We are aware that some modellers might still prefer to work with a graphical representation and thus, we started working on some functionality to convert a textual definition in our language to a visual representation in WSBM. In a similar manner, it would also be useful to provide some functionality to transform a visual representation of the model to a textual one in the embedded language. Doing so, this embedded language approach would be linked to a visual modelling tool and thus, a hybrid of scripting and visual editing approach would be adopted to modelling.

Acknowledgment

Thanks to Jana Koehler (IBM Zurich Research Laboratory) for providing us with the required information and resources to work with IBM's WebSphere Business Modeler Advanced.

References

[CSS03] Koen Claessen, Mary Sheeran, and Satnam Singh. Functional hardware description in Lava. In *The Fun of Programming*, Cornerstones of Computing, pages 151–176. Palgrave, 2003.

[HJ94] Paul Hudak and Mark P. Jones. Haskell vs. ada vs. c++ vs awk vs ... an experiment in software prototyping productivity. Technical report, 1994.

[Hud98] Paul Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings of the 5th International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[JES00] Simon Peyton Jones, Jean Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *ICFP '00: Proceedings of the 5th ACM SIGPLAN international conference on Functional programming*, pages 280–292, New York, NY, USA, 2000. ACM.

[JJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: exploring the design space. In *Proceedings of the Haskell Workshop 1997*, June 1997.

[Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.

[KGK+07] Jana Koehler, Thomas Gschwind, Jochen Küster, Cesaro Pautasso, Ksenia Ryndina, Jussi Vanhatalo, and Hagen Völzer. Combining quality assurance and model transformations in business-driven development. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Proceedings of the Applications of Graph Transformations with Industrial Relevance (Agtive 2007)*, pages 1–16, Wilhelmshöhe, Kassel, Germany, 2007.

[KHK+08] Jana Koehler, Rainer Hauser, Jochen Küster, Ksenia Ryndina, Jussi Vanhatalo, and Michael Wahler. The role of visual modeling and model transformations in business-driven development. *Electron. Notes Theor. Comput. Sci.*, 211:5–15, 2008.

[KHSW05] Jana Koehler, Rainer Hauser, Shane Sendall, and Michael Wahler. Declarative techniques for model-driven business process integration. *IBM Systems Journal*, 44(1):47–65, 2005.

[KV07a] Jana Koehler and Jussi Vanhatalo. Process anti-patterns: How to avoid the common traps of business process modeling, part 1 modeling control flow. *IBM WebSphere Developer Technical Journal*, 10(2), February 2007.

[KV07b] Jana Koehler and Jussi Vanhatalo. Process anti-patterns: How to avoid the common traps of business process modeling, part 2 modeling data flow. *IBM WebSphere Developer Technical Journal*, 10(4), April 2007.

[LM99] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages 1999*, pages 109–122. ACM Press, 1999.

[Mit05] Tilak Mitra. Business-driven development. IBM developerworks article, IBM, December 2005.

[Obj08] Object Management Group (OMG). *Business Process Modeling Notation (BPMN) Specification Version 1.1*, January 2008.

[Rhi03] Morten Rhiger. A foundation for embedded languages. *ACM Transactions on Programming Languages and Systems*, 25(3):291–315, May 2003.

[RtHEvdA04] Nick Russell, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow data patterns. Technical report, Queensland University of Technology, Brisbane, Australia, April 2004.

[RtHvdAM06] Nick Russell, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, and Natalya Mulyar. Workflow control-flow patterns: A revised view. Technical report, BPMcenter.org, 2006.

[She05] Mary Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158, July 2005.

[VVL07] Jussi Vanhatalo, Hagen Vlzer, and Frank Leymann. Faster and more focused control-flow analysis for business process models through sese decomposition. In Bernd J. Krmer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Proceedings of Service-Oriented Computing ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 43–55. Springer, 2007.