Cross Viewpoint Consistency in Open Distributed Processing (Intra language Consistency)

Prepared by Eerke Boiten, Howard Bowman, John Derrick, Maarten Steen

Computing Laboratory
University of Kent at Canterbury
Canterbury
Kent CT2 7NF

Phone +44-1227-764000
Fax +44-1227-762811
Email (E.A.Boiten,H.Bowman,J.Derrick,mwas)@ukc.ac.uk

Contents

Chapter 1

Introduction

1.1 Introduction

Open Distributed Processing (ODP) is recognised as an important standardisation activity. The ODP model seeks to provide an architecture for building potentially global distributed systems with components from many vendors. Thus, ODP will realise the open systems ethos in the distributed systems domain.

A central concept in ODP is that of a viewpoint. Distributed systems are viewed to be so complex that a process of separation of concerns must be employed when describing such systems. Viewpoints provide such a separation of concerns by presenting five distinct views of a single system; these are the *enterprise viewpoint*, *information viewpoint*, *computational viewpoint*, *engineering viewpoint* and *technology viewpoint*.

It should be clear that in such viewpoint models it is essential that specifications in different viewpoints are related in order to determine whether the multiple specifications impose conflicting requirements. The project being reported here responds to these needs by investigating how to check that multiple viewpoint specifications are in some sense *consistent*.

The objective of the project is to perform a verification of the concept of cross viewpoint consistency checking. We will determine the feasibility of performing such checks by developing prototype techniques and tools for relating viewpoint specifications written in Z and LOTOS. These two languages have been chosen because they are formal; enabling formal reasoning to be applied, which we argue is an essential prerequisite for successful consistency checking. In addition, Z and LOTOS represent two of the most different specification techniques being advocated as viewpoint languages, thus, consistency checking between these two languages bounds the difficulty of the problem.

This deliverable describes the initial phase of our work; it focusses on consistency checking methods for individual FDTs. A second deliverable will be produced which extends this work by developing cross-language consistency checks between Z and LOTOS.

1.2 Overview of Project

Figure 1.1 depicts the work plan for the project. Depicted is the full three year trajectory of the project. The two milestone deliverables for BT are shown as bold tasks.

The project is divided into 5 workpackages:

1. **WP1: Consistency Framework**
   The objective of this workpackage is to develop a general framework for consistency in ODP. This will define the basic concepts and mechanisms involved in consistency in a general, FDT independent, fashion. Particular instantiations of the framework can be made by substituting specific FDTs and their correctness properties into the general framework. An important

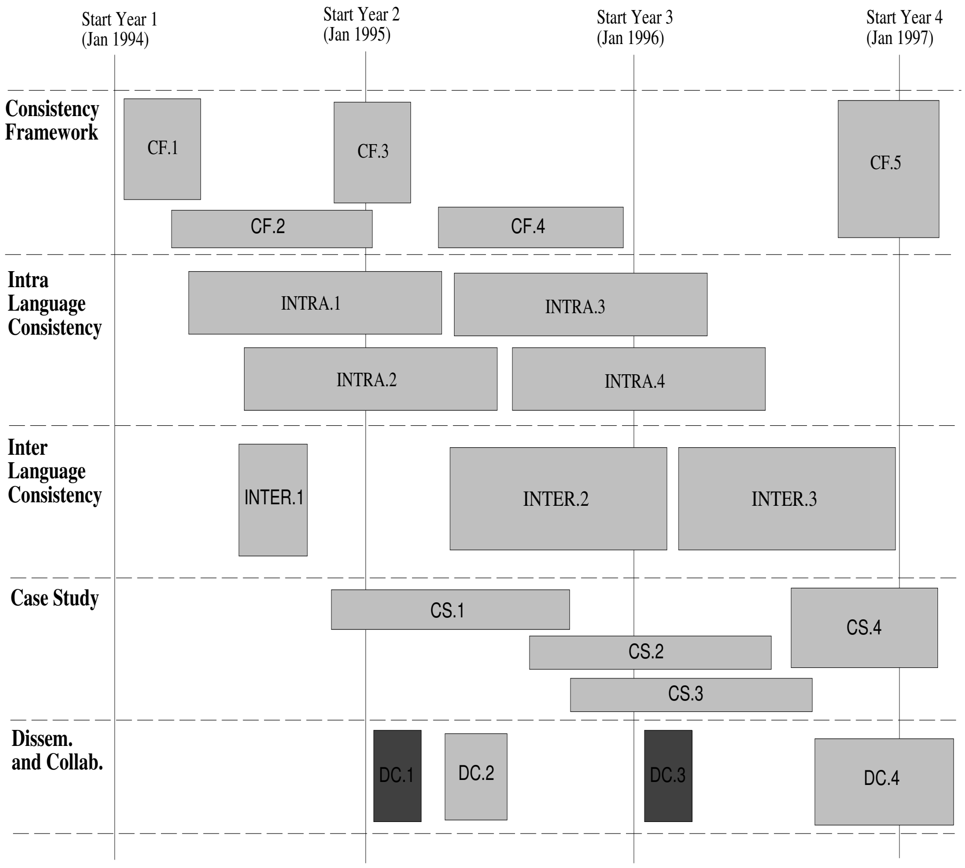

Figure 1.1: Plan of Work

role of the framework is to ensure that consistency is treated uniformly in each FDT. The framework will provide an interpretation of the main consistency concepts: *consistency*, *unification*, *translation* etc and will define general strategies for consistency checking. The properties of these concepts and strategies will be determined and classified. These properties will highlight the character of the consistency checking problem in ODP. The framework will necessarily be formal in nature.

2. **WP2: Intra Language Consistency**
   Consistency checking between specifications in the same language will be investigated in this workpackage. Specifically, the consistency checking concepts and strategies developed in the previous workpackage will be instantiated for the two formal specification languages, Z and LOTOS. These strategies will be realised by the development of consistency checking tool support for both languages.

3. **WP3: Inter Language Consistency**
   Consistency checking between specifications in different languages will be investigated in this workpackage. As a verification of concept, cross language consistency checking between Z and LOTOS will be explored. This is an extremely demanding area for which there are currently few positive research results. It is intended that tool support for such Z and LOTOS inter language consistency checking will be developed.

4. **WP4: Case Study**
   In this workpackage the suitability of the techniques and tools developed in the previous

workpackages will be assessed against a number of case studies. A significant part of this workpackage will involve locating suitable examples of ODP systems which can act as case studies. These example ODP specifications must be in multiple viewpoints and have both Z and LOTOS viewpoint specifications.

5. **WP5: Dissemination and Collaboration**
   Dissemination of the results of the project will be targetted at three main groups: the ODP standardisation community, interested industrial parties and the distributed sytems research community. Dissemination to the first two of these groups will be facilitated by our collaboration with B.T. under the Formosa project. Two deliverables will be produced for B.T.; the first describing our results on intra language consistency and the second describing our results on inter language consistency. In addition, contributions will be made directly to the ODP standardisation forum through the BSI and to the research community through papers at major conferences and in learned journals.

Each of the five workpackages is divided into tasks. We list these here. The role of each task should be evident from the tasks title.

WP1, CF.1: Initial formulation of Consistency Framework
WP1, CF.2: Study of RM-ODP Definitions
WP1, CF.3: Refined Consistency Framework
WP1, CF.4: Location of Correspondence Rules
WP1, CF.5: Consistency Framework - Final Revision

WP2, INTRA.1: Z Consistency Techniques
WP2, INTRA.2: LOTOS Consistency Techniques
WP2, INTRA.3: Z Consistency Tool
WP2, INTRA.4: LOTOS Consistency Tool

WP3, INTER.1: Preliminary Study of Potential Approaches
WP3, INTER.2: Z and LOTOS Consistency Techniques
WP3, INTER.3: Z and LOTOS Consistency Tool

WP4, CS.1: Location of Possible Case Studies
WP4, CS.2: Z to Z Case Studies
WP4, CS.3: LOTOS to LOTOS Case Studies
WP4, CS.4: Z and LOTOS Inter Lang. Consistency Case Studies

WP5, DC.1: First BT Deliverable
WP5, DC.2: Input to RM-ODP Part 1
WP5, DC.3: Second BT Deliverable
WP5, DC.4: Final Deliverable and Recommendation to ODP

1.3 Overview of deliverable

This deliverable has the following structure:-

- **Chapter 1: Introduction.** This first chapter introduces the project and describes the structure of the deliverable.
- **Chapter 2: Background.** Background on the ODP initiative and the role of viewpoints within this work is presented in this chapter. Three aspects of ODP are considered with reference to the viewpoints model: the architectural semantics, system development for ODP and conformance assessment.

- **Chapter 3: Definitions of Consistency.** The consistency framework developed during the first phase of the project is presented in this chapter. Central to this framework is a precise definition of consistency. We argue that this definition is general enough to embrace all the interpretations of consistency that have already been proposed within ODP. In particular, we show how our definition of consistency can be related to all the interpretations of consistency in the RM-ODP.

- **Chapter 4: Consistency in LOTOS.** Consistency checking within LOTOS is investigated in this chapter. We present a number of possible LOTOS instantiations of the framework concepts and then relate these instantiations. In addition, we present specific mechanisms for consistency checking in LOTOS and give an example of a LOTOS consistency check.

- **Chapter 5: Consistency in Z.** This chapter describes the work on consistency checking in Z. A general algorithm for unifying two Z specifications is presented. Consistency checking by validating the implementability of the derived unification is also explored. An example of unification and consistency checking of two Z specifications is presented.

- **Chapter 6: Conclusions.** The deliverable is summarised and concluded in this chapter.

Chapter 2

Background

In this chapter, some of the developments in the light of which the research project described in this deliverable should be seen, are high-lighted. Our research was mainly triggered by the progressing standardisation of the Reference Model for Open Distributed Processing (RM-ODP). In section 2.1, a brief overview is given of the ODP concepts most relevant to this deliverable. This project is also closely related to the DTI and EPSRC funded FORMOSA project between BT and the University of Stirling. The major aim of the FORMOSA project is to advance the formulation of architectural semantics for the ODP standards using the Formal Description Techniques (FDTs) LOTOS and Z. In section 2.2, we identify the relationships between our work on consistency checking and the work on defining an architectural semantics for ODP, such as in the FORMOSA project. Another project that influenced our research on consistency checking is the PROST project.

2.1 The Reference Model for Open Distributed Processing

The standardisation of a Reference Model for Open Distributed Processing [30] is a joint effort of the International Standardisation Organisation (ISO) and the International Telecommunication Union (ITU-T). The objective is to enable the construction of distributed systems in a multi-vendor environment through the provision of a general architectural framework that such systems must conform to. One of the cornerstones of this framework is a model of multiple viewpoints which enables different participants each to observe a system from a suitable perspective and at a suitable level of abstraction [37]. Section 2.1.1 deals in more detail with the ODP viewpoints.

2.1.1 Viewpoints

The complete specification of any non-trivial distributed system involves a very large amount of information. Attempting to capture all aspects of the design in a single description is generally unworkable. Most design methodologies aim to establish a coordinated, interlocking set of models each aimed at capturing one facet of the design, satisfying the requirements which are the concern of some particular group involved in the design process.

In ODP, this separation of concerns is established by identification of five *viewpoints*, each with an associated *viewpoint language* which expresses the concepts and the rules relevant to a particular area of concern.

The viewpoints defined in the Reference Model for ODP are: Enterprise viewpoint, Information viewpoint, Computational viewpoint, Engineering viewpoint and Technological viewpoint, see figure 2.1.

The **enterprise viewpoint** is concerned with business policies, management policies and human user roles with respect to the systems and the environment with which they interact. The use of the word enterprise here does not imply a limitation to a single organisation. The model constructed may well describe the constraints placed on the interaction of a number of distinct

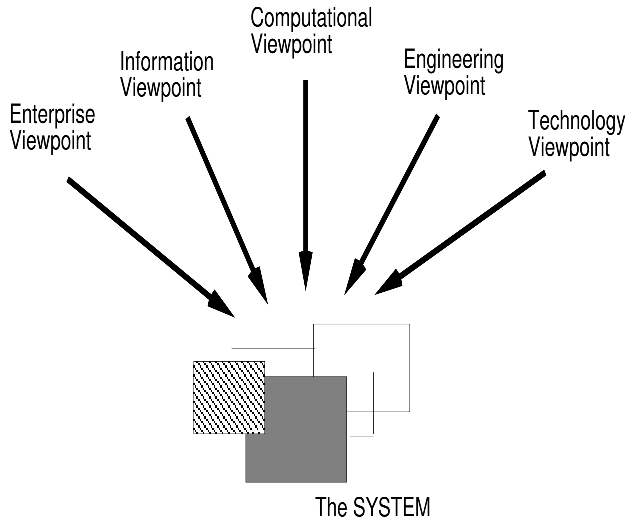

Figure 2.1: The Multiple Viewpoints Model

organisations. The **enterprise language** introduces concepts to support the expression of policy, particularly with regard to agreements and responsibilities between parts of the enterprise. Agents perform actions and artifacts represent resources. Agents can be assigned roles in a contract which expresses permissions or prohibitions. Groupings of agents are considered as communities which may be administered by a particular agent.

The **information viewpoint** is concerned with information modelling. By factoring an information model out of the individual components, it provides a consistent common view which can be referenced by the specifications of information sources and sinks and the information flows between them. The **information language** defines concepts for information schema definition. The language distinguishes between an instantaneous view of information (a static schema), a statement of information which is necessarily unchanged by the system (an invariant schema) and a description of information reflecting the behaviour and evolution of the system (a dynamic schema).

The **computational viewpoint** is concerned with the algorithms and data flows which provide the distributed system function. This viewpoint specifies the individual components which are the sources and sinks of information flows. The **computational language** enables the representation of the system and its environment in terms of objects which interact by transfer of information via interfaces. Interfaces are given types and rules are defined for the matching of these types, so that object interfaces need not be specified in an identical way in order to enable interaction between the objects. Objects are chosen to achieve a functional decomposition, but also identify the candidate boundaries for physical distribution.

The **engineering viewpoint** is concerned with the distribution mechanisms and the provision of the various transparencies needed to support distribution. The **engineering language** defines a number of functional building blocks which can be combined together to provide the requested transparencies.

The **technology viewpoint** is concerned with the hardware and software components from which the distributed system is constructed.

Requirements and specifications of an ODP system can be made from any of these viewpoints (as depicted in figure 2.1). However, these viewpoints are not independent. They are each partial views of the complete system specification. Some entities can, therefore, occur in more than one viewpoint, and there are a set of *consistency constraints* arising from the correspondences between terms in two viewpoint languages and the statements relating the various terms within each language. The checking of such consistency is an important part of demonstrating the correctness of the full set of specifications.

2.2 Architectural Semantics for ODP

Each viewpoint language consists of a set of definitions and a set of rules which constrain the ways in which the definitions can be related. The notion of language used is an abstract one; the rules are, in effect, the foundations for the grammar of a set of possible detailed languages or notations.

The reference model is not prescriptive in the choice of a specific notation; rather the intention is that a number of existing notations will be used as viewpoint languages, by supporting the concepts and rules defined in the RM-ODP. To this end, a clear interpretation of the architectural concepts of the reference model should be available for those formal notations.

The need for an architectural semantics was recognised from the start of the work on the ODP reference model and is reflected by the inclusion of the architectural semantics as Part 4 of the standard. RM-ODP Part 4 provides an interpretation of the ODP modelling and specification concepts in LOTOS, Estelle, SDL and Z. Thus, this work will act as a bridge between the ODP model and the semantic models of the FDTs and will enable formal descriptions of standards for ODP systems to be developed in a sound and uniform way.

In order to achieve these requirements, the architectural semantics should be consistent in two ways. Firstly, it is necessary to demonstrate that the interpretations of the same architectural entity in different FDTs are consistent. Secondly, the architectural semantics of different viewpoints are related and should therefore be checked for consistency.

The architectural semantics will also provide the basis for uniform and consistent comparison between formal descriptions of the same system or standard in different FDTs. It is, therefore, of great significance to realistic consistency checking techniques.

2.3 ODP System development and conformance assessment

The RM-ODP provides a general architectural framework for the specification of open distributed systems. It does not prescribe a particular system development methodology. The PROST project has investigated a general system development strategy for ODP, which is outlined in section 2.3.1.

Within the framework, domain specific ODP standards can be formulated. When particular distributed systems conform to ODP standards, this will guarantee interoperability and exchangeability of components. Conformance assessment for ODP is discussed in section 2.3.2.

2.3.1 System development

A number of development strategies could be envisaged for ODP. The multiple viewpoint approach to specification puts particular requirements on the system development methodology. Each viewpoint specification is, at least potentially, at the same level of abstraction; suggesting that viewpoints are related horizontally relative to a vertical system development. This is in contrast to classic waterfall development methodologies. In the PROST project such a, fully general, system development methodology has been investigated. The general development scenario is depicted in figure 2.2. The methodology promotes a number of specification to specification transformations, such as *translation*, *refinement* and *unification*, with the aim to derive a composite 'implementation' specification from the multiple viewpoint specifications.

Translation maps a specification from one language onto another while maintaining semantic equivalence. Refinement has the usual meaning of making a specification less abstract, and thus bringing it closer to an implementation while maintaining the captured requirements. Unification is a transformation which enables specifications to be combined.

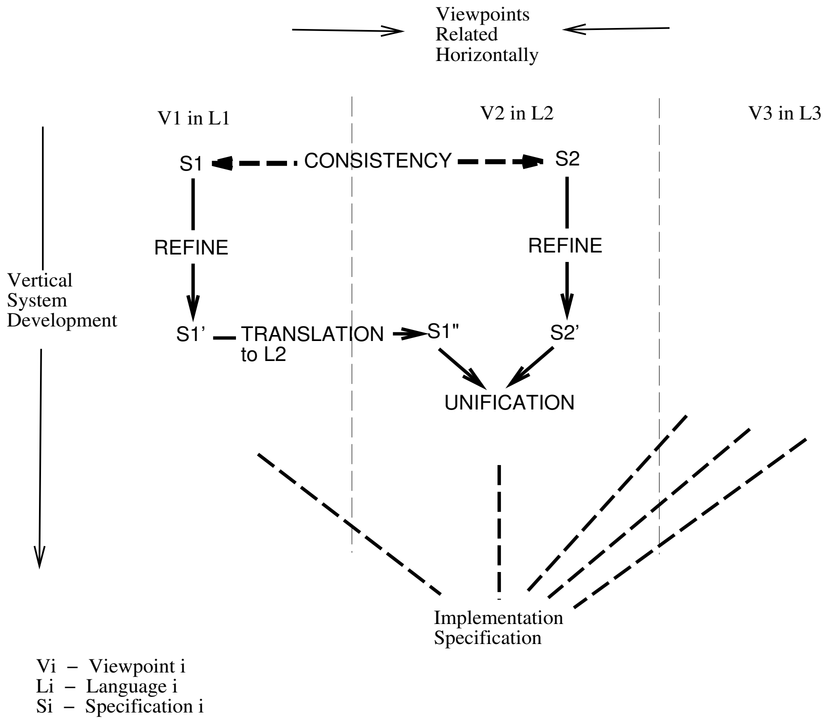

Figure 2.2: PROST System Development Scenario

Consistency is implicit in such a system development methodology. For example, two specifications would be viewed as inconsistent if a common unified specification did not exist. Thus, consistency arises during unification of specifications in models of ODP system development.

2.3.2 Conformance assessment

Conformance assessment has been considered from early on in the work on ODP. The meaning of conformance has been built into the RM-ODP. An ODP system conforms to an ODP standard if it satisfies the conformance requirements of that standard.

Conformance assessment of distributed systems is potentially more complex and costly than in traditional communications protocols. The complexity inherent in a distributed system infrastructure may require that in addition to direct testing, techniques such as verification and validation may be necessary to achieve sufficient confidence in the conformance of a particular product.

Results from the PROST project indicate that conformance assessment for distributed systems can be divided into two categories of activities: *conformance testing* and *specification checking*. Conformance testing is the activity that relates real implementations to specifications by applying a series of tests to the implementation that were derived from the specification. Specification checking involves activities such as validation, verification and consistency checking. Both validation and verification apply to specifications within one viewpoint. Validation checks a specification against its requirements. Verification checks a refinement against its specification. Consistency checking, potentially, relates specifications across viewpoints. Obviously, no product can conform to a set of viewpoint specifications that are inconsistent.

2.4 Summary

The research project on consistency checking, described in the rest of this deliverable, was mainly triggered by the progression of the standardisation of the Reference Model for Open Distributed Processing. One of the cornerstones of the RM-ODP is a model of multiple viewpoints from which ODP systems can be specified. One of the consequences of adopting a multiple viewpoint approach is that descriptions of the same or related objects can appear in different viewpoints and must co-exist. *Consistency* of specifications across viewpoints thus becomes a central issue.

The ongoing work on the definition of architectural semantics for ODP manifests important input to the work on consistency checking as it provides a formalisation of the ODP viewpoint languages. Conversely, techniques for consistency checking can also prove valuable in validating the architectural semantics.

Although the RM-ODP does not prescribe a particular system development methodology, it is clear that the multiple viewpoint approach to specification requires a horizontal relating of viewpoint specifications with respect to a vertical development. Several transformations of specifications may be required, such as translation and unification, which also play a role in consistency checking.

Conformance assessment for ODP encompasses both *conformance testing* (i.e. relating real implementations to specifications) and *specification checking* (i.e. relating specifications to specifications). Verification of cross viewpoint consistency is an important example of specification checking.

Chapter 3

Definitions of Consistency

In order for consistency to be treated uniformly, a single basic definition of the concept must be given. We seek a fully general interpretation that can be instantiated for different languages as appropriate; e.g. can be intanstiated for the correctness properties of both Z and LOTOS.

3.1 Three Possible Definitions of Consistency

This section highlights three possible interpretations of consistency. These definitions all appear in the RM-ODP, the first two appear in part 1 (clause 12.2) and the third appears in part 3 [30] (clause 10). Although, the first of these definitions is only alluded to and the IS version of the reference model is considerably less prescriptive about the definition to use than previous drafts of the standard.

Definition 1
(1.1) Two specifications are consistent iff they do not impose contradictory requirements.
(1.2) Two specifications are consistent iff it is possible for at least one example of a product (or implementation) to exist that can conform to both of the specifications.
(1.3) Two specifications are consistent iff they are both behaviourally compatible with the other.

This last interpretation is a rewording of the RM-ODP definition. This is because the RM-ODP definition is expressed in terms of relating specific viewpoints. We are considering more generalised notions of consistency, thus, we have brought the definition into line with the other definitions in order to facilitate a direct comparison. In addition note that all these definitions are symmetric, i.e. if a specification S is consistent with a specification R then R is consistent with S. This is a reasonable intuitive requirement for a large class of consistency problems (see section 3.2.3).

Behavioural compatibility is defined as follows:

Definition 2 (Behavioural Compatibility) *A specification is behaviourally compatible with a second specification, with respect to a set of criteria, if the first specification can replace the second specification without the environment being able to notice the difference in the specification's behaviour on the basis of the set of criteria.*

This definition slightly adapts the RM-ODP presentation of this concept. Specifically, the RM-ODP definition is expressed in terms of objects. However, we would like to be more general than this and hence we have presented the concept in terms of the notion of a specification. In addition, these three consistency interpretations blur over the fact that specifications may be in different FDTs and that it may not be possible to relate specifications directly without some element of translation.

Each of these notions of consistency is intuitively reasonable. However, the question arises: what is the relationship between the interpretations and, in particular, are these definitions of consistency themselves consistent? In fact, the different interpretations are likely to be applicable in different settings. For example, definition 1 is relevant to consistency checking in a logical setting, e.g. in an FDT such as Z which is based on first order logic.

We seek to reconcile these interpretations through formalisation. We formalise the first notion of consistency as follows,

Definition 3 $S_1\ C_1\ S_2$ *iff* $\neg(\exists\,\psi\ s.t.\ S_1 \models \psi\ \wedge\ S_2 \models \neg\psi)$

where $\models$ is the satisfaction relation of the specification's logic. This definition states that two specifications are consistent if and only if there is no property that holds over one of the specifications and its negation holds over the other specification.

To interprete consistency 1.2 we need a formal interpretation of conformance. There is a difficulty here because conformance relates real physical implementations to specifications and implementations are not amenable to formal interpretation. The classical approach to handling this difficulty is to only consider conformance up to a, so called, *implementation specification.* This is a specification that describes a real implementation in as much detail that a direct mapping from the implementation specification to the real implementation can be found. Thus, it is normal just to consider conformance relations between specifications, see [10] [12] [36] for typical approaches. However, implementation specifications relate to real implementations in different ways for different FDTs and, in particular, for some FDTs not all implementation specifications are implementable. This would, for example, be the case for Z, see discussion in section 3.2.2.

Our approach then is to divide conformance testing into two parts. Firstly, we consider conformance up to implementation specifications, using a relation $conf \subseteq SPEC \times SPEC$, and then we consider conformance of implementation specifications to real implementations, using a relation $Conf \subseteq IMP \times SPEC$ [1], where $SPEC$ is the set of possible ODP specifications and IMP is the set of possible ODP implementations.

By way of clarification, $S_2\ conf\ S_1$ expresses the property that specification S_2 conforms to specification S_1, i.e. according to tests derived from S_1, S_2 cannot be distinguished from S_1. It should be noted that we have not specified how and what form of tests are derived from S_1; there are many options for such derivation [10] [12]. In a similar way $I\ Conf\ S$ expresses the property that I conforms to S. Interpretation 1.2 is now formalized as:-

Definition 4 $S_1\ C_{2.1}\ S_2$ *iff* $\exists\, S \in SPEC, I \in IMP\ s.t.\ S\ conf\ S_1\ \wedge\ S\ conf\ S_2\ \wedge I\ Conf\ S.$

i.e., two specifications are consistent iff an implementation specification which conforms to both and a real implementation of the implementation specification can be found. This definition is correct, but is not very useful since it uses $Conf$, which is not subject to formal interpretation. In order to resolve this difficulty we introduce the concept of *internal validity* which holds whenever a specification is implementable:-

Definition 5 *S is internally valid, denoted $\Psi(S)$, iff* $\exists\, I \in IMP\ s.t.\ I\ Conf\ S$

We will return to this notion of implementability in section 3.2.2. For example, a Z specification which contains contradictions would not be internally valid. Now we can redefine C_2 in a more usable way:

Definition 6 $S_1\ C_{2.2}\ S_2$ *iff* $\exists\, S \in SPEC\ s.t.\ S\ conf\ S_1\ \wedge S\ conf\ S_2\ \wedge\ \Psi(S).$

The third and final consistency interpretation hinges on the notion of behavioural compatibility which is defined in terms of an environment and unspecified criteria. We will consider specific instantiations of behavioural compatibility when we look at specific FDTs; at this stage we formulate the interpretation completely generally, for bc a particular instantiation of behavioural compatibility.

[1] The order of our relations is in accordance with LOTOS conventions and is opposed to Z conventions

Definition 7 $S_1\ C_3\ S_2$ *iff* $S_1\ bc\ S_2 \wedge S_2\ bc\ S_1$.

This definition is parameterised on the notion of behavioural compatibility. Thus, we will often make this parameterisation explicit and denote the interpretation as C_3^{bcs}, where bcs is a symmetric subset of bc, i.e. the consistency relation induced by bcs.

Discussion. We would like to relate these three definitions, C_1, $C_{2.2}$ and C_3, however, a number of aspects of these RM-ODP definitions are FDT dependent, such as behavioural compatibility or internal validity. So, we can only make such a comparison for specific FDTs. [9] has specialized definitions $C_{2.2}$ and C_3 for LOTOS by instantiating the definitions with obvious notions of internal validity, conformance and behavioural compatibility. The main results of this work will be revisited in chapter 4 as instances of the general notion of consistency that we will adopt in this report.

Probably the most important implication of [9] is that consistency checking must be performed selectively. In particular, it is inappropriate to view consistency checking as a single mechanism which can be applied to any pair of specifications. For example, it would be inappropriate to check two specifications which express exactly corresponding functionality with the same notion of consistency that is applicable to checking consistency between specifications which extend each other's functionality. Thus, in order to apply suitable consistency checks the relationship of the specifications being checked must be made available. The RM-ODP has no provision for the communication of such information. The correspondence rule concept is used in the reference model as a means to locate portions of viewpoint specifications that should be compared. However, there is no means to define how these portions of specifications should be related.

3.2 A General Definition of Consistency

This subsection presents a general definition of consistency. Our work with the three RM-ODP definitions has shown that each is a specialized notion of consistency that is applicable in a certain setting, e.g. C_1 to consistency in Z, but none of the definitions gives the "big picture" and is general enough to be instantiated reasonably for many FDTs and many notions of ODP consistency. We will give general definitions of the consistency checking relationships: *consistency*, both *intra* and *inter* language, and *unification*. First though we will present the notation that we will work with. Importantly, this notation reflects the search for a general interpretation of consistency by defining very general notational conventions. These conventions will be specialized for particular FDTs and particular forms of consistency.

3.2.1 Notation

We begin by assuming a set $COMP$ of all possible computations. Subsets of $COMP$ include IMP the set of physical implementations and DES the set of formal descriptions. The latter of these is the domain that we will be working in; DES contains both formal specifications in languages such as LOTOS and Z and semantic descriptions in notations such as labelled transition systems and ZF set theory.

We assume a set DR of *description relations*. Members of this set relate pairs of descriptions in DES. DR embraces all possible ways of relating descriptions, e.g. refinement relations or semantic maps. For a particular relation $r \in DR$, where $r \subseteq DES \times DES$ we define the left and right projections of r as: $p_l(r) = \{D : \exists\, D' s.t. (D, D') \in r\}$ and $p_r(r) = \{D : \exists\, D' s.t. (D', D) \in r\}$

DR is subdivided into DEV the set of *development relations* and SEM the set of *semantic maps*. Importantly, although members of DEV and of SEM have very different functions, both can be viewed as relations between pairs of descriptions, possibly in different languages.

Development relations are written dv or dev and if $X\ dv\ X'$ then, in some sense, X is a valid development of X'. Our concept of a development relation generalises all notions of evolving a formal description towards an implementation and thus embraces the many such notions that have been proposed. In particular, DEV contains *refinement* relations, *equivalences* and relations which can broadly be classed as *implementation* relations [36] such as the LOTOS conformance relation

conf. These different classes of development are best distinguished by their basic properties. Refinement is typically reflexive and transitive (i.e. a preorder); equivalences are reflexive, symmetric and transitive; and implementation relations are only reflexive. The distinction between refinement and implementation relations is particularly significant; transitivity is a crucial property in enabling incremental development of specifications towards realizations and implementation relations are typically lacking in this respect.

In general though we do not require that development relations support any specific properties. In particular, we cannot even assume reflexivity in the general case. This is because, in order to support inter language consistency checking, we allow development relations to relate descriptions in different notations. In these circumstances reflexivity is not a sensible concept.

Members of SEM are semantic maps between descriptions in formal techniques. Typically they map descriptions from one formal technique to a second formal technique. Elements of SEM will usually be denoted $[\![]\!]$.

Descriptions are written in formal techniques. The set of all such techniques is denoted FT. Formal techniques are triples; they are elements of $\mathcal{P}DES \times \mathcal{P}DEV \times \mathcal{P}SEM$. Thus, every formal technique is characterised by the set of possible descriptions in the notation, a set of associated development relations and a set of semantic maps. We require that the left projection of all elements of DEV and SEM contains a subset of DES. For a particular formal technique ft we denote the set of all descriptions in ft as DES_{ft}, the set of all development relations as DEV_{ft} and the set of all semantic maps as SEM_{ft}.

3.2.2 Consistency

Basic Definition. In its general form consistency is a check which takes any number of descriptions and returns true if all the descriptions are consistent and false otherwise. This check will be performed according to a list of development relations, one per description, and is denoted, $C[dv_1, dv_2, ...dv_n](X_1, X_2, ..., X_n)$. The validity of the check has two elements: *type correctness* and *consistency*.

Definition 8 (Type Correctness) *$C[dv_1, dv_2, ...dv_n](X_1, X_2, ..., X_n)$ is type correct iff* $(X_1 \in p_r(dv_1) \ \wedge\ X_2 \in p_r(dv_2) \ \wedge\ ... \wedge\ X_n \in p_r(dv_n)) \ \wedge\ (p_l(dv_1) \cap p_l(dv_2) \cap ... \cap p_l(dv_n) \neq \varnothing)$.

Type correctness ensures, firstly, that for every description the corresponding development relation, i.e. dv_i for X_i, is correctly typed with regard to the description. In addition, type correctness ensures that the target types of the relations has some intersection. This check has the function of determining that the consistency check being attempted is sensible. Type correctness will not be an issue for intra language consistency, but will be necessary when determining an appropriate inter language consistency check to apply. When writing $C[dv_1, dv_2, ...dv_n](X_1, X_2, ..., X_n)$ unless otherwise stated we will assume the check has already been shown to be type correct.

Once type correctness has been determined we can investigate consistency. Intuitively we view n specifications $X_1, X_2, ..., X_n$ as consistent if and only if there exists a physical implementation which is a realization of all the specifications, i.e. X_1, X_2 through to X_n can be implemented in a single system. However, we can only work in the formal setting, so we express consistency in terms of a common (formal) description, X, and a list of development relations, $dv_1, dv_2, ...dv_n$. The definition states that n descriptions are consistent if and only if a description can be found which is a development of X_1 according to dv_1, X_2 according to dv_2, through to X_n according to dv_n, and the third description is internally valid, written $\Psi(X)$. The structure of the consistency check is depicted in figure 3.1 and is formalized in definition 9.

Definition 9 (Consistency)
$X_1, X_2, ..., X_n$ are consistent by $dv_1, dv_2, ...dv_n$, i.e. $C[dv_1, dv_2, ...dv_n](X_1, X_2, ..., X_n)$ holds, iff $\exists\, X \in DES$ *s.t.* $(X \ \ dv_1\ X_1 \ \wedge X \ dv_2\ X_2 \ \wedge ... \wedge\ X \ dv_n\ X_n) \ \wedge \Psi(X)$.

For n descriptions to be consistent this definition requires that X is a common development of X_i for all i between 1 and n. Notice that we allow the descriptions to be related to their

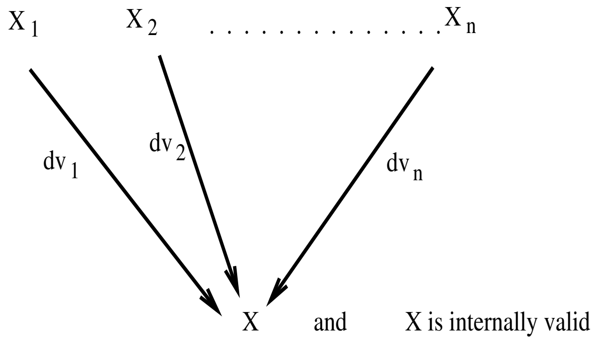

Figure 3.1: A Consistency Check

common development in different ways, i.e. if $dv_i \neq dv_j$. This is important in order to support the full generality of ODP viewpoints. A particular specialization of the viewpoints may for example require a viewpoint to be related to a second viewpoint directly by refinement. In order to reflect alternative classes of specialization we will distinguish between *balanced* and *unbalanced* consistency. These two alternatives will be discussed in sections 3.2.3 and 3.2.4.

The internal validity check in the above definition formalizes the notion of implementability. It is required because descriptions relate to physical implementations in different ways for different languages and, in particular, for some FDTs not all specifications are implementable. For example, a Z specification that contains an operation $[n! : \mathbb{N} \mid n! = 5 \wedge n! = 3]$ has no real implementation. Thus, for some FDTs it is possible to find a description which is a common development of a pair of specifications, but is not itself implementable. The property $\Psi(X)$ is true if and only if the description X has a real implementation. Thus, Ψ acts as a receptacle for properties of particular languages that make descriptions in that language unimplementable. For example, a Z specification which contains contradictions would not be internally valid.

In most cases $X_1, X_2, ..., X_n$ in the above definition will all be specifications, however, X will commonly be a semantic representation. In particular, if some of $X_1, X_2, ..., X_n$ are in different languages then X is almost certain to be in a common semantic notation. The properties that enable a semantic notation to be suitable for representing common developments of specifications in different formal techniques will be discussed in section 3.4. If $X_1, X_2, ..., X_n$ are in the same formal technique then $C[dv_1, dv_2, ...dv_n](X_1, X_2, ..., X_n)$ is called an *intra language* consistency check and if for some i and j between 1 and n, X_i and X_j are in different formal techniques then $C[dv_1, dv_2, ...dv_n](X_1, X_2, ..., X_n)$ is called an *inter language* consistency check. We will denote this interpretation of consistency as C when we refer to it in text.

Binary Consistency. An important special case is binary consistency, i.e. the consistency check $C[dv_1, dv_2](X_1, X_2)$ is performed. Binary consistency is a binary relation and is often written, $X_1 \; C_{dv_1, dv_2} \; X_2$. The possibility of inter language consistency make some of the standard properties of binary relations problematic. For example, we can consider reflexivity, symmetry and transitivity.

Proposition 1
Binary consistency is neither (i) reflexive, (ii) symmetric or (iii) transitive.

Proof

(i) Reflexivity is the case $C[dv](X)$, and this will be false whenever X is not internally valid.

(ii) Assuming $C[dv_1, dv_2](X_1, X_2)$ is true, in the case of inter language consistency $C[dv_1, dv_2](X_2, X_1)$ is likely not even to be type correct. Thus, in its most general form symmetry of consistency does

not even yield a type correct consistency check.
(iii) Assuming $C[dv_1, dv_2](X_1, X_2)$ and $C[dv_3, dv_4](X_2, X_3)$ hold then transitivity requires us to show that X_1 and X_3 are consistent, however, according to what development relations will we check consistency? The transitivity varient that we would like is that $C[dv_1, dv_4](X_1, X_3)$ follows from the assumptions. However, nothing in our assumption guarantees that, $p_l(dv_1) \cap p_l(dv_4) \neq \emptyset$, thus, $C[dv_1, dv_4](X_1, X_3)$ may not be type correct. Furthermore, even if we assume type correctness of $C[dv_1, dv_4](X_1, X_3)$, consistency will not always hold, since $C[dv_1, dv_2](X_1, X_2)$ and $C[dv_3, dv_4](X_2, X_3)$ are likely to have different common developments that cannot be related. □

Implementation Complete. There are a number of languages in which all specifications are internally valid. This is for example the case for LOTOS, for which even a deadlock has an implementation equivalent. Thus, we introduce the following notation:-

Notation 1 (Implementation Complete) *A formal technique ft is called implementation complete iff* $\forall\, X \in DES_{ft},\ \Psi(X)$.

The following result is almost trivial, but it enables us to start characterising reflexivity of consistency.

Proposition 2
If $X \in D$ *for* $D \subseteq DES$, *then* $C[dv_1, dv_2](X, X)$ *holds, i.e. reflexivity of consistency, iff* $\forall\, X \in D\ \exists\, X' \in DES$ *s.t.* $X'\ (dv_1 \cap dv_2)\ X \wedge \Psi(X')$.

Proof
$(\Longrightarrow) X\ C_{dv_1, dv_2}\ X \Longrightarrow \exists\, X' s.t.\ X'\ dv_1\ X\ \wedge\ X'\ dv_2\ X\ \wedge\ \Psi(X') \Longrightarrow\ X'\ (dv_1 \cap dv_2)\ X\ \wedge\ \Psi(X')$.
$(\Longleftarrow)\ \exists\, X'\ s.t.\ X'\ (dv_1 \cap dv_2)\ X \wedge \Psi(X')\ \Longrightarrow\ X'$ is the required common development. □

Note that $C[dv_1, dv_2](X, X)$ can also be written as $C[dv_1 \cap dv_2](X)$. Proposition 2 has the following immediate corollary.

Corollary 1
If ft is implementation complete and dv_1 *and* dv_2 *are reflexive, then* $\forall\, X \in DES_{ft},\ C[dv_1, dv_2](X, X)$ *holds, i.e. reflexivity of consistency.*

Proof
This result follows from the reflexivity of the development relations, which implies that X is a common development, and from the right to left implication in proposition 2. □

This corollary implies that consistency is reflexive for a language such as LOTOS in which all specifications are internally valid and development is at least reflexive.

Pairwise Consistency. An important issue is in what way we can determine consistency, for example, can we assert consistency between three or more descriptions by performing a series of binary consistency checks. In order to determine this we consider the notion of a pairwise consistency check:-

Definition 10 (Pairwise Consistency) *Descriptions* $X_1, X_2, ..., X_n$ *are pairwise consistent according to development relations* $dv_1, dv_2, ...dv_n$ *iff* $\forall\, X_i, X_j\ s.t.\ 1 \leq i, j \leq n,\ X_i\ C_{dv_i, dv_j}\ X_j$.

The following result characterizes the broad relationship between pairwise and normal consistency.

Proposition 3
(i) Consistency implies pairwise consistency.
(ii) Pairwise consistency of three or more specifications does not imply consistency.

Proof
(i) Assume $\exists X \in DES\ s.t.\ (X\ dv_1\ X_1\ \wedge\ X\ dv_2\ X_2\ \wedge ... \wedge\ X\ dv_n\ X_n)\ \wedge \Psi(X)$. Now clearly $X_i C_{dv_i, dv_j} X_j$ for any $1 \leq i, j \leq n$ since X can act as the internally valid common development.
(ii) We demonstrate this by counterexample. Consider the three specifications: $S_1 = [x!, y! : \mathbb{N} \mid x! = y!]$, $S_2 = [x!, z! : \mathbb{N} \mid x! = z!]$ and $S_3 = [z!, y! : \mathbb{N} \mid z! \neq y!]$ Intuitively these are balanced pairwise consistent, i.e. $S_1\ C\ S_2$, $S_2\ C\ S_3$, $S_1\ C\ S_3$, but, they are not globally consistent. □

Intuitively, the second part of the above proposition arises because pairwise consistency only requires the existence of a common development. Thus, many pairwise consistency results may exist each of which focuses on a different common development. This is not sufficient to induce "global" consistency which requires the existence of a single common development. We should also emphasise that the generalized consistency that the ODP viewpoints model induces is our normal consistency, not pairwise consistency, since a single development of all the viewpoints (i.e. the realization) is required. In later sections we will characterize circumstances in which pairwise and normal consistency are the same.

3.2.3 Balanced Consistency

Balanced consistency reflects the situation in which the specifications being checked for consistency are at the same level of abstraction; balanced consistency is written: $C[dv](X_1, X_2, ..., X_n)$. It should be noted that some of our previous papers have only considered balanced consistency, e.g. [9] and presented this as consistency in its entirety. This report presents a generalization of that work.

Definition 11 (Balanced Consistency)
$C[dv_1, dv_2, .., dv_n](X_1, X_2, ..., X_n)$, is balanced iff $dv_i = dv_j$, $\forall\, dv_i, dv_j$ s.t. $1 \leq i, j \leq n$.

We have the following result:-

Proposition 4
$C[dv](X_1, X_2, ..., X_n)$ is well founded, i.e. $C[dv](X_1, ..., X_n) = C[dv](Y)$ where Y is any possible permutation of $X_1, ..., X_n$.

Proof
Immediate from definition of consistency and balanced consistency. □

This proposition states that the ordering of $X_1, ..., X_n$ in the argument list of C is not important in balanced consistency. This is once again a very obvious result, but does contrast with the situation for unbalanced consistency, where ordering is crucial and permuting the order of descriptions may invalidate type correctness.

Once again we can consider the special case of binary balanced consistency, $C[dv](X_1, X_2)$, which is often written as C_{dv}. The next result follows naturally from the previous result:-

Proposition 5
C_{dv} is symmetric.

Proof
From proposition 4. □

This proposition characterizes symmetry of binary balanced consistency and proposition 2 characterizes reflexivity of all binary consistency; transitivity of C_{dv} is, however, more difficult to characterize. We have the following partial characterization.

Proposition 6 *dv is transitive and symmetric $\Longrightarrow$ C_{dv} is transitive*

Proof Assume $\exists\, X, X'\ s.t.\ X\ dv\ X_1 \wedge X\ dv\ X_2 \wedge \Psi(X)$ and $X'\ dv\ X_2 \wedge X'\ dv\ X_3\ \wedge\ \Psi(X')$; then from symmetry of dv we get $X_2\ dv\ X$ and transitivity can be applied twice to get $X'\ dv\ X_1$. Thus, X' is the required common development of X_1 and X_3. □

The following results which relate the characteristics of the development relation used to the induced balanced consistency are also easily obtained:-

Proposition 7
(i) If dv is reflexive and $\Psi(X_1)$, then $X_1\ dv\ X_2 \implies X_1\ C_{dv}\ X_2$.
(ii) If dv is symmetric and transitive then $X_1\ C_{dv}\ X_2 \implies X_1\ dv\ X_2$.

Proof
(i) Assume $X_1\ dv\ X_2$ and $\Psi(X_1)$; from reflexivity of X_1 we get X_1 is the required common development.
(ii) Assume $\exists\, X\ s.t.\ X\ dv\ X_1\ \wedge\ X\ dv\ X_2\ \wedge\ \Psi(X)$; then from symmetry $X_1\ dv\ X$ and from transitivity $X_1\ dv\ X_2$ as required. □

Corollary 2
If ft is implementation complete and dv is an equivalence relation, then for all descriptions in ft, $dv = C_{dv}$.

This result will be valuable when we seek to relate behavioural compatibility to our interpretation of consistency. See section 4.2.2 for a discussion of this.

In addition, as the following result shows, if we impose some strong requirements on the development relation we can relate pairwise consistency to consistency in the balanced case:-

Proposition 8
dv is transitive and symmetric $\implies$ (balanced pairwise consistency $\iff$ balanced consistency).

Proof Assume that dv is transitive and symmetric, then we can prove the equivalence of pairwise consistency and consistency as follows:-
($\Longleftarrow$) We already have this from 3.
($\Longrightarrow$) Assume $\forall\, X_i, X_j\ s.t.\ 1 \le i, j \le n \wedge i \neq j\ \exists\, Y_k\ s.t.\ 1 \le k \le n(n-1)/2\ \wedge\ \Psi(Y_k)\ \wedge\ Y_k\ dv\ X_i \wedge Y_k\ dv\ X_j$. (In this proof we assume that all Y_ks are distinct. This is the worst case situation to prove, if different pairs have the same common development the situation only becomes easier). We will show that any of the Y_ks could act as a common development for all the descriptions. So, pick Y_r s.t. $1 \le r \le n(n-1)/2$. Firstly, note that $\Psi(Y_r)$ by the assumption. Now pick X_t s.t. $1 \le t \le n$. Clearly, $\exists\, X_s\ s.t.\ Y_r\ dv\ X_s$ and $\exists\, Y_m\ s.t.\ Y_m\ dv\ X_t \wedge Y_m\ dv\ X_s$. From symmetry of dv we get $X_s\ dv\ Y_m$ and we can apply transitivity twice to $Y_r\ dv\ X_s$, $X_s\ dv\ Y_m$ and $Y_m\ dv\ X_t$ to get $Y_r\ dv\ X_t$ as required. □

3.2.4 Unbalanced Consistency

Unbalanced consistency reflects the situation in which the specifications being checked for consistency are at different levels of abstraction or have different granularities. Such relationships are easy to imagine for particular specializations of the ODP viewpoints. In circumstances in which confusion cannot arise unbalanced consistency is denoted $C[dv_1, dv_2, .., dv_n](X_1, X_2, ..., X_n)$, although if we wish to specifically distinguish the unbalanced case from the general case we will write $C^u[dv_1, dv_2, .., dv_n](X_1, X_2, ..., X_n)$. In general unbalanced consistency is considerably more difficult to work with than balanced consistency. We have the following general definition:-

Definition 12 (Unbalanced Consistency)
$C[dv_1, dv_2, .., dv_n](X_1, X_2, ..., X_n)$, is unbalanced iff $\exists\, dv_i, dv_j\ s.t.\ 1 \le i, j \le n\ \wedge\ dv_i \neq dv_j$.

Some results can be derived on binary unbalanced consistency, which is often denoted as C_{dv_1, dv_2} or as $C^u_{dv_1, dv_2}$ if we wish to distinguish this class of consistency from the general binary case.

Proposition 9
If ft is implementation complete and dv is a preorder (i.e. reflexive and transitive) then $\forall X_1, X_2 \in DES_{ft},\ X_1\ dv\ X_2 \iff X_1\ C_{dv^{-1},dv}\ X_2$.

Proof
($\Longrightarrow$) Firstly, $X_1\ dv\ X_2$ by assumption, but also $X_1\ dv^{-1}\ X_1$ by reflexivity of dv. So, X_1 is the required common development.
($\Longleftarrow$) Assume $\exists X\ s.t.\ X_1\ dv\ X \wedge X\ dv\ X_2$ then by transitivity of dv, $X_1\ dv\ X_2$. □

Interestingly, we also have:-

Proposition 10
If ft is implementation complete and dv is a preorder then $\forall X_1, X_2 \in DES_{ft},\ X_1\ dv\ X_2 \iff X_1\ C_{(dv \cap dv^{-1}),dv}\ X_2$.

Proof
($\Longrightarrow$) $X_1\ dv\ X_2$ by assumption, also $X_1\ dv \cap dv^{-1}\ X_1$ by reflexivity.
($\Longleftarrow$) Since $X_1\ C_{(dv \cap dv^{-1}),dv}\ X_2 \implies X_1\ C_{dv^{-1},dv}\ X_2$ and using previous proposition □

This final result characterizes the relationship between dv and $C_{\omega,dv}$ where ω is the equivalence defined by $\omega = dv \cap dv^{-1}$. These results will be important when we characterize the consistency arising from the LOTOS refinement preorders, e.g. reduction and extension, and their equivalence, testing equivalence.

Corollary 3
For implementation complete formal techniques and dv a preorder, $dv = C_{dv^{-1},dv} = C_{(dv \cap dv^{-1}),dv}$.

3.3 Unification

Unification is the mechanism by which descriptions are composed in such a way that the composition is a development of all the descriptions.

Definition 13 (Unification Set) $\mathcal{U}[dv_1, dv_2, .., dv_n](X_1, X_2, ..., X_n) = \{\ X\ :\ X \in DES\ \wedge (X\ dv_1\ X_1\ \wedge X\ dv_2\ X_2\ \wedge ... \wedge\ X\ dv_n\ X_n)\}$.

The unification set is the set of all common developments of a list of descriptions, i.e. the set of all unifications. Clearly, $C[dv_1, dv_2, .., dv_n](X_1, X_2, ..., X_n)$ holds if and only if $\exists X \in \mathcal{U}$ such that $\Psi(X)$. In fact, one approach to consistency checking is to perform a unification and then to show that this unification is internally valid. This will be the approach taken for consistency checking in Z.

In the same way as for consistency we can consider binary unification and distinguish between balanced and unbalanced unification. We begin by considering balanced unification.

3.3.1 Balanced Unification

Definition 14 $\mathcal{U}[dv_1, dv_2, .., dv_n](X_1, X_2, ..., X_n)$ *is a balanced unification set iff* $\forall\ dv_i, dv_j\ s.t.\ 1 \leq i, j \leq n,\ dv_i = dv_j$.

The balanced unification set is denoted $\mathcal{U}[dv](X_1, X_2, ..., X_n)$ and the following results are easily obtained:

Proposition 11
(i) $\mathcal{U}[dv](X_1, X_2, ..., X_n)$ *is well founded, i.e. any permutation of the order of* $X_1, X_2, ..., X_n$ *will give the same unification set.*
(ii) $\mathcal{U}[dv](X_1, X_2) = \mathcal{U}[dv](X_2, X_1)$ *- commutativity*
(iii) $\forall X \in \mathcal{U}[dv](X_1, X_2, ..., X_n),\ X\ dv\ X_1, X_2, ..., X_n$ *- common development*

Proof Straightforward. □

3.3.2 Unbalanced Unification

Definition 15 *$\mathcal{U}[dv_1, dv_2, ..., d_n](X_1, X_2, ..., X_n)$ is an unbalanced unification set iff $\exists\, dv_i, dv_j$ s.t. $1 \leq i, j \leq n \;\wedge\; dv_j \neq dv_j$.*

Wellfoundedness and commutativity will not hold for unbalanced unification, common development will hold.

3.3.3 Representative Unification

A particular unification algorithm will construct just one member of the unification set. Importantly, we need to know that the unification that we construct is internally valid if and only if an internally valid unification exists; otherwise we may construct an internally invalid unification despite the fact that an alternative unification may be internally valid.

Thus, we introduce the concept of a representative unification, which is defined as follows:-

Definition 16 *$X \in \mathcal{U}[dv_1, dv_2, ..., d_n](X_1, X_2, ..., X_n)$ is a representative unification iff $(\exists\, X' \in \mathcal{U}[dv_1, dv_2, ..., d_n](X_1, X_2, ..., X_n)$ s.t. $\Psi(X')) \implies \Psi(X)$.*

We denote a representative unification as $U[dv_1, dv_2, ..., d_n](X_1, X_2, ..., X_n)$. The following result is very straightforward:-

Proposition 12
ft is implementation complete and $X_1, ..., X_n \in DES_{ft} \implies \forall\, X \in \mathcal{U}[dv_1, ..., d_n](X_1, ..., X_n)$, X is a representative unification.

We believe that the *least unification* will, in general, generate a representative unification. The importance of taking the least unification is that the contradications contained in the unification will reflect contradictions occuring in the original specifications and will not have been introduced during development.

Unfortunately, for inter language consistency the least of the set of unifications is a problematic concept. Specifically, descriptions in the unification set, $\mathcal{U}[dv_1, dv_2, .., dv_n](X_1, X_2, ..., X_n)$, are likely to be in a different notation from $X_1, X_2, ..., X_n$, thus it is unlikely that the unifications can be related in a type correct manner using $dv_1, dv_2, .., dv_n$. Thus, we will consider the least unification in the intra language case.

Definition 17 (Least Unification (Intra Language)) *$X \in \mathcal{U}[dv_1, dv_2, .., dv_n](X_1, X_2, ..., X_n)$ is the least unification iff $\neg\exists\, X' \in \mathcal{U}[dv_1, dv_2, .., dv_n](X_1, X_2, ..., X_n)$, s.t. $X \;(dv_1 \cap dv_2 \cap ... \cap dv_n)\; X' \;\wedge\; \neg(X' \;(dv_1 \cap dv_2 \cap ... \cap dv_n)\; X)$.*

We denote the least unification $LU[dv_1, dv_2, ..., d_n](X_1, X_2, ..., X_n)$.

3.4 Inter Language Consistency

The basic definition of consistency that we presented in section 3.2.2 is able to relate descriptions in different formal techniques and thus to support inter language consistency. This would be the case if the unification sought is a description in a common semantics, i.e. a semantic notation that can represent the formal techniques of both the original descriptions. However, it is important that we consider what constitues a satisfactory common semantic representation. This section makes a first effort to define the criteria that such a common semantics should satisfy.

We use the notion of a development relation in different notations correlating under certain conditions. These conditions amount to a full abstraction requirement.

Definition 18 (Correlation between relations) *A development relation dv_2 correlates to a development relation dv_1, written $dv_1 \rightarrow dv_2$ with regard to a semantics $[\![]\!]$ for a formal technique ft iff $\forall\, X_1, X_2 \in DES_{ft}, X_1 \;dv_1\; X_2 \iff [\![X_1]\!] \;dv_2\; [\![X_2]\!]$.*

Definition 19 (Fully Correlated) *A semantics* $[\![]\!]$ *is fully correlated with regard to a formal technique ft iff* $\forall\, dv_1$ *used in ft* $\exists\, dv_2$ *used in* $p_r([\![]\!])$ *such that* $dv_1 \rightarrow dv_2$ *with regard to the semantics* $[\![]\!]$.

3.5 Conclusion

This section concludes this chapter with a discussion of the properties of our consistency definition. In particular, the next section considers the generality of C.

3.5.1 Generality of the Definition

Reconciling the RM-ODP Definitions

As we have indicated previously all of the three RM-ODP definitions are reasonable, but each is relevant to consistency checking in a particular setting. What we have sought in this chapter is a single general definition, which can be instantiated as appropriate for different notations. We believe we have obtained such a general interpretation. In particular, our definition embraces one of the reference models definitions directly and the other two through imposition of certain constraints on the development relation used. We will consider these results here.

Reconciling $C_{2.2}$. The following proposition represents a straightforward instantiation into C.

Proposition 13
If dv is instantiated as conformance then $C_{dv} = C_{2.2}$.

Reconciling C_1. Our approach is to define a development relation with the required characteristics and instantiate this into C. We define dev as:

(dev.i) $X_1 \; dev \; X_2 \iff (X_1 \models \gamma \impliedby X_2 \models \gamma)$ and
(dev.ii) dev is a reflexive development relation.

These represent strong constraints on development. However, neither are unreasonable and could be defined in specific settings, e.g. the first constraint could be defined for Z and the second constraint is a standard requirement of refinement relations. In addition, we define internal validity as,

$\Psi(X) \iff \neg(\exists\, \gamma \; s.t. \; X \models \gamma, \neg\gamma)$

which is a natural instantiation.

We need a simple lemma.

Lemma 1
$X_1 \; C_1 \; X_2 \implies (\neg(\exists\, \gamma \; s.t. \; X_1 \models \gamma, \neg\gamma) \;\wedge\; \neg(\exists\, \gamma' \; s.t. \; X_2 \models \gamma', \neg\gamma'))$.

Proof
We will show that $X_1 \; C_1 \; X_2 \implies \neg(\exists\, \gamma \; s.t. \; X_1 \models \gamma, \neg\gamma)$. We will use contradiction, so assume $\exists\, \gamma \; s.t. \; X_1 \models \gamma, \neg\gamma$. Now if we consider X_2 it is clear that either $X_2 \models \gamma \;\vee\; X_2 \models \neg\gamma$. However, if either of these hold then C_1, i.e. $\neg(\exists\, \gamma'' \; s.t. \; X_1 \models \neg\gamma'' \;\wedge\; X_2 \models \gamma'')$, is contradicted. This gives us the required contradiction. We can make a similar argument to show that $\neg(\exists\, \gamma' \; s.t. \; X_1 \models \gamma', \neg\gamma')$ follows from C_1. $\square$

Now we can prove the equality that we want.

Proposition 14
$C_1 = C_{dev}$.

Proof
$(C_1 \implies C_{dev})$
Assume C_1, i.e. $\neg(\exists\, \gamma \; s.t. \; X_1 \models \neg\gamma \wedge X_2 \models \gamma)$. From this we can draw the following implications:-

$$\neg(\exists\,\gamma\ s.t.\ X_1 \models \neg\gamma\ \wedge\ X_2 \models \gamma) \implies \forall\,\gamma.\neg(X_1 \models \neg\gamma\ \wedge\ X_2 \models \gamma) \implies$$
$$\forall\,\gamma.\neg(\neg(X_1 \models \gamma)\ \wedge\ X_2 \models \gamma) \implies \forall\,\gamma.\neg(\neg(X_1 \models \gamma)\ \wedge\ \neg(\neg(X_2 \models \gamma))) \implies$$
$$\forall\,\gamma.(X_1 \models \gamma\ \vee\ \neg(X_2 \models \gamma)) \implies \forall\,\gamma.(X_1 \models \gamma\ \Longleftarrow\ X_2 \models \gamma)$$

thus $X_1\ dev\ X_2$, by (dev.i). Now by (dev.ii) we also have that $X_1\ dev\ X_1$. So, X_1 is a common development and from lemma 1 we have that $\Psi(X_1)$. Thus, $X_1\ C_{dev}\ X_2$ as required.

$(C_1 \Longleftarrow C_{dev})$
We will use contradiction. Thus, assume C_{dev} and the negation of C_1:
$\exists\,X\ s.t.\ X\ dev\ X_1\ \wedge\ X\ dev\ X_2\ \wedge\ \neg(\exists\,\gamma\ s.t.\ X \models \gamma, \neg\gamma)$ and $\exists\,\gamma'\ s.t.\ X_1 \models \gamma'\ \wedge\ X_2 \models \neg\gamma'$,
but from (dev.i) these assumptions imply $X \models \gamma', \neg\gamma'$ which is the required contradiction. □

Reconciling C_3. As a concept, behavioural compatibility is extremely general; the notion is, firstly, FDT dependent and, secondly, can be interpreted a number of ways for each FDT, thus, a direct relating of C_3 and C is not possible. However, we can give stong evidence that C_3 can be fully embraced. In particular, the following results give a general relationship for implementation complete formal techniques.

Proposition 15
For implementation complete languages and ω an equivalence $C_3^\omega = C_\omega$.

Proof
Directly from 2. □

Thus, if *ft* is implementation complete and behavioural compatibility induces an equivalence on C_3 we can make a straightforward instantiation of behavioural compatibility in the development relation and obtain an equivalent definition. Furthermore, the restriction to implementation complete formal techniques is not overly restrictive, since the target of C_3 is the behavioural portion of notations such as, LOTOS, Estelle and SDL, which can be viewed to be inherently implementation complete.

We will further justify that C_3 can be embraced by C by showing, in section 4.2.2, that all the obvious LOTOS instantiations of behavioural compatibility can be given an equivalent C interpretation. This is strong evidence since LOTOS is a main target for the behavioural compatibility concept. We will summarise these results here.

Firstly, using proposition 15 above we can reconcile any LOTOS instantiation that interprets C_3 as an equivalence, e.g. testing equivalence or weak or strong bisimulation. In addition, we will show that the single remaining interpretation can also be embraced. Under this interpretation behavioural compatibility is viewed as the LOTOS **conf** relation, which is a realistic interpretation of conformance. By using a relation based on **conf**, denoted **xcs**, as development relation we can get the required relationship between C_3 and C.

Proposition 16 *For LOTOS specifications and bc =***conf***, $C_3 = C_{\mathbf{xcs}}$.*

Proof
See section 4.2.2.

We will explain the relation **xcs** and prove this result in section 4.2.2.

Aspects of Generality

The previous subsection has indicated that the three RM-ODP definitions of consistency can be embraced by C. This suggests that our interpretation of consistency is general relative to the RM-ODP definitions. It is also worth highlighting the particular aspects of our interpretation that make it general:-

- Different development relations can be instantiated which are appropriate both to different FDTs and to assessing different forms of consistency.

- Notions of internal validity relevant to different languages can be employed.
- Both intra and inter language consistency are incorporated.
- Consistency checking between an arbitrary number of descriptions can be supported and checked according to a list of different development relations. Binary consistency is just a special case of this global consistency.
- Both balanced and unbalanced consistency are incorporated.
- Both logical and behavioural notions of consistency are embraced.

3.5.2 Discussion

We will ultimately need global consistency, however, there are a number of outstanding issues to consider on this topic. In particular, we need to determine which unification to choose when doing incremental consistency checking in order that we can obtain global consistency between a group of n (larger than 2) specifications. Taking representative and least unifications is clearly relevant to this issue, but more work is required in order to fully characterise the problem. Thus, in the remainder of this report we will concentrate on consistency between pairs of specifications.

Chapter 4

Consistency in LOTOS

In this chapter we present an overview of the work done on consistency checking and unification within the formal specification language LOTOS. The approach to consistency checking and unification outlined in this chapter is general: ODP correspondence rules or the object based nature of ODP specifications, for example, are not yet taken into account. As a consequence, the unification methods given in this chapter also apply to composition of partial specifications beyond the scope of Open Distributed Processing.

LOTOS is a process algebra based specification language which is used for the formal specification of distributed, concurrent information processing systems (see [6] for a general introduction). In particular, LOTOS [27] was adopted by the International Standardisation Organisation (ISO) to formally describe the services and protocols of the Open Systems Interconnection Reference Model (OSI/RM) [26, 29, 28]. Currently, LOTOS is also being used for the specification of ODP systems and standards.

The LOTOS language has two parts: a behavioural part and a data part. The former of these is a process algebraic language, related to CCS, CSP and ACP, in which systems are described in terms of the temporal relationship between externally observable actions. The subset of LOTOS that consists solely of the behavioural part is usually referred to as Basic LOTOS. The latter is an abstract data typing language, ACT-ONE. In this chapter, we will largely focus on Basic LOTOS, i.e. the behavioural part. Basic LOTOS is a natural point of focus since it is the subset of LOTOS that is most fundamentally different to Z and ultimately we are interested in addressing the hardest consistency checking between Z and LOTOS, which will arise in this circumstance. Nevertheless, since the semantics of a Full LOTOS specifications can be expressed in Basic LOTOS, the consistency checking mechanisms developed here for Basic LOTOS apply equally well to Full LOTOS behaviour specifications.

Structure of Chapter. The general framework for consistency checking outlined in chapter 3 relies on the existence of development relations for the formal techniques used. In section 4.1, some of the existing development relations for LOTOS are briefly reviewed. This is followed in section 4.2 by an investigation of the generality of our definition of consistency, definition 9, in the LOTOS setting. The three RM-ODP consistency definitions for LOTOS are interpreted and then related to the general definition of consistency given in chapter 3. Section 4.3 considers part of the spectrum of possible LOTOS instantiations of definition 9 and relates these instantiations in both the unbalanced and balanced case. This is followed in section 4.4 by a presentation of some specific consistency checking techniques for balanced consistency. Operational semantics for a number of classes of unification are presented. Some of the developed techniques are then applied to two example viewpoint specifications of a shared memory system in section 4.5. Finally, we present some concluding remarks in section 4.6.

Notation	Meaning
$P \xrightarrow{\mu} P'$	denotes a transition, i.e. P can do μ and consequently behaves as P'.
$P \xrightarrow{\mu}$	$\exists\, P'$ such that $P \xrightarrow{\mu} P'$
$P \stackrel{\mu}{\not\longrightarrow}$	$\nexists\, P'$ such that $P \xrightarrow{\mu} P'$
$\stackrel{\epsilon}{\Longrightarrow}$	reflexive and transitive closure of $\xrightarrow{i}$
$P \stackrel{\alpha\sigma}{\Longrightarrow} P'$	$\exists\, Q, Q' \cdot P \stackrel{\epsilon}{\Longrightarrow} Q \xrightarrow{\alpha} Q' \stackrel{\sigma}{\Longrightarrow} P'$
$P \stackrel{\sigma}{\Longrightarrow}$	$\exists\, P' \cdot P \stackrel{\sigma}{\Longrightarrow} P'$
$P \stackrel{\sigma}{\not\Longrightarrow}$	$\nexists\, P' \cdot P \stackrel{\sigma}{\Longrightarrow} P'$
$P \stackrel{\sigma\alpha}{\rightsquigarrow} P'$	$\exists\, Q \cdot P \stackrel{\sigma}{\Longrightarrow} \xrightarrow{\alpha} P'$

Table 4.1: Derived transition denotations

4.1 Development Relations

In this section, some well-known development relations for LOTOS are recapitulated. We assume the reader has some familiarity with LOTOS and its semantics. Before definitions of these relations are given, we introduce some notation that will allow us to reason about processes.

Notation. LOTOS has a well-defined operational semantics which maps LOTOS behaviour expressions onto Labelled Transition Systems (LTSs). As a result of the existence of such a mapping, and the possibility to express any LTS in LOTOS, we can use behaviour expressions and their corresponding LTSs interchangeably. In particular, relations defined on transition systems are likewise applicable to behaviour expressions and the processes they define.

A labelled transition system is a tuple, $< S, \mathcal{L}, T, s0 >$. S is a set of states which ranges over the possible process behaviours that the system can reach; $\mathcal{L}$ is a set of action labels; T is a set of transitions of the form $P \xrightarrow{\mu} P'$; and $s0$ is a starting state.

In the following P, P', Q, Q' stand for processes. $\mathcal{L}$ is the alphabet of observable actions associated with a certain process, while i is the invisible or internal action. We use the variables α, α_i to range over $\mathcal{L}$, and the variables μ, μ_i to range over $\mathcal{L} \cup \{i\}$. Furthermore, $\mathcal{L}^*$ denotes strings over $\mathcal{L}$. The constant $\epsilon \in \mathcal{L}^*$ denotes the empty string, and the variables σ, σ_i are used to range over $\mathcal{L}^*$. Elements of $\mathcal{L}^*$ are also called traces. In table 4.1 the notion of transition is generalised to traces. The definitions of $\stackrel{\sigma}{\Longrightarrow}$ and $\stackrel{\sigma}{\rightsquigarrow}$ are inductive on the length of σ.
Using the notation derived in table 4.1, we can now define some other useful concepts:

$Tr(P) = \{\sigma \mid P \stackrel{\sigma}{\Longrightarrow}\}$, denotes the set of traces of a process P.
$out(P, \sigma) = \{\alpha \mid \sigma\alpha \in Tr(P)\}$, denotes the set of possible observable actions after the trace σ.
$P\ after\ \sigma = \{P' \mid P \stackrel{\sigma}{\Longrightarrow} P'\}$, denotes the set of all states reachable from P by the trace σ.
$P \longrightarrow\!\!\!\dashv_\sigma = \{P' \mid P \stackrel{\sigma}{\rightsquigarrow} P'\} \subseteq P\ after\ \sigma$, denotes the set of states that P leads to under $\sigma \neq \epsilon$.
$Ref(P, \sigma) = \{X \mid \exists\, P' \in (P\ after\ \sigma)$, such that $\forall\, \alpha \in X : P' \stackrel{\alpha}{\not\Longrightarrow}\}$, denotes the refusal set of P after the trace σ.

4.1.1 Trace preorder

An important category of system properties that one would like have satisfied by a LOTOS specification, are safety properties. Safety properties state that something bad should not happen, where something bad can be interpreted as a certain trace of the specification. Observe that if S is a safety property, then $\forall\, \sigma_1, \sigma_2$ we have if $\sigma_1 \leq \sigma_2$ then $S(\sigma_2) \Rightarrow S(\sigma_1)$, i.e. if S holds for the trace σ_2, it also holds for all its prefixes. In particular, all safety properties hold for the empty trace ϵ.

When a specification is refined, it seems reasonable to require that the refinement is at least as safe as the specification. This intuition is reflected by the trace preorder.

Definition 20 (trace preorder)
Given two process specifications P and Q, then P refines Q by reducing the possible traces, denoted $P \leq_{tr} Q$, iff:

- $Tr(P) \subseteq Tr(Q)$, *or equivalently*
- $\forall\, \sigma \in \mathcal{L}^* \cdot P \stackrel{\sigma}{\Longrightarrow}$ *implies* $Q \stackrel{\sigma}{\Longrightarrow}$.

4.1.2 Conformance

In addition to safety properties we are sometimes also interested in the liveness (or deadlock) properties of a system specification. A liveness property states that something good must eventually happen. It may be required that a development of a specification does not deadlock in a situation where the specification would not deadlock, in other words, every trace that the specification *must* do, the development *must* do as well. This requirement is formalised by the **conf** relation [11] [12], which has been adopted as the primary interpretation of conformance in LOTOS.

Definition 21 (conformance)
*Given two process specifications P and Q, then P conforms to Q, denoted P **conf** Q, iff:*

- $\forall\, \sigma \in Tr(Q)$ *and* $\forall\, A \subseteq \mathcal{L}$ *we have*
  if $\exists\, P' \in (P \ \mathit{after}\ \sigma)$ *such that* $\forall\, \alpha \in A \cdot P' \stackrel{\alpha}{\not\Longrightarrow}$,
  then $\exists\, Q' \in (Q \ \mathit{after}\ \sigma)$ *such that* $\forall\, \alpha \in A \cdot Q' \stackrel{\alpha}{\not\Longrightarrow}$, *or equivalently*
- $\forall\, \sigma \in Tr(Q) \cdot Ref(P, \sigma) \subseteq Ref(Q, \sigma)$

We will also use a development relation which is a symmetric subset of **conf**. This relation is called **conf** symmetric and is denoted **cs**; it will play a central role in instantiations of C_3, the RM-ODP definition of consistency based on behavioural compatibility. In particular, since the ODP architectural semantics adopt **conf** as their interpretation of behavioural compatibility **cs** is the obvious interpretation of C_3.

Definition 22 (conf symmetric)
*Given two process specifications P and Q, then P **cs** Q iff P **conf** $Q \ \wedge\ Q$ **conf** P.*

4.1.3 Reduction

A refinement relation that combines both the preservation of safety and liveness properties is the reduction relation, **red**, defined in [11].

Definition 23 (reduction)
*Given two process specifications P and Q, then P (deterministically) reduces Q, denoted P **red** Q, iff:*

1. $P \leq_{tr} Q$, *and*
2. P **conf** Q

4.1.4 Extension

Another refinement relation proposed in [11] is the extension relation. This relation allows for the introduction of new possible traces in a refinement, while preserving the liveness properties of the specification. Extension seems particularly relevant in the context of partial specification.

Definition 24 (extension)
*Given two process specifications P and Q, then P extends Q, denoted P **ext** Q, iff:*

1. $Tr(P) \supseteq Tr(Q)$, *and*

2. $P\,\mathbf{conf}\,Q$

4.1.5 Structural Refinement - Testing Equivalence

Another view of refinement is that the refinement provides more detail on the subdivision of the system into smaller components than the specification. The specification and its refinement are semantically equivalent, i.e. they express the same external or observable behaviour. The intension of both descriptions is not the same though, as the refinement gives more detail about the internal structure of the system under consideration. The weakest interpretation of observational equivalence is captured by the testing equivalence relation.

Definition 25 (testing equivalence)
*Given two process specifications P and Q, then P is testing equivalent to Q, denoted P **te** Q, iff:*

- P **red** Q *and* Q **red** P, *or equivalently*

- P **ext** Q *and* Q **ext** P, *or equivalently*

- $Tr(P) = Tr(Q) \ \wedge\ \forall\, \sigma \in Tr(P) \cdot Ref(P, \sigma) = Ref(Q, \sigma)$.

4.1.6 Structural Refinement - Bisimulation Equivalences

An alternative interpretation of observational identity is given by the bisimulation equivalences, strong and weak bisimulation [40]. The observational identity that they induce is in some circumstances seen to be too strong [35], for example, when the observer is viewed as a tester for the process.

The definition of weak bisimulation equivalence, $\approx$, of LOTOS processes is given below.

Definition 26 (weak bisimulation)
Given two LOTOS process specifications P_1 and P_2, we define
$P_1 \approx P_2 \Longleftrightarrow$
$\forall\, \sigma \in \mathcal{L}^*$

1. *if* $\exists\, P_1' \cdot P_1 \stackrel{\sigma}{\Longrightarrow} P_1'$ *then* $\exists\, P_2' \cdot P_2 \stackrel{\sigma}{\Longrightarrow} P_2'$ *and* $P_1' \approx P_2'$; *and*

2. *if* $\exists\, P_2' \cdot P_2 \stackrel{\sigma}{\Longrightarrow} P_2'$ *then* $\exists\, P_1' \cdot P_1 \stackrel{\sigma}{\Longrightarrow} P_1'$ *and* $P_1' \approx P_2'$.

Strong bisimulation, denoted $\sim$, is defined in a similar manner to weak bisimulation, except i actions are matched in addition to observable actions. Hence strong bisimulation is an even stronger notion of observational identity than weak bisimulation.

4.1.7 Discussion: Properties of the Development Relations

Apart from **cs** the properties of the development relations presented above have been well documented in the literature. We will review some of these properties here.

Proposition 17
(i) $\leq_{tr}$, **red** *and* **ext** *are preorders.*
(ii) **te**, $\sim$, *and* $\approx$ *are equivalences.*
(iii) **conf** *is reflexive, but neither symmetric or transitive.*
(iv) **cs** *is (a) reflexive and (b) symmetric, but (c) not transitive.*

Proof
(i), (ii) and (iii) are all standard results from the theory of LOTOS and process algebra in general, see for instance [35], [24] and [40]. However, (iv) needs some justification:-

(iv.a) This is a consequence of **conf** being reflexive.

(iv.b) This is immediate from the definition of **cs**.

(iv.c) The following counterexample justifies this. Let $P := b;\ stop[]i;\ a;\ stop$; $Q := i;\ a;\ stop$ and $R := b;\ c;\ stop[]i;\ a;\ stop$; then P **cs** Q, Q **cs** R, but $\neg(P$ **cs** $R)$. This is because $\neg(P$ **conf** $R)$ as P refuses c after the trace b, but R cannot refuse c after the same trace. □

Thus, $\leq_{tr}$, **red** and **ext** can be classed together as preorder *refinement* relations; **te**, $\sim$, and $\approx$ can be classed together as *equivalences*; **conf** and **cs** are weaker *implementation* relations.

4.2 Relating the RM-ODP Definitions

Section 3.5.1 has related the three RM-ODP definitions of consistency to C in a very general way, in this section we will specialize this to LOTOS instantiations of the RM-ODP definitions. This will give us even more evidence that C is fully general. This is particularly an issue since C_3 is dependent upon the FDT specific interpretation of behavioural compatibility adopted and thus only a language specific relating of C_3 and C can be given.

The section begins by giving a set of LOTOS instantiations of relevant definitions and, in particular, the RM-ODP definitions, and then these instantiations are related to C in the following subsection.

4.2.1 RM-ODP Instantiations

Firstly, we have the following:-

Proposition 18
$\forall\, P \in DES_{LOTOS},\ \ \Psi(P)$.

This follows intuitively from considering the nature of LOTOS specifications. In particular, at least theoretically, we can view all LOTOS specifications as implementable. Even degenerate specifications, such as those containing deadlocks, for example, have a physical implementation equivalent. This is a fundamental characteristic of behavioural languages that distinguishes them from logically based specification notations.

Corollary 4
LOTOS is implementation complete and all unifications are representative.

This corollary is important as it considerably simplifies the class of consistency that must be considered for LOTOS. Furthermore, we assume that all consistency checks are type correct. This is reasonable since we are only considering consistency intra the LOTOS language.

Of the specific RM-ODP definitions, we could relate C_1 via an interpretation of LOTOS in logic, for example, the temporal logic interpretation in [20], however, this is a complex interpretation with a number of subtle issues. Thus, we will view this as beyond the immediate scope of our work and we will not consider C_1 further in the context of LOTOS. In contrast, $C_{2.2}$ and C_3 are immediately appropriate to LOTOS. We will consider these in turn.

Instantiation of $C_{2.2}$. This is very straightforward, we give the following definition:-

Definition 27 *For* $P_1, P_2, P \in DES_{LOTOS}$ $P_1\ C_{2.2}\ P_2$ *iff* $\exists P \cdot P\ \mathbf{conf}\ P_1 \wedge P\ \mathbf{conf}\ P_2$.

Although it should be noted that this instantiation is dependent on the interpretation of conformance adopted. **conf** is a weak interpretation, in particular, it does not enforce the preservation of safety properties. However, **conf** is a realistic reflection of the capabilities of conformance testing and is now accepted as the basis of work on test case generation for LOTOS [10] [12].

Instantiation of C_3. Consistency definition C_3 is dependent upon the interpretation of behavioural compatibility, which in turn hinges on the interpretation of a specification's environment and the criteria imposed on that environment. The looseness of the definition of behavioural compatibility implies that one of a number of interpretations of C_3 could be made. It is our view that C_3 could be interpreted as any of the following:-

Definition 28
(i) $P_1 C_3^{\sim} P_2$ *iff* $P_1 \sim P_2$ *- Strong Bisimulation*
(ii) $P_1 C_3^{\approx} P_2$ *iff* $P_1 \approx P_2$ *- Weak Bisimulation*
(iii) $P_1 C_3^{\mathbf{te}} P_2$ *iff* $P_1 \mathbf{te} P_2$ *- Testing Equivalence*
(iv) $P_1 C_3^{\mathbf{cs}} P_2$ *iff* $P_1\ \mathbf{cs}\ P_2$ *- Conf symmetric*

Definitions 28(i) and 28(ii) view the environment as an unconstrained observer, in the sense of bisimulation equivalences. Note definition 28(i) is particularly interesting because $C_3^{\sim}$ has the advantage of being a congruence for LOTOS.

In contrast, 28(iii) and 28(iv) view the environment as a tester for the specifications. The distinction between 28(iii) and 28(iv) is that 28(iii) implies robustness testing and 28(iv) implies restricted testing, see [10] [12] for a discussion of these alternatives. Amongst these definitions $C_3^{\mathbf{cs}}$ is particularly important for a number of reasons. Firstly, this interpretation agrees with the LOTOS definition of behavioural compatibility in the RM-ODP architectural semantics [30]. In addition, as indicated in the following proposition, $C_3^{\mathbf{cs}}$ is the weakest of the LOTOS interpretations of C_3.

Proposition 19
$C_3^{\sim} \subset C_3^{\approx} \subset C_3^{\mathbf{te}} \subset C_3^{\mathbf{cs}}$.

Proof
$C_3^{\sim} \subset C_3^{\approx} \subset C_3^{\mathbf{te}}$ are standard process algebra results. $C_3^{\mathbf{te}} \subset C_3^{\mathbf{cs}}$ requires some justification. Firstly, it is straightforward to see that $\mathbf{te} \subseteq \mathbf{cs}$. In addition, we can provide the two processes $P := a;\ stop[]i;\ b;\ stop$ and $Q := i;\ b;\ stop$ as counterexamples to justify that $\mathbf{cs} \not\subseteq \mathbf{te}$, since P $\mathbf{cs}$ Q, but $\neg(P\ \mathbf{te}\ Q)$ as the trace sets of the two processes are not equal. □

Furthermore, [9] has shown that C_3 is the strongest of the RM-ODP interpretations of consistency, thus, $C_3^{\mathbf{cs}}$ bounds the relationship between C_3 and the other RM-ODP consistency definitions and warrants particular attention.

4.2.2 Relating Definitions

This subsection specializes the results of section 3.5.1 to LOTOS.

Reconciling $C_{2.2}$. This interpretation of consistency is very straightforward:-

Proposition 20
For LOTOS $C_{2.2} = C_{\mathbf{conf}}$.

Proof
Immediate from a comparison of instantiations.

Reconciling C_3. Three of the interpretations made in section 4.2.1 can be related to our general definition easily.

Corollary 5
(i) $C_3^{\sim} = C_{\sim}$
(ii) $C_3^{\approx} = C_{\approx}$
(iii) $C_3^{\mathbf{te}} = C_{\mathbf{te}}$

Proof
Immediate from corollary 2.

Thus, interpretations of behavioural compatibility in LOTOS which are based on one of the language's equivalences are easily reflected in our general definition of consistency. But, $C_3^{\mathbf{cs}}$ is not transitive, proposition 17, so corollary 2 does not get us a relationship between $C_3^{\mathbf{cs}}$ and $C_{\mathbf{cs}}$. In fact, we have the following result.

Proposition 21
$C_3^{\mathbf{cs}} \subset C_{\mathbf{cs}}$.

Proof
Firstly, $P_1\ C_3^{\mathbf{cs}}\ P_2 \implies P_1\ C_{\mathbf{cs}}\ P_2$, follows immediately from the reflexivity of **cs**, i.e. either of P_1 or P_2 could act as the required common **cs**-development.

In addition, we can provide a counterexample to show that, $C_{\mathbf{cs}} \not\subseteq C_3^{\mathbf{cs}}$. Consider, $P_1 := i;\ a;\ stop[]b;\ c;\ stop$, $P_2 := i;\ a;\ stop[]b;\ stop$ and $P := i;\ a;\ stop$. Now, P **cs** P_1 and P **cs** P_2, but $\neg(P_1\ \mathbf{cs}\ P_2)$. This is because $\neg(P_2\ \mathbf{conf}\ P_1)$ as P_2 refuses c after the trace b, but P_1 cannot refuse c after the same trace. $\square$

This result is disappointing, but interesting. The counterexample provided is one of the few situations in which the unification has a smaller trace set than both the original specifications and futhermore a unification with a larger trace set does not seem to exist for this example. This observation motivates the following, which considers a development relation in which the trace set increases. Thus, we define extended **conf** symmetric, denoted **xcs** as:-

Definition 29 P_1 **xcs** P_2 *iff* P_1 **cs** $P_2 \ \wedge\ Tr(P_1) \supseteq Tr(P_2)$.

It should be clear that an alternative derivation of **xcs** is: P_1 **xcs** P_2 iff P_1 **ext** $P_2 \ \wedge\ P_2$ **conf** P_1. So, we have added a trace extension constraint on the development. Note in particular that using **xcs** as development relation in C will rule out the counterexample used in the previous proposition. So let us try to relate $C_{\mathbf{xcs}}$ and $C_3^{\mathbf{cs}}$.

Proposition 22
$C_{\mathbf{xcs}} \subseteq C_3^{\mathbf{cs}}$

Proof
Assume $P_1\ C_{\mathbf{xcs}}\ P_2$, i.e., $\exists P \cdot P\ \mathbf{conf}\ P_2 \ \wedge\ P_2\ \mathbf{conf}\ P \ \wedge\ Tr(P) \supseteq Tr(P_2) \ \wedge\ P\ \mathbf{conf}\ P_1 \ \wedge\ P_1\ \mathbf{conf}\ P \ \wedge\ Tr(P) \supseteq Tr(P_1)$

which expands to:-

$(i)\ \forall\sigma\ \in Tr(P_2),\ \ Ref(P,\sigma) \subseteq Ref(P_2,\sigma) \wedge$
$(ii)\ \forall\sigma'\ \in Tr(P),\ \ Ref(P_2,\sigma') \subseteq Ref(P,\sigma') \wedge$
$(iii)\ \forall\sigma''\ \in Tr(P_1),\ \ Ref(P,\sigma'') \subseteq Ref(P_1,\sigma'') \wedge$
$(iv)\ \forall\sigma'\ \in Tr(P),\ \ Ref(P_1,\sigma') \subseteq Ref(P,\sigma') \wedge$
$(v)\ Tr(P) \supseteq Tr(P_1),\ Tr(P_2)$

From properties (i), (iv) and (v) we get:-
$\forall\sigma_1 \in Tr(P_2),\ \ Ref(P_1,\sigma_1) \subseteq Ref(P,\sigma_1) \subseteq Ref(P_2,\sigma_1)$
i.e. P_1 **conf** P_2. Similarly, P_2 **conf** P_1 since properties (iii), (ii) and (v) give us:
$\forall\sigma_2 \in Tr(P_1),\ \ Ref(P_2,\sigma_2) \subseteq Ref(P,\sigma_2) \subseteq Ref(P_1,\sigma_2)$
Notice these relationships can only be derived because $Tr(P) \supseteq Tr(P_1),\ Tr(P_2)$. □

So, we have the direction of implication that we could not get with $C_{\mathbf{cs}}$, but now the other implication direction is more difficult as we need to show a unification with trace extension exists. Before we consider this we need a simple result.

Proposition 23
P_1 **cs** $P_2 \implies \forall\sigma \in Tr(P_1) \cap Tr(P_2),\ \ Ref(P_1,\sigma) = Ref(P_2,\sigma).$

Proof
A straightforward consequence of the definition of **cs**. □

We will use the following unification construction:-

Denote $\mathcal{U}_x(P_1, P_2)$ as the set of all LOTOS specifications characterised by the following constraints:-

$$
\begin{array}{ll}
Tr(\mathcal{U}_x(P_1,P_2)) = Tr(P_1) \cup Tr(P_2) \quad \wedge & -(a) \\
\forall\sigma \in Tr(\mathcal{U}_x(P_1,P_2)), & \\
\quad \sigma \in Tr(P_1) \cap Tr(P_2) \implies Ref(\mathcal{U}_x(P_1,P_2),\sigma) = Ref(P_1,\sigma) = Ref(P_2,\sigma) \quad \wedge & -(b) \\
\quad \sigma \in Tr(P_1) - Tr(P_2) \implies Ref(\mathcal{U}_x(P_1,P_2),\sigma) = Ref(P_1,\sigma) \quad \wedge & -(c) \\
\quad \sigma \in Tr(P_2) - Tr(P_1) \implies Ref(\mathcal{U}_x(P_1,P_2),\sigma) = Ref(P_2,\sigma) & -(d)
\end{array}
$$

Notice that (b) is only possible because of proposition 23. It should also be noted that this construction is well founded and will always yield a LOTOS specification. One justification for this is that [35] performs the same construction with his rooted failure tree model (definition 6.4.1 on page 153) and shows that the resulting tree is well-formed, i.e. can be mapped to a labelled transition system.

Proposition 24
$C_3^{\mathbf{cs}} \subseteq C_{\mathbf{xcs}}$.

Proof
Assume $P_1\ C_3^{\mathbf{cs}}\ P_2$, i.e. P_1 **cs** P_2, then take $X \in \mathcal{U}_x(P_1,P_2)$, we suggest that X is a common **xcs** development of P_1 and P_2, as required by $C_{\mathbf{xcs}}$. Let us show that X **xcs** P_1. We will show first that X **conf** P_1, then that P_1 **conf** X and then that $Tr(X) \supseteq Tr(P_1)$.

(i) (X **conf** P_1).
Take $\sigma \in Tr(P_1)$. Now, if σ is also a trace of P_2, by (b), $Ref(X,\sigma) = Ref(P_1,\sigma)$, however, if $\sigma \notin Tr(P_2)$, by (c), $Ref(X,\sigma) = Ref(P_1,\sigma)$.

(ii) (P_1 **conf** X).
Take $\sigma \in Tr(X)$. We have the following cases:-
(a) $\sigma \in Tr(P_1) \cap Tr(P_2) \implies Ref(P_1,\sigma) = Ref(X,\sigma)$, by (b).
(b) $\sigma \in Tr(P_1) - Tr(P_2) \implies Ref(P_1,\sigma) = Ref(X,\sigma)$, by (c).
(c) $\sigma \in Tr(P_2) - Tr(P_1) \implies Ref(P_1,\sigma) = \varnothing$ as $\sigma \notin Tr(P_1)$, hence $Ref(P_1,\sigma) \subseteq Ref(X,\sigma)$.

(iii) ($Tr(X) \supseteq Tr(P_1)$).
This is immediate from (a).

Thus, X **xcs** P_1 and it can be similarly verified that X **xcs** P_2. □

Corollary 6
$C_3^{\mathbf{cs}} = C_{\mathbf{xcs}}$.

This result completes our relating of C_3 to C and shows that all, obvious LOTOS instantiations of behavioural compatibility in C_3 can be given an equivalent formulation in C and justifies proposition 16 quoted in chapter 3.

4.3 General LOTOS Instantiations of Consistency

The previous section has shown how C is general with regard to LOTOS interpretations of the RM-ODP definitions. In this section we will consider the broad properties of LOTOS instantiations of C. We will provide categorisations of a number of the definitions. The section is divided into two subsections, the first considers unbalanced conistency and the second balanced consistency.

4.3.1 Unbalanced Consistency

The main motivation for considering unbalanced consistency is to enable us to address situations in which a viewpoint is a direct development of a second viewpoint, or in which the viewpoint specifications have a different level of granularity. Thus, we would like to give LOTOS instantiations of consistency that model the LOTOS development relations, **conf**, **red**, **ext**, etc. Firstly, it is clear that notions of development based on equivalence, e.g. $\approx$, $\sim$ and **te**, can be easily embraced. In addition, development using one of the LOTOS preorders can be easily embraced as corollary 3 suggests. We have the following instantiations of this result for LOTOS preorders:-

Proposition 25
(i) $\leq_{tr} = C_{\leq_{tr}^{-1}, \leq_{tr}} = C_{(\leq_{tr}^{-1} \cap \leq_{tr}), \leq_{tr}}$.
(ii) $\mathbf{red} = C_{\mathbf{red}^{-1},\mathbf{red}} = C_{\mathbf{te},\mathbf{red}}$.
(iii) $\mathbf{ext} = C_{\mathbf{ext}^{-1},\mathbf{ext}} = C_{\mathbf{te},\mathbf{ext}}$.

Proof
Follows immediately from corollary 3. □

However, since **conf** is not transitive we have to work a bit harder to relate this notion of development. Firstly, we note the following negative result:-

Proposition 26
$\mathbf{conf} \not\supseteq C_{\mathbf{conf}^{-1},\mathbf{conf}}$

Proof
We can use our **conf** transitivity counterexample again here. Specifically, let $P_1 := b;\ stop[]i;\ a;\ stop$, $P_2 := b;\ c;\ stop[]i;\ a;\ stop$ and $P := i;\ a;\ stop$ then, P is the required common development to give P_1 $C_{\mathbf{conf}^{-1},\mathbf{conf}}$ P_2, but $\neg(P_1$ **conf** $P_2)$. □

However, the following stronger result gives us the necessary relationship:-

Proposition 27
$\mathbf{conf} = C_{\mathbf{te},\mathbf{conf}}$.

Proof
($\mathbf{conf} \subseteq C_{\mathbf{te},\mathbf{conf}}$)
Assume P_1 **conf** P_2, but in addition from reflexivity of **te**, P_1**te** P_1 and, thus, P_1 is the required common development.

($C_{\mathbf{te},\mathbf{conf}} \subseteq$ **conf**)
Take P such that P **te** P_1 and P **conf** P_2. If we expand these out we get:
$Tr(P) = Tr(P_1) \ \wedge \ \forall \sigma \in Tr(P), \ Ref(P,\sigma) = Ref(P_1,\sigma) \ \ \wedge$
$\forall \sigma \in Tr(P_2), \ Ref(P,\sigma) \subseteq Ref(P_2,\sigma)$
Equality of the traces of P_1 and P implies that there are no traces of P_2 that P_1 could do, but P could not do, thus,
$\forall \sigma \in Tr(P_2), \ Ref(P,\sigma) = Ref(P_1,\sigma)$ and thus, $\forall \sigma \in Tr(P_2), \ Ref(P_1,\sigma) \subseteq Ref(P_2,\sigma)$. So, P_1 **conf** P_2 as required. □

This completes our relating of LOTOS development relations to C.

4.3.2 Balanced Consistency

In this section balanced consistency is instantiated with the LOTOS development relations. This class of consistency is the easiest to work with and thus we will be able to obtain a complete categorisation of the relationship between the different instantiations. Our presentation works from the weakest interpretations of consistency to the strongest.

We begin with a suprising result:-

Proposition 28
Take $P_1, \ P_2 \in DES_{LOTOS}$, then:
(i) $P_1 \ C_{\leq_{tr}} P_2 \ \equiv true$.
(ii) $P_1 \ C_{\mathbf{ext}} P_2 \ \equiv true$.
(iii) $P_1 \ C_{\mathbf{conf}} P_2 \ \equiv true$.

Proof
We justify these results in turn:-

(i) For any two processes P and Q, we have **stop** $\leq_{tr} P$ and **stop** $\leq_{tr} Q$, i.e. the empty process is always a common development of any two specifications.

(ii) In [32] it is shown that for any two LTSs a third LTS can be found that is an extension of both. Since LTSs provide the semantics for processes, this result extends to LOTOS specifications.

(iii) Since **ext** $\Longrightarrow$ **conf** this is an easy consequence of (ii). □

Corollary 7
$C_{\leq_{tr}} = C_{\mathbf{ext}} = C_{\mathbf{conf}} = true$

The implication of these results is that all pairs of LOTOS specifications will be found to be consistent by $C_{\leq_{tr}}$, $C_{\mathbf{ext}}$ and $C_{\mathbf{conf}}$. Thus, these instantiations of consistency are very weak and are unable to distinguish any specifications. In other words, when $\leq_{tr}$, **ext** or **conf** is the appropriate development relation, there is no need for a consistency check.

Example 1 *We illustrate the second case, (ii), of proposition 28.* **ext** *supports functionality extension and any pair of LOTOS specifications can be reconciled according to such extension. Figure 4.1 illustrates this fact with some examples. The following properties hold:-*

$P \in \mathcal{U}[\mathbf{ext}](P_1, P_2)$
$Q \in \mathcal{U}[\mathbf{ext}](Q_1, Q_2)$
$Q' \notin \mathcal{U}[\mathbf{ext}](Q_1, Q_2)$
$R \in \mathcal{U}[\mathbf{ext}](R_1, R_2)$
$R' \notin \mathcal{U}[\mathbf{ext}](R_1, R_2)$

Notice that (b) shows that $\mathcal{U}[\mathbf{ext}]$ must not introduce new non-determinism, e.g. Q is a unification, but Q' is not as it may refuse either d or e after performing a and Q_1 cannot refuse d after a and Q_2 cannot refuse e after a. Additionally, (c) shows that unification may limit non-determinism. Specifically, R is a unification, but R' is not as it can refuse everything after the empty trace, while R_2 must offer either a or c.

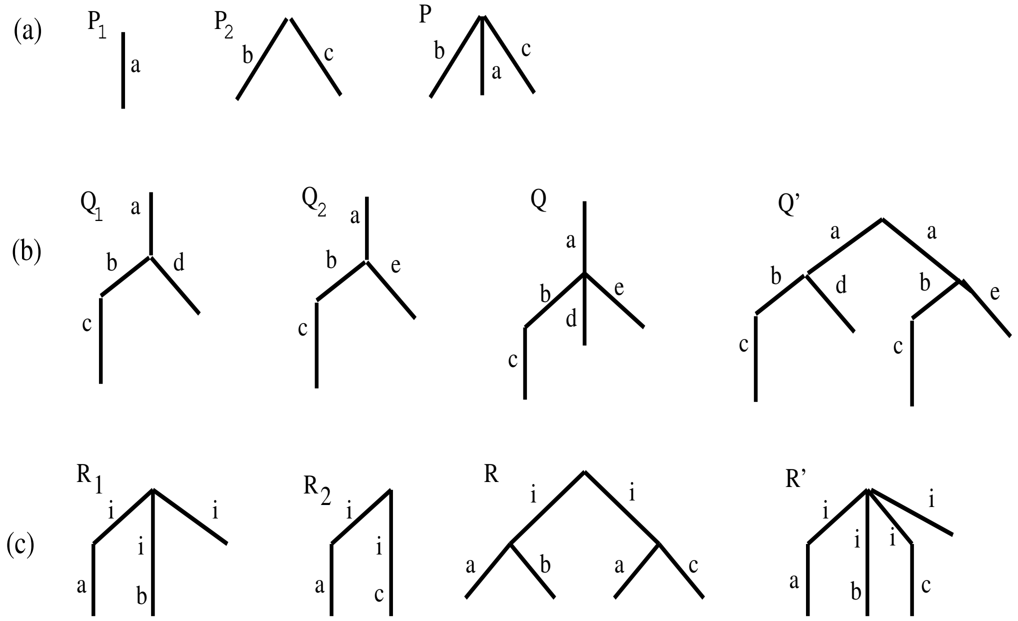

Figure 4.1: Unification by extension examples

However, other instantiations of consistency are distinguishing. We begin with the following result:-

Proposition 29
$C_{\mathbf{red}} \subset$ *true.*

Proof
We simply need to exhibit a pair of specifications that are not consistent. Consider $P_1 := a;\ stop$ and $P_2 := b;\ stop$. The only process which is a trace subset of both P_1 and P_2 is $P := stop$, but P cannot be a common reduction of either P_1 or P_2 since it will refuse a and b immediately. $\square$

Thus, $C_{\mathbf{red}}$ is stronger than $C_{\leq_{tr}}$, $C_{\mathbf{ext}}$ and $C_{\mathbf{conf}}$.

Example 2 *Consider the examples in figure 4.2. The following properties hold:-*

$\mathcal{U}[\mathbf{red}](P_1, P_2) = \varnothing$
$Q \in \mathcal{U}[\mathbf{red}](Q_1, Q_2)$
$R_2 \in \mathcal{U}[\mathbf{red}](R_1, R_2)$

The consistency relation $C_{\mathbf{cs}}$ was introduced in section 4.2.2 in an attempt to reconcile $C_3^{\mathbf{cs}}$ with C, our results there showed that $C_{\mathbf{cs}}$ is weaker than $C_3^{\mathbf{cs}}$, however, we are interested to determine how much weaker. If we can show that $C_{\mathbf{cs}}$ is stronger than $C_{\mathbf{red}}$ then we will have an upper and lower bound on the strength of $C_{\mathbf{cs}}$. Thus, we will consider the relationship between $C_{\mathbf{cs}}$ and $C_{\mathbf{red}}$. We will use the following result:-

Lemma 2
$P_1\ C_{\mathbf{cs}}\ P_2 \implies \forall\, P \in \mathcal{U}[\mathrm{cs}](P_1, P_2),\ \forall\, \sigma \in Tr(P) \cap Tr(P_1) \cap Tr(P_2),\ Ref(P, \sigma) = Ref(P_1, \sigma) = Ref(P_2, \sigma)$.

Proof
Assume $\sigma \in Tr(P) \cap Tr(P_1) \cap Tr(P_2)$ then from P **cs** P_1 we apply proposition 23 to get $Ref(P, \sigma) = Ref(P_1, \sigma)$ and from P **cs** P_2 we get $Ref(P, \sigma) = Ref(P_2, \sigma)$ and the result follows directly. $\square$

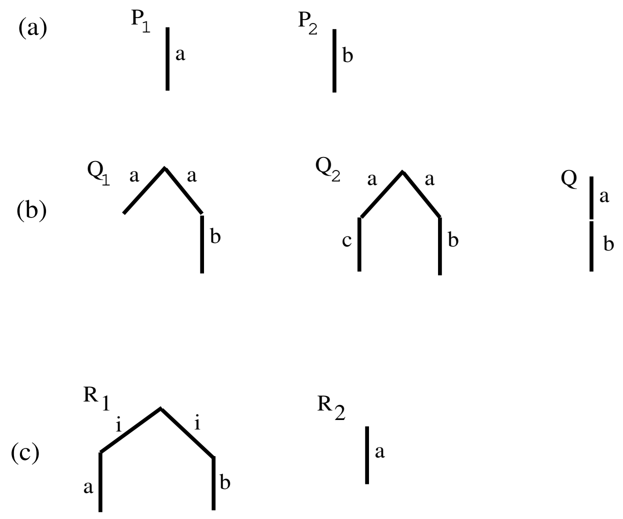

Figure 4.2: Unification by reduction, examples

Using this result we can obtain the following:-

Proposition 30
$C_{\mathbf{red}} \not\subseteq C_{\mathbf{cs}}$

Proof
We show that P_1 and P_2 exist such that $P_1\ C_{\mathbf{red}}\ P_2$, but $\neg(P_1\ C_{\mathbf{cs}}\ P_2)$. Consider $P_1 := a;\ b;\ stop[]a;\ stop$ and $P_2 := a;\ b;\ stop$. Now $P_1\ C_{\mathbf{red}}\ P_2$, because P_2 can act as the required common reduction.

We argue by contradiction that $\neg(P_1\ C_{\mathbf{cs}}\ P_2)$. So, assume P is such that P **cs** P_1 and P **cs** P_2. Firstly, P must be able to perform the trace a, because if $a \notin Tr(P)$ then $\{a\} \subseteq Ref(P, \epsilon)$, but since $\{a\} \not\subseteq Ref(P_2, \epsilon)$ this implies that $Ref(P, \epsilon) \not\subseteq Ref(P_2, \epsilon)$ and $\neg(P\ \mathbf{conf}\ P_2)$.

So, we assume $a \in Tr(P)$, hence $a \in Tr(P) \cap Tr(P_1) \cap Tr(P_2)$ and we can apply lemma 2 which implies that $Ref(P, a) = Ref(P_1, a) = Ref(P_2, a)$. But this cannot be the case as $\{b\} \subseteq Ref(P_1, a)$ and $\{b\} \not\subseteq Ref(P_2, a)$ so $Ref(P_1, a) \neq Ref(P_2, a)$. Which gives us the required contradiction and implies that such a P does not exist. $\square$

So, $C_{\mathbf{cs}}$ is not weaker than $C_{\mathbf{red}}$. Using the following small result we will be able to further clarify the relationship between $C_{\mathbf{cs}}$ and $C_{\mathbf{red}}$.

Lemma 3
P is fully deterministic (in the usual sense) $\Longrightarrow$ $(\forall \sigma \in Tr(P),\ a \in out(P, \sigma) \iff \neg \exists X \in Ref(P, \sigma)\ \cdot a \in X)$.

Proof
Standard from theory of LOTOS. $\square$

This result states that for a deterministic process an action cannot be both offered and refused.

Proposition 31
$C_{\mathbf{cs}} \subseteq C_{\mathbf{red}}$.

Proof
Assume $P_1\ C_{\mathbf{cs}}\ P_2$, i.e. $\exists P\ \cdot P$ **cs** $P_1 \wedge P$ **cs** P_2. Now construct P' as the fully deterministic

process characterised by:
$Tr(P') = Tr(P) \cap Tr(P_1) \cap Tr(P_2)$
Noting from this construction that $Tr(P') \subseteq Tr(P_1), Tr(P_2)$ and from lemma 2 that $\forall \sigma \in Tr(P'),\ Ref(P,\sigma) = Ref(P_1,\sigma) = Ref(P_2,\sigma)$. In order to show that P' is the required common reduction of P_1 and P_2 we need to show that $\forall \sigma \in Tr(P'),\ Ref(P',\sigma) \subseteq Ref(P_1,\sigma), Ref(P_2,\sigma)$. This is enough because any $\sigma' \in Tr(P_1) \cup Tr(P_2)\ \cdot \sigma' \notin Tr(P')$ will give $Ref(P',\sigma') = \varnothing$, which trivially gives us the required refusals relationship.

We argue by contradiction that $\forall \sigma \in Tr(P'),\ Ref(P',\sigma) \subseteq Ref(P_1,\sigma), Ref(P_2,\sigma)$. So, assume $\exists \sigma \in Tr(P') \cdot Ref(P',\sigma) \supset (Ref(P_1,\sigma) = Ref(P_2,\sigma) = Ref(P,\sigma))$. Thus, $\exists \{a\} \not\subseteq (Ref(P_1,\sigma) = Ref(P_2,\sigma) = Ref(P,\sigma))$, such that $\{a\} \in Ref(P',\sigma)$. From here we can use lemma 3 to get $a \notin out(P',\sigma)$, but it must also be the case that $a \in out(P_1,\sigma), out(P_2,\sigma), out(P,\sigma)$ and thus we have a contradiction as the trace $\sigma.a$ is in $Tr(P) \cap Tr(P_1) \cap Tr(P_2)$. So, it must be the case that $Ref(P',\sigma) \subseteq Ref(P_1,\sigma), Ref(P_2,\sigma)$ and P' **red** P_1 and P' **red** P_2 as required. □

Corollary 8
$C_{\mathbf{cs}} \subset C_{\mathbf{red}}$.

Proof
From propositions 30 and 31.

Thus, $C_{\mathbf{cs}}$ is strictly stronger than $C_{\mathbf{red}}$.

When the above results are combined with proposition 21 and corollary 6 we obtain a precise classification of $C_{\mathbf{cs}}$, as follows:-

Corollary 9
$C_{\mathbf{xcs}} \subset C_{\mathbf{cs}} \subset C_{\mathbf{red}}$

Proof
Immediate from proposition 21, corollary 6 and corollary 8. □

In addition, we can put an upper bound on the strength of these relationships using the following proposition:-

Proposition 32
$C_{\mathbf{te}} \subset C_{\mathbf{xcs}}$.

Proof
This follows immediately from proposition 19, which relates $C_3^{\mathbf{cs}}$ to $C_3^{\mathbf{te}}$, corollary 5 and corollary 6, which shows that $C_3^{\mathbf{cs}} = C_{\mathbf{xcs}}$. □

The relationship between the different interpretations of consistency are shown in figure 4.3. In addition, consistency based on behavioural compatibility can be incorporated into this categorisation through the following properties:-

$$C_{\mathbf{xcs}} = C_3^{\mathbf{cs}};\ C_{\mathbf{te}} = C_3^{\mathbf{te}};\ C_{\sim} = C_3^{\sim};\ \text{and}\ C_{\approx} = C_3^{\approx}.$$

These instantiations present us with a number of possible interpretations of consistency in LOTOS. This situation reflects our view that consistency checking must be performed selectively, this issue was discussed in some depth in [9]. In particular, it is inappropriate to view consistency checking as a single mechanism which can be applied to any pair of specifications. For example, it would be inappropriate to check two specifications which express exactly corresponding functionality with $C_{\mathbf{ext}}$. An implication of this is that, in order to apply suitable consistency checks the relationship of the specifications being checked must be made available by the specifier(s).

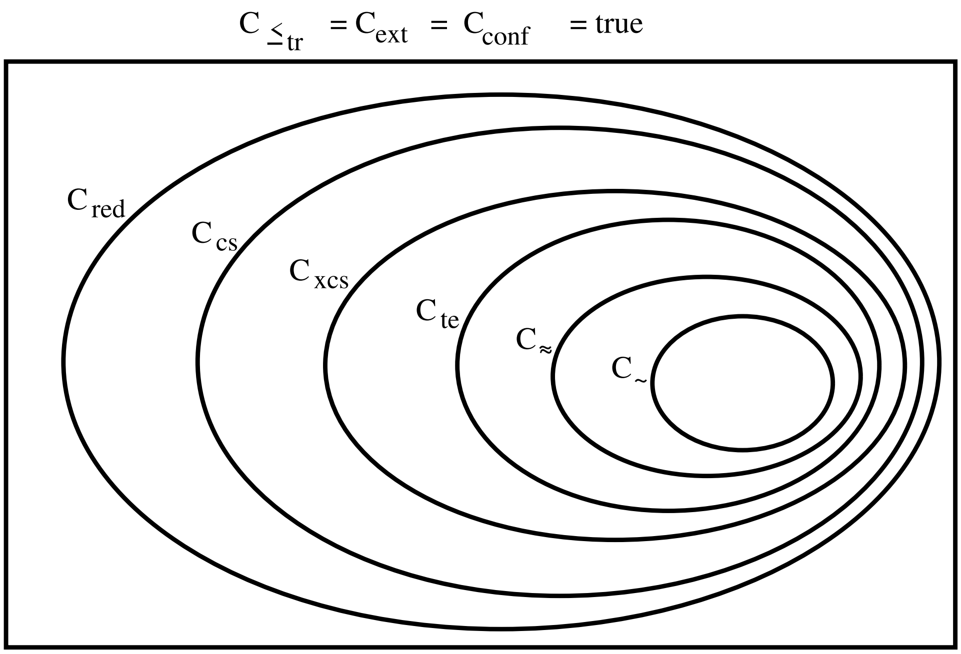

Figure 4.3: LOTOS Consistency Relations

4.4 Consistency Checking and Unification Techniques

The previous sections have classified the different LOTOS instantiations of consistency. In this section we will provide some results on how actually to check for consistency. Thus, we will provide mechanisms by which certain classes of consistency can be checked. Specifically, we provide LOTOS unification strategies for balanced binary consistency.

Although consistency implies that the set of unifications is not empty and we can determine the denotational semantics of the least unification in some cases, it would be useful for system development purposes to have a method to construct a unification within the specification language. This unification can then be used for further refinement or as the implementation specification.

The purpose of this section is to find operators for LOTOS that can be used to unify specifications. In case the original specifications are consistent (with respect to some notion of development), the unification should obviously be a common development (with respect to that notion of development). Such operational definitions of unification have several advantages over the unification algorithms that are applied to a denotational model of the specifications. The operational semantics of the unification operators give unification a constructive character which is useful for simulation and implementation purposes.

We will consider in order trace preorder, reduction, extension and then testing equivalence.

4.4.1 Trace preorder preserving unification

Parallel composition of specifications preserves the safety properties.

Proposition 33 (Unification w.r.t. $\leq_{tr}$)
If P_1 and P_2 are two arbitrary LOTOS process specifications, then the process $S := P_1 \mid\mid P_2$ is a unification of P_1 and P_2 with respect to $\leq_{tr}$.

Proof
We only prove $P \mid\mid Q \leq_{tr} P$ as the other case is symmetric.

We derive that $\forall \alpha \in \mathcal{L} \cdot (P \,||\, Q \stackrel{\alpha}{\Longrightarrow} \;\Rightarrow P \stackrel{\alpha}{\Longrightarrow})$, and therefore $Tr(P \,||\, Q) \subseteq Tr(P)$:
$P \,||\, Q \stackrel{\alpha}{\Longrightarrow} \;\Rightarrow$ (from definition of $\stackrel{\alpha}{\Longrightarrow}$)
$\exists P', Q' \cdot P \,||\, Q \stackrel{\epsilon}{\Longrightarrow} P' \,||\, Q' \wedge P' \,||\, Q' \stackrel{\alpha}{\longrightarrow} \;\Rightarrow$ (from definition of $||$)
$\exists P' \cdot P \,||\, Q \stackrel{\epsilon}{\Longrightarrow} P' \wedge P' \stackrel{\alpha}{\longrightarrow} \;\Rightarrow$ (by definition of $\stackrel{\alpha}{\Longrightarrow}$)
$P \stackrel{\alpha}{\Longrightarrow}$. □

4.4.2 Reduction preserving unification

The following theorem gives a necessary and sufficient condition on two specifications for them to be consistent with respect to reduction. The condition requires that P_1 and P_2 can at least refuse all the actions they may not both do after a certain trace.

Theorem 1 (consistency w.r.t. reduction)
Let P_1, P_2 be two LOTOS specifications using the alphabet $\mathcal{L}$, then:
$P_1 \, C_{\mathbf{red}} \, P_2 \iff \forall \sigma \in Tr(P_1) \cap Tr(P_2) \cdot \mathcal{L} - out(P_1, \sigma) \cap out(P_2, \sigma) \in Ref(P_1, \sigma) \cap Ref(P_2, \sigma)$

Proof

"$\Longleftarrow$" We need to prove that

$$\forall \sigma \in Tr(P_1) \cap Tr(P_2) \cdot \mathcal{L} - out(P_1, \sigma) \cap out(P_2, \sigma) \in Ref(P_1, \sigma) \cap Ref(P_2, \sigma)$$
implies $\exists P \cdot P \,\mathbf{red}\, P_1$ and $P \,\mathbf{red}\, P_2$.

Now, take P to be the fully deterministic process that is completely determined (modulo strong bisimulation) by the intersection of the traces of P_1 and P_2, i.e. $Tr(P) = Tr(P_1) \cap Tr(P_2)$. For such a deterministic process, we have

$$\forall \sigma \in Tr(P) \cdot Ref(P, \sigma) = \mathcal{P}(\mathcal{L} - out(P, \sigma)),$$

where $\mathcal{P}(X)$ denotes the powerset of the set X.

Next, we prove that $P \,\mathbf{red}\, P_1$ and $P \,\mathbf{red}\, P_2$:

1. From $Tr(P) = Tr(P_1) \cap Tr(P_2)$, it follows that $Tr(P) \subseteq Tr(P_1)$ and $Tr(P) \subseteq Tr(P_2)$
2. From the definition of P, we derive that

$$\forall \sigma \in Tr(P_1) \cap Tr(P_2), \forall X \in Ref(P, \sigma) \cdot X \subseteq (\mathcal{L} - out(P, \sigma)).$$

As $Tr(P) = Tr(P_1) \cap Tr(P_2)$, it follows, by the definition of $out(P, \sigma)$ and the prefix-closedness of tracesets, that

$$\forall \sigma \in Tr(P_1) \cap Tr(P_2) \cdot out(P, \sigma) = out(P_1, \sigma) \cap out(P_2, \sigma).$$

From the condition of the proposition, we can now derive that

$$\forall \sigma \in Tr(P_1) \cap Tr(P_2) \cdot X \in Ref(P, \sigma) \text{ implies } X \in Ref(P_1, \sigma) \cap Ref(P_2, \sigma).$$

Finally, we have $\forall \sigma \in Tr(P_1) - Tr(P) \cdot Ref(P, \sigma) = \varnothing \subseteq Ref(P_1, \sigma)$ and similarly for P_2, by $Ref(P, \sigma) = \varnothing \Leftrightarrow \sigma \notin Tr(P)$.

"$\Longrightarrow$" Assume that there exists an P such that $P \,\mathbf{red}\, P_1$ and $P \,\mathbf{red}\, P_2$. By contradiction:

Suppose $\mathcal{L} - out(P_1, \sigma) \cap out(P_2, \sigma) \notin Ref(P_1, \sigma) \cap Ref(P_2, \sigma)$, for a certain $\sigma \in Tr(P_1) \cap Tr(P_2)$.

Then there exists an $\alpha \in \mathcal{L} - out(P_1, \sigma) \cap out(P_2, \sigma)$ such that $\{\alpha\} \notin Ref(P_1, \sigma) \cap Ref(P_2, \sigma)$. It follows that either

$$\alpha \in out(P_1, \sigma) \wedge \alpha \notin out(P_2, \sigma) \wedge \{\alpha\} \notin Ref(P_1, \sigma) \cap Ref(P_2, \sigma), \text{ or} \tag{4.1}$$
$$\alpha \notin out(P_1, \sigma) \wedge \alpha \in out(P_2, \sigma) \wedge \{\alpha\} \notin Ref(P_1, \sigma) \cap Ref(P_2, \sigma). \tag{4.2}$$

As both cases are symmetric, we only prove case (1). Since $\alpha \notin out(P_2,\sigma)$ implies $\{\alpha\} \in Ref(P_2,\sigma)$ and $\{\alpha\} \notin Ref(P_1,\sigma) \cap Ref(P_2,\sigma) \wedge \{\alpha\} \in Ref(P_2,\sigma)$ implies $\{\alpha\} \notin Ref(P_1,\sigma)$, we derive:

$$\alpha \in out(P_1,\sigma) \wedge \alpha \notin out(P_2,\sigma) \wedge \{\alpha\} \notin Ref(P_1,\sigma) \wedge \{\alpha\} \in Ref(P_2,\sigma)$$

For $P\,\mathbf{red}\,P_1$ and $P\,\mathbf{red}\,P_2$ to hold, it is necessary that (a) $\sigma.\alpha \notin Tr(P)$, because $\sigma.\alpha \notin Tr(P_2)$, and that (b) $\{\alpha\} \notin Ref(P,\sigma)$, because $\{\alpha\} \notin Ref(P_1,\sigma)$. Clearly, (a) and (b) can never be satisfied by any process. Hence, there does not exist a reduction of both P_1 and P_2. This contradicts with the assumption that such a common reduction exists. □

Unfortunately, none of the existing LOTOS operators yields a common reduction when applied to two behaviour expressions. Therefore, we define a new operator. This binary operator, coined the *conjunction* operator, resolves all non-determinism present and removes all internal actions in either operand. The conjunction of two processes can only perform a trace if both processes are able to perform that trace.

Definition 30 (conjunction operator)
Let P and Q be LOTOS behaviour expressions. We define their conjunction, $P \otimes Q$, by the following inference rule ($\Sigma(X)$ represents generalised choice):

$$\frac{P \stackrel{\alpha}{\Longrightarrow},\ Q \stackrel{\alpha}{\Longrightarrow}}{P \otimes Q \stackrel{\alpha}{\longrightarrow} \Sigma(P \longrightarrow\!\!\!\dashv_{\alpha}) \otimes \Sigma(Q \longrightarrow\!\!\!\dashv_{\alpha})}$$

Proposition 34 (correctness of conjunction)
If $P_1\; C_{\mathbf{red}}\; P_2$, then
$P_1 \otimes P_2\,\mathbf{red}\,P_1$ *and* $P_1 \otimes P_2\,\mathbf{red}\,P_2$

Proof
We will prove only that $P_1 \otimes P_2\,\mathbf{red}\,P_1$. The other case is symmetric. The proof consists of two parts:

1. $Tr(P_1 \otimes P_2) \subseteq Tr(P_1)$:
   Inspecting the only inference rule, we see that $P_1 \otimes P_2 \stackrel{\alpha}{\Longrightarrow}$ *implies* $P_1 \stackrel{\alpha}{\Longrightarrow}$.

2. $P_1 \otimes P_2\,\mathbf{conf}\,P_1$
   Suppose there exists a P such that $P_1 \otimes P_2 \stackrel{\sigma}{\Longrightarrow} P \stackrel{\alpha}{\not\Longrightarrow}$, where $\sigma \in Tr(P_1), \alpha \in \mathcal{L}$. From the inference rule, it follows that $P = P_1' \otimes P_2'$, where $P_1' = \Sigma(P_1 \longrightarrow\!\!\!\dashv_{\sigma})$ and $P_2' = \Sigma(P_2 \longrightarrow\!\!\!\dashv_{\sigma})$. For P to refuse α either $P_1' \stackrel{\alpha}{\not\Longrightarrow}$ or $P_2' \stackrel{\alpha}{\not\Longrightarrow}$. In the first case, we are ready, because $P_1 \stackrel{\sigma}{\Longrightarrow} P_1' \stackrel{\alpha}{\not\Longrightarrow}$.
   In the case where $P_2' \stackrel{\alpha}{\not\Longrightarrow}$, we take the consistency condition $\mathcal{L} - out(P_1,\sigma) \cap out(P_2,\sigma) \in Ref(P_1,\sigma) \cap Ref(P_2,\sigma)$ into consideration. $P_2' \stackrel{\alpha}{\not\Longrightarrow}$ *implies* $\alpha \notin out(P_2,\sigma)$. By the condition, we derive that $\{\alpha\} \in Ref(P_1,\sigma) \cap Ref(P_2,\sigma)$, and therefore $\{\alpha\} \in Ref(P_1,\sigma)$. Hence there exists a $P' \in (P_1\ \mathit{after}\ \sigma)$ such that $P_1 \stackrel{\sigma}{\Longrightarrow} P' \stackrel{\alpha}{\not\Longrightarrow}$. □

Although the conjunction operator yields a common reduction, it is usually not the least reduction of its two operands. This is due to the fact that all non-determinism is resolved. When both original specifications contain non-determinism it may not be necessary to resolve all non-determinism, as is shown in the following example.

Example 3 *Take $P_1 := i;\ a;\ stop[]i;\ b;\ stop[]i;\ c;\ stop$ and $P_2 := i;\ a;\ stop[]i;\ b;\ stop$. Then $P_1 \otimes P_2 = a;\ stop[]b;\ stop$, which is a common reduction, but not the least common reduction, which is $i;\ a;\ stop[]i;\ b;\ stop$.*

4.4.3 Extension preserving unification

From proposition 28 we know that any two specifications are consistent with respect to extension. However, none of the existing LOTOS operators will always yield a common extension. Therefore, we define a new operator. This binary operator, coined the *join* operator, merges as it were those patterns of behaviour that the two operand specifications have in common, and then provides a choice between the two behaviours when they start to differ. Note that the composition will be completely deterministic until a choice for either behaviour has been made.

Definition 31 (join operator)
Let P and Q be LOTOS behaviour expressions. We define their join, $P \bowtie Q$, by the following inference rules:

$$(1)\ \frac{P \overset{\alpha}{\Longrightarrow},\ Q \overset{\alpha}{\Longrightarrow}}{P \bowtie Q \overset{\alpha}{\longrightarrow} \Sigma(P \dashv_{\alpha}) \bowtie \Sigma(Q \dashv_{\alpha})},\ (2)\ \frac{P \overset{\alpha}{\Longrightarrow},\ Q \overset{\alpha}{\not\Longrightarrow}}{P \bowtie Q \overset{\alpha}{\longrightarrow} \Sigma(P \dashv_{\alpha})},\ (3)\ \frac{P \overset{\alpha}{\not\Longrightarrow},\ Q \overset{\alpha}{\Longrightarrow}}{P \bowtie Q \overset{\alpha}{\longrightarrow} \Sigma(Q \dashv_{\alpha})}$$

Proposition 35 (correctness of join)
If P_1 and P_2 are two arbitrary LOTOS process specifications, then the process $P := P_1 \bowtie P_2$ is a unification of P_1 and P_2 with respect to **ext**.

Proof
We will prove only that $P_1 \bowtie P_2$ **ext** P_1. The other case is symmetric. The proof consists of two parts:

1. $Tr(P_1 \bowtie P_2) \supseteq Tr(P_1)$:
   Inspecting the inference rules, we see that $P_1 \overset{\alpha}{\Longrightarrow}$ *implies* $P_1 \bowtie P_2 \overset{\alpha}{\Longrightarrow}$.

2. $P_1 \bowtie P_2$ **conf** P_1
   Suppose there exists a P such that $P_1 \bowtie P_2 \overset{\sigma}{\Longrightarrow} P \overset{\alpha}{\not\Longrightarrow}$, where $\sigma \in Tr(P_1), \alpha \in \mathcal{L}$. From the inference rules, we derive the following three possible cases:

   (a) $P = P_1' \bowtie P_2'$, where $P_1' = \Sigma(P_1 \dashv_{\sigma})$ and $P_2' = \Sigma(P_2 \dashv_{\sigma})$, and $P_1' \overset{\alpha}{\not\Longrightarrow}$ and $P_2' \overset{\alpha}{\not\Longrightarrow}$. In this case, it follows directly that there exists a $P'(= P_1')$ such that $P_1 \overset{\sigma}{\Longrightarrow} P' \overset{\alpha}{\not\Longrightarrow}$.

   (b) $P = P_1'$, where $P_1' = \Sigma(P_1 \dashv_{\sigma})$, and $P_1' \overset{\alpha}{\not\Longrightarrow}$. Clearly, $P_1 \overset{\sigma}{\Longrightarrow} P_1' \overset{\alpha}{\not\Longrightarrow}$.

   (c) $P = P_2'$, where $P_2' = \Sigma(P_2 \dashv_{\sigma})$, and $P_2' \overset{\alpha}{\not\Longrightarrow}$. We argue that this case will not occur. For it to exist there should exist traces σ' and σ'' such that $\sigma'\sigma'' = \sigma$, process $P' \in (P_1 \dashv_{\sigma'})$ and a $Q' \in (P_2 \dashv_{\sigma'})$ such that $P_1 \bowtie P_2 \overset{\sigma'}{\Longrightarrow} P' \bowtie Q'$ and $P' \overset{\sigma''}{\not\Longrightarrow}$ while $Q' \overset{\sigma''}{\Longrightarrow}$. However, $\sigma'\sigma'' \in Tr(P_1)$ and because $P' = \Sigma(P_1 \dashv_{\sigma'})$ we have $P' \overset{\sigma''}{\Longrightarrow}$. □

In the definition of the operational semantics of the join operator, we make use of a so-called negative premise (see [21]). This is potentially dangerous, because the transition relation may not be uniquely defined by the collection of all the inference rules of the language. Indeed, using unguarded recursion, which is allowed in LOTOS, we can show that the use of the negative premise here is not safe. Consider, for example, the following recursive process definition: $P := a;\ stop \bowtie i;\ P$. Since the $\bowtie$-operator abstracts from internal actions, the associated LTS is non image-finite, which makes it impossible to determine the next state. Even if unguarded recursion would be forbidden, then it would still be possible to make guarded recursion unguarded by applying the **hide** operator. Despite this, the $\bowtie$ operator is meant to compose process specifications, with the aim to yield a new composed specification. A system will not be composed of itself and some other process, i.e. specifications of the shape $S_1 := S_1 \bowtie S_2$ do not make sense. Therefore, if the join operator is used for composition of specifications, its usage is safe.

Another drawback of the join operator is the fact that it does not yield the least common extension as is shown in the example below.

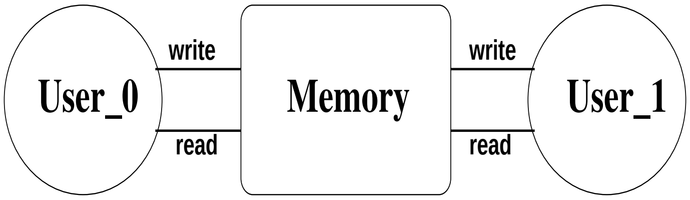

Figure 4.4: Shared Memory example

Example 4 *Consider the following two gambling machine specifications:* P_1 := *coin; lose; stop* [] *coin; win; stop,* P_2 :=*coin; lose; stop* [] *coin; win; (coin; jackpot; stop* [] *coin; lose; stop). Although the unification* $P_1 \bowtie P_2$ = *coin; (lose; stop* [] *win; coin; (jackpot; stop* [] *lose; stop)) is an extension of both specifications, the* $\bowtie$*-operator makes it completely deterministic. This is clearly not the desired effect here. The least possible unification is given by* P_2.

4.4.4 Testing equivalence preserving unification

With respect to testing equivalence, either of the two original specifications will do as the composition, because both specifications are testing equivalent to each other. However, this may result in the loss of the intention of the other specification.

4.5 Example of Unification in LOTOS

In order to demonstrate consistency checking and unification of partial specifications of a distributed information system, a simple Shared Memory system is introduced. We give two partial specifications of the system. One specification focuses on the computational aspects of the system, i.e. it describes the functional components of the system and the possible communication patterns between them. The second specification focuses on the information flow within the whole of the system without identifying the components it is composed of: it describes an invariant that should be satisfied by the data contained in the system. We thus obtain a nice separation of concerns.

Note, that the given specifications are not intended to be ODP compliant. They merely serve as a means to demonstrate the techniques for consistency checking and unification developed above.

The Shared Memory system is depicted in figure 4.4. It consists of a memory and two users. The users can access the memory through the *read* and *write* interfaces, but cannot communicate directly to one another. For simplicity we assume that the memory only contains one data value at a time.

4.5.1 Computational specification

In the computational specification the Shared Memory system is viewed as a collection of communicating processes. Two types of components are identified: a Memory component and a collection of User components. For this example, there are only two instantiations of the User process, but this could easily be extended to an arbitrary number. The User components do not communicate directly with each other. Therefore the two instantiations of User in process Users are placed in parallel (|||). There is communication between the Memory and the Users, which is represented by the synchronisation operator (||) between the processes Memory and Users.

The computational specification is not concerned with the specific data values that are exchanged between communicating components. We see this reflected in the definition of the behaviour of a User. It specifies that a user can, at any time, either perform a read or a write operation, but this viewpoint does not care what the read or written data value is, represented by a non-deterministic choice.

```
process ComputationalSpec[read, write] : noexit:=
Memory [read, write] || Users [read, write]
where
  process Users[read, write] : noexit:=
  User [read, write] (0) ||| User [read, write] (1)
  where
    process User[read, write](uid : UserId) : noexit:=
    choice x : Data []
    i;
       (read!uid !x ; User [read, write] (uid)
         []
        write!uid ?x : Data ; User [read, write] (uid) )
    endproc (* User *)
  endproc (* Users *)

  process Memory[read, write] : noexit:=
  ConcurrentReads [read]
   [>
    write?uid : UserId ?input : Data ; Memory [read, write]
  where
    process ConcurrentReads[read] : noexit:=
    SequentialReads [read] ||| SequentialReads [read]
    where
      process SequentialReads[read] : noexit:=
      choice uid : UserId, output : Data []
      i; read!uid !output ; SequentialReads [read]
      endproc (* SequentialReads *)
    endproc (* ConcurrentReads *)
  endproc (* Memory *)
endproc (* ComputationalSpec *)
```

The specification of the Memory component expresses that read operations can take place concurrently (process ConcurrentReads), but that these can at any time be disabled ([>) by a write operation. This disabling ensures that write operations take place atomically in order to avoid data inconsistencies. Taking a closer look at process ConcurrentReads, we see that it actually only allows two concurrent read operations at one time. Obviously, this can easily be extended to a higher degree of concurrency. The individual threads of ConcurrentReads, process SequentialReads, allow for read operations by arbitrary users to happen sequentially. Again the data value is non-deterministically chosen.

4.5.2 Information specification

In the information specification, only the information flow within the Shared Memory system is considered. It specifies that the Memory is initialised with an arbitrarily chosen data value first (InitMem). From then on the following invariant must hold: the data value associated with each consecutive read operation is equal to the last written value. This is ensured by the process MemoryInvariant, which has one parameter to pass the last written value to it. The process MemoryInvariant is defined using a process Read_n, which allows arbitrary users to do read operations while ensuring that *mem* is the data value read. Process Read_n can at any time be disabled by a write operation taking place. The UserId and the data value associated with the write operation can be randomly chosen, but the written data value will consequently be passed to a new invocation of MemoryInvariant.

```
process InformationSpec[read, write] : noexit:=
InitMem >> accept mem : Data in MemInvariant [read, write] (mem)
```

```
where
  process InitMem : exit(Data):=
  choice mem : Data [] i; exit(mem)
  endproc (* InitMem *)

  process MemInvariant[read, write](mem : Data) : noexit:=
  Read_n [read] (mem)
   [>
    (choice uid : UserId, x : Data []
      write!uid !x ; MemInvariant [read, write] (x) )
  where
    process Read_n[read](mem : Data) : noexit:=
    choice uid : UserId [] i; read!uid !mem ; Read_n [read] (mem)
    endproc (* Read_n *)
  endproc (* MemInvariant *)
endproc (* InformationSpec *)
```

Note that we have generally avoided variable declarations of the form read ?uid:UserId ?x:Data as a shorthand for the set of actions {read<uid,x> | uid∈UserId, x∈Data}. Instead we have made the choice more abstract in terms of non-determinism by using the construct:

choice uid : UserId, x : Data [] **i**; read!uid !x.

Although this 'style' of specification is not required in a constraint oriented style, it is necessary to make the specifications consistent by reduction.

4.5.3 Consistency check and unification

First, we need to identify which instantiation of consistency applies here. As both viewpoint specifications in this example use the same event structure and the intention is for them to work together and not extend each others functionality, $C_{\mathbf{red}}$ seems more applicable than $C_{\mathbf{ext}}$. As it is unlikely the two specifications describe exactly the same behaviour, $C_{\mathbf{te}}$ is not applicable here. As for $C_{\leq_{tr}}$, this form of consistency is also covered by $C_{\mathbf{red}}$.

In order to show that the two specifications are consistent by $C_{\mathbf{red}}$ we have to verify the consistency condition of Theorem 1. Unfortunately, the presence of data variables in the specifications leads to a state explosion, which makes it hard to verify this condition.

Fortunately, it is possible to assess the consistency of the two specifications in an alternative way. If the specifications are consistent, the conjunction operator will yield a common reduction of both specifications. Thus, we first apply the conjunction operator, and then verify whether the unification is a reduction of either specification. Part of the LTS representing the conjunction, ComputationalSpec ⊗ InformationSpec, is shown in figure 4.5. In order to represent infinitely branching transitions caused by variable declarations, we have represented such transitions symbolically. It can be verified that this composition is indeed a reduction of both original specifications.

4.6 Summary and Discussion

This chapter has investigated consistency in LOTOS. We explored instantiations of consistency with a number of the LOTOS development relations. We have given appropriate LOTOS instantiations of the RM-ODP definitions of consistency and related these to our definition. This work supports the view presented in chapter 3 that our definition is general and can embrace all the RM-ODP definitions.

We have also characterised the relative strengths of different LOTOS instantiations of C. The results of this are summarised in figure 4.3. Further, necessary and sufficient conditions were investigated for specifications to be consistent with respect to particular notions of development.

Several forms of unification are already supported in standard LOTOS. In fact, it can be argued that all binary operators enable some form of unification. The parallel operator has proved

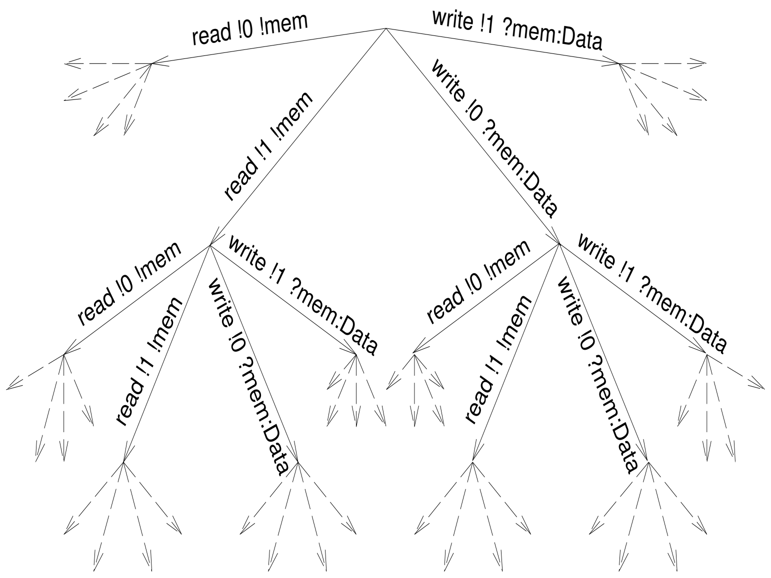

Figure 4.5: Unification of Shared Memory

especially useful for the unification of constraints in the constraint-oriented specification style [52]. However, the || operator only supports unification w.r.t. trace preorder ($\leq_{tr}$), which is but a weak notion of development.

We have proposed two new operators for LOTOS to support unification with respect to refinement by reduction and extension. The latter operator, $\bowtie$, was inspired by the *specification merge operator*, $\oplus$, in a pioneering paper on incremental specification [25]. Our $\bowtie$ operator is an improvement of the $\oplus$-operator, in the sense that it can deal with non-deterministic specifications.

The problem of composing specifications with respect to reduction and extension have been reported on before. For processes modelled by acceptance trees, an algorithm to unify such processes with respect to reduction is given in [31]. In order to apply this algorithm to LOTOS specifications a denotational semantics in terms of acceptance trees must be provided. In [32] and in [35] algorithms are given that can be used to compose two specifications, such that the composition is an extension of both. The first algorithm applies to LTSs, but uses acceptance trees as an intermediate model. The second algorithm applies to rooted failure trees (RFTs), which can provide a denotational semantics for a subset of LOTOS. In contrast to these composition methods, we have given an operational semantics for unification. This enables us to unify specifications on the specification language level, rather than on the semantic level. This is useful for simulation purposes.

The general framework for consistency and unification will allow us to investigate more instantiations of consistency (for example with implementation relations based on action refinement) in future research. Further, we intend to develop software tools for simulation and construction of unification, and for assessing the consistency of specifications.

Chapter 5

Consistency Checking Mechanisms in Z

In this part we describe a general strategy for unifying two Z specifications. In order to increase its applicability, it is not specific to any particular ODP viewpoint, nor is it tied to any particular instantiation of the architectural semantics. We show how to use this unification to check the consistency of two viewpoints written in Z. This is illustrated with a number of examples, including an information viewpoint specification from OSI Management. The link into the general consistency checking framework is made by using the logical definition of consistency, which can be integrated into the framework as previously discussed.

5.1 Unifying Viewpoint Specifications in Z

In this section we describe a general strategy for unifying two Z specifications. As described above we would like unification of two specifications to yield the least common refinement of both viewpoints. Such least unification is what we will investigate in this chapter. Unification of Z specifications will therefore depend upon the Z refinement relation, which is given in terms of two separate components - data refinement and operation refinement, [41]. Two specifications will thus be consistent if their unification is *internally valid*, and for Z this holds when the unification is free from contradictions (assuming the specifications that were unified were both internally valid). Thus to check the consistency of two specifications, we check for contradictions within the unified specification.

Z is a state based FDT, and Z specifications consist of informal English text interspersed with formal mathematical text. The formal part describes the abstract state of the system (including a description of the initial state of the system), together with the collection of available operations, which manipulate the state. One Z specification refines another if the state schemas are data refinements and the operation schemas are operation refinements of the original specifications state and operation schemas. Details of the language and its refinement relation are contained in introductionary texts, for example [41, 42, 50].

The unification algorithm we describe is divided into three stages: normalization, common refinement (which we usually term unification itself), and re-structuring. This algorithm can be shown to be the least refinement of both viewpoints, [8]. Related work on the combination of Z specifications includes [2, 1].

Normalization identifies commonality between two different viewpoint specifications, and re-writes each specifications into a normal form suitable for unification in the following manner. Clearly, the two specifications that are to be unified have to represent the world in the same way within them (e.g. if an operation is represented by a schema in one viewpoint, then the other viewpoint has to use the same name for its (possibly more complex) schema too), and that the correspondences between the specifications have to have been identified by the specifiers involved.

These will be given by mappings that describe the naming, and other, conventions in force. Once the commonality has been identified, normalization re-names the appropriate elements of the specifications. Normalization will also expand data-type and schema definitions into a normal form. Examples of normalization are given in [41, 42].

Unification itself takes two normal forms and produces the least refinement of both. Because normalization will hide some of the specification structure introduced via the schema calculus, it is necessary to perform some re-structuring after unification to re-introduce the structure chosen by the specifier. We do not discuss re-structuring here.

5.1.1 State Unification

The purpose of state unification is to find a common state to represent both viewpoints. The state of the unification must be the least data refinement of the states of both viewpoints, since viewpoints represent partial views of an overall system description.

The essence of all constructions will be as follows. We unify declarations rather than types, so non-identical types with name clashes are resolved by re-naming, then we unify declarations as follows. If an element x is declared in both viewpoints as $x : S$ and $x : T$ respectively, then the unification will include a declaration $x : U$ where U is the least refinement of S and T. The type U will be the smallest type which contains a copy of both S and T. For example, if S and T can be embedded in some maximal type then U is just the union $S \cup T$.

Given two viewpoint specifications both containing the following fragment of state description given by a schema D:

$$
\begin{array}{|l}
\multicolumn{1}{l}{D} \\
x : S \\
\hline
pred_S \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{D} \\
x : T \\
\hline
pred_T \\
\hline
\end{array}
$$

we unify as follows

$$
\begin{array}{|l}
\multicolumn{1}{l}{D} \\
x : S \cup T \\
\hline
x \in S \implies pred_S \\
x \in T \implies pred_T \\
\hline
\end{array}
$$

whenever $S \cup T$ is well founded. If S and T cannot be embedded in a single type then the unification will declare x to be a member of the disjoint union of S and T, and the mechanism to describe disjoint unions has to be included in the unification. In these circumstances we again achieve the least refinement of both viewpoints.

Axiomatic descriptions are unified in exactly the same manner.

5.1.2 Operation Unification

Once the data descriptions have been unified, the operations from each viewpoint need to be defined in the unified specification. We assume all renaming of names visible to the environment has taken place. Unification of schemas then depends upon whether there are duplicate definitions. If an operation is defined in just one viewpoint, then it is included in the unification (with appropriate adjustments to take account of the unified state).

For operations which are defined in both viewpoint specifications, the unified specification should contain an operation which is the least refinement of both, w.r.t. the unified representation of state. The unification algorithm first adjusts each operation to take account of the unified state in the obvious manner, then combines the two operations to produce an operation which is a refinement of both viewpoint operations.

The unification of two operations is defined via their pre- and post-conditions. Given a schema it is always possible to derive its pre- and post-conditions, [33]. Given two schemas A and B representing operations, both applicable on some unified state, then the unification of A and B is:

$$\begin{array}{|l}\hline \mathcal{U}(A,B) \\ \vdots \\ \hline \mathit{pre}\ A \lor \mathit{pre}\ B \\ \mathit{pre}\ A \implies \mathit{post}\ A \\ \mathit{pre}\ B \implies \mathit{post}\ B \\ \hline \end{array}$$

where the declarations are unified in the manner of the preceding subsection. This definition ensures that if both pre-conditions are true, then the unification will satisfy both post-conditions. Whereas if just one pre-condition is true, only the relevant post-condition has to be satisfied. This provides the basis of the consistency checking method for object behaviour which we discuss below.

We show, in [8], that this construction is the least refinement of the two viewpoint specifications. It is also associative, allowing the natural extension of unification to an arbitrary finite number of viewpoints.

5.1.3 Example 1 - A classroom

As an illustrative example we perform state and operation unification on a simple specification of a classroom. The example consists of the state represented by the schema *Class*, and operation *Leave*. The two viewpoint specifications to be unified are:

$Max : \mathbb{N}$

$$\begin{array}{|l}\hline \mathit{Class} \\ d : \mathbb{P}\{1,2\} \\ \hline \#d \leq Max \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \mathit{Leave} \\ \Delta Class \\ p? : \{1,2\} \\ \hline p? \in d \\ d' = d \setminus \{p?\} \\ \hline \end{array}$$

$Min : \mathbb{N}$

$$\begin{array}{|l}\hline \mathit{Class} \\ d : \mathbb{P}\{2,3,4\} \\ \hline \#d \geq Min \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \mathit{Leave} \\ \Delta Class \\ p? : \{2,3,4\} \\ \hline \#d > Min + 1 \\ p? \in d \\ d' = d \setminus \{p?, 2\} \\ \hline \end{array}$$

As described above, we first unify the state model, i.e. the schema *Class* in this example, which becomes:

$$\begin{array}{|l}\hline \mathit{Class} \\ d : \mathbb{P}\{1,2\} \cup \mathbb{P}\{2,3,4\} \\ \hline d \in \mathbb{P}\{1,2\} \implies \#d \leq Max \\ d \in \mathbb{P}\{2,3,4\} \implies \#d \geq Min \\ \hline \end{array}$$

With this unified state model we can unify the operation *Leave* on this state. To do so we calculate the pre and post-conditions in the usual manner, and for this we need to expand the schema *Leave* into normal form in each viewpoint. This will involve, for example, declaring $p? : \mathbb{Z}$ and containing $p? \in \{1,2\}$ as part of the predicate for the description of *Leave* in the first viewpoint. The pre-condition of *Leave* in the first viewpoint is then $p? \in d \cap \{1,2\}$ (in fact this is the part of the pre-condition which is distinct from the pre-condition in the second viewpoint, the rest acting as a state invariant). Hence, the unified *Leave* becomes:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Leave}} \\
\hline
\Delta \mathit{Class} \\
p? : \mathbb{Z} \\
\hline
(p? \in d \cap \{1,2\}) \lor (p? \in d \cap \{2,3,4\} \land \# d > \mathit{Min} + 1) \\
(p? \in d \cap \{1,2\}) \implies d' = d \setminus \{p?\} \\
(p? \in d \cap \{2,3,4\} \land \# d > \mathit{Min} + 1) \implies d' = d \setminus \{p?, 2\} \\
\hline
\end{array}
$$

To show that the unified *Leave* is indeed a refinement of *Leave* in viewpoint one we will decorate elements in viewpoint one with a subscript one. We can then use the retrieve relation

$$
\begin{array}{|l}
\multicolumn{1}{l}{R_1} \\
\hline
\mathit{Class} \\
\mathit{Class}_1 \\
\hline
d_1 \in \{d\} \cap \mathbb{P}\{1,2\} \\
\hline
\end{array}
$$

to describe the refinement between the unified state and the state in the first viewpoint. To demonstrate the refinement is correct, we make the following deductions. Suppose $\mathit{pre}\ \mathit{Leave}_1 \land \Delta R_1 \land \mathit{Leave}$, we have to show the result of this schema is compatible with $\mathit{post}\ \mathit{Leave}_1$. Now if $\mathit{pre}\ \mathit{Leave}_1$, then $p? \in d_1 \in \{d\} \cap \mathbb{P}\{1,2\}$, and hence $d' = d \setminus \{p?\}$. Then $d_1' \in \{d'\} \cap \mathbb{P}\{1,2\} = \{d \setminus \{p?\}\} \cap \mathbb{P}\{1,2\}$. So $d_1' = d' \cap \{1,2\} = (d \setminus \{p?\}) \cap \{1,2\} = d_1 \setminus \{p?\}$, since by $\mathit{pre}\ \mathit{Leave}_1$, $p? \in \{1,2\}$. The deduction that $\mathit{pre}\ \mathit{Leave}_1 \land R_1 \implies \mathit{pre}\ \mathit{Leave}$ is similar. These two deductions complete the proof that the unification is a refinement of viewpoint one. The case for viewpoint two is symmetrical.

5.1.4 Example 2 - Dining Philosophers

As a second illustration of unification in Z, we shall consider the following viewpoint specifications of the dining philosophers problem. The dining philosophers problem, [18], is a classic problem in synchronisation. A group of N philosophers sit round a table, laid with N forks. There is one fork between each adjacent pair of philosophers. Each philosopher alternates between thinking and eating. To eat, a philosopher must pick up its right-hand fork and then the left-hand fork. A philosopher cannot pick up a fork if its neighbour already holds it. To resume thinking, the philosopher returns both forks to the table.

The synchronisation needed is that no philosopher thinks or eats forever, and that the philosophers must not starve through deadlock. The three viewpoint specifications defined are the philosophers, forks and tables viewpoints. The philosophers and forks describe individual philosopher and fork objects and the operations available on those objects. The table viewpoint describes a system constructed from those objects and the synchronisation mechanism between operations upon them. We shall then describe the unification of the three viewpoints.

Although this example is not one of an ODP system, it provides a suitable illustration of the issues involved in viewpoint specification and consistency checking.

The Philosophers Viewpoint

This viewpoint considers the specification from the point of view of a philosopher. A philosopher either thinks, eats or holds her right fork. Note that since the latter is just a state of mind (for a philosopher!) there is no need to describe the operations from a forks point of view at all in this viewpoint.

$$\mathit{PhilStatus} ::= \mathit{Thinking} \mid \mathit{HasRightFork} \mid \mathit{Eating}$$

A philosopher object is just defined by the state of the philosopher.

$$\begin{array}{|l}\hline \multicolumn{1}{l}{\mathit{PHIL}} \\ status : PhilStatus \\ \hline\end{array}$$

Initially a philosopher is thinking.

$$\begin{array}{|l}\multicolumn{1}{l}{\mathit{InitPHIL}} \\ \hline PHIL' \\ \hline status' = Thinking \\ \hline\end{array}$$

We can now describe the operations available. A thinking philosopher can pick up its right-hand fork.

$$\begin{array}{|l}\multicolumn{1}{l}{\mathit{GetRightFork}} \\ \hline \Delta PHIL \\ \hline status = Thinking \\ status' = HasRightFork \\ \hline\end{array}$$

Philosophers who hold their right fork can begin eating upon picking up their left-hand fork. Finally to resume thinking, a philosopher releases both forks.

$$\begin{array}{|l}\multicolumn{1}{l}{\mathit{GetLeftFork}} \\ \hline \Delta PHIL \\ \hline status = HasRightFork \\ status' = Eating \\ \hline\end{array} \qquad \begin{array}{|l}\multicolumn{1}{l}{\mathit{DropForks}} \\ \hline \Delta PHIL \\ \hline status = Eating \\ status' = Thinking \\ \hline\end{array}$$

The Forks Viewpoint

This viewpoint specifies a fork object. Each fork is either free or busy. The fact that the philosopher might change state when a fork is picked up or dropped does not concern forks.

$$ForkStatus \ ::= \ Free \mid Busy$$

The state of the fork is given by:

$$\begin{array}{|l}\multicolumn{1}{l}{\mathit{FORK}} \\ \hline fstatus : ForkStatus \\ \hline\end{array}$$

Initially a fork is free.

$$\begin{array}{|l}\multicolumn{1}{l}{\mathit{InitFORK}} \\ \hline FORK' \\ \hline fstatus' = Free \\ \hline\end{array}$$

We can now describe the operations available. A free fork can be picked up, and both forks can be released.

$$\begin{array}{|l}\multicolumn{1}{l}{\mathit{Acquire}} \\ \hline \Delta FORK \\ \hline fstatus = Free \\ fstatus' = Busy \\ \hline\end{array} \qquad \begin{array}{|l}\multicolumn{1}{l}{\mathit{Release}} \\ \hline \Delta FORK \\ \hline fstatus = Busy \\ fstatus' = Free \\ \hline\end{array}$$

The Tables Viewpoint

This viewpoint has a number of schemas from the other viewpoints as parameters, these are given as empty schema definitions. Upon unification the non-determinism in this viewpoint will be resolved by the other viewpoint specifications, and thus unification will allow functionality extension of these parameters.

The parameters we require are:

$$\text{---}\ PHIL\ \text{---}$$

$$\text{---}\ InitPHIL\ \text{---}$$

$$\begin{array}{|l}\hline \multicolumn{1}{l}{GetRightFork} \\ \Delta PHIL \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \multicolumn{1}{l}{GetLeftFork} \\ \Delta PHIL \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \multicolumn{1}{l}{DropForks} \\ \Delta PHIL \\ \hline\end{array}$$

$$\text{---}\ FORK\ \text{---}$$

$$\text{---}\ InitFORK\ \text{---}$$

$$\begin{array}{|l}\hline \multicolumn{1}{l}{Acquire} \\ \Delta FORK \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \multicolumn{1}{l}{Release} \\ \Delta FORK \\ \hline\end{array}$$

The system from the table viewpoint is defined by a collection of fork and philosopher objects:

$$\begin{array}{|l}\hline \multicolumn{1}{l}{Table} \\ forks : 1..N \rightarrow FORK \\ phils : 1..N \rightarrow PHIL \\ \hline\end{array}$$

Initially the table consists of forks and philosophers all in their respective initial states.

$$\begin{array}{|l}\hline \multicolumn{1}{l}{InitTable} \\ Table' \\ \hline \exists\, InitFORK, InitPHIL \bullet \operatorname{ran} forks' = \{\theta InitFORK\} \wedge \operatorname{ran} phils' = \{\theta InitPHIL\} \\ \hline\end{array}$$

Here we use promotion (ie the θ operator) in the structuring of viewpoints, which allows an operation defined on an object in one viewpoint to be *promoted* up to an operation defined over that object in another viewpoint. As we can see, this can be used effectively to reference schemas in different viewpoints without their full definition.

In order to define operations on the table, we define a schema $\Phi Table$ which will allow individual object operations to be defined in this viewpoint. See [42] for a discussion of the use of promotion.

$$\begin{array}{|l}\hline \multicolumn{1}{l}{\Phi Table} \\ \Delta Table \\ \Delta PHIL \\ \Delta FORK \\ m? : 1..N \\ n? : 1..N \\ \hline phils(n?) = \theta PHIL \\ phils' = phils \oplus \{phils(n?) = \theta PHIL'\} \\ forks(m?) = \theta FORK \\ forks' = forks \oplus \{forks(m?) = \theta FORK'\} \\ \hline\end{array}$$

Note that we use two inputs $m?, n?$, because we want to control later the synchronisation between operations on forks and those on philosophers. System operations to get the left and right forks, and to drop both forks can now be defined.

$GLF \mathrel{\widehat=} (\Phi Table \wedge GetLeftFork \wedge Acquire \wedge [\, n?, m? : 1..N \mid m? = n?\,]) \setminus (\Delta FORK, \Delta PHIL)$

$GRF \mathrel{\widehat{=}} (\Phi Table \land GetRightFork \land Acquire \land [\, n?, m? : 1..N \mid m? = (n? \bmod N + 1)\,]) \setminus (\Delta FORK, \Delta PHIL)$

$DF \mathrel{\widehat{=}} (\Phi Table \land DropForks \land Release \land [\, n?, m? : 1..N \mid m? = n?\,]) \setminus (\Delta FORK, \Delta PHIL)$

The last schema in each conjunction performs the correct synchronisation between the individual object operations. For example, it forces operations *GetLeftFork* and *Acquire* to be performed on $phils(n?)$ and $forks(n?)$ in *GLF*; whilst in *GRF*, *GetRightFork* will be performed on $phils(n?)$ and *Acquire* on $forks(n? \bmod N + 1)$.

Unifying the Philosophers and Forks Viewpoints

Since the fork and philosopher object descriptions are independent, ie there are no state or operation schemas in common, the unification of these two viewpoints is just the concatenation of the two specifications. We do not re-write that concatenation here.

Unifying the Table, Philosophers and Forks Viewpoints

The Table specification does have commonality with the other two viewpoints. For each state or operation schema defined in two viewpoints (ie the Table and one other), we build one schema in the unification. In fact, the separation and object-based nature (in a loose sense) of this example means that we will not make extensive use of unification by pre- and post-conditions. This is desirable, since it reduces the search for contradictions in the consistency checking phase. In fact, our experiences with viewpoint specifications confirms that such a viewpoint methodology is really only feasible if one adopts this object-based approach.

For example, the schema *FORK* defined in the Table viewpoint is just a parameter from the fork viewpoint, and consequently its unification will just be:

$$\begin{array}{|l}
\multicolumn{1}{l}{FORK} \\
\hline
fstatus : ForkStatus \\
\hline
\end{array}$$

Similarly the unification of *GetLeftFork* from the Table and Philosophers viewpoint is

$$\begin{array}{|l}
\multicolumn{1}{l}{GetLeftFork} \\
\hline
\Delta PHIL \\
\hline
status = HasRightFork \\
status' = Eating \\
\hline
\end{array}$$

since the pre-condition of *GetLeftFork* in Table is just false. Notice that this provides a mechanism in Z by which to achieve functionality extension across viewpoints in a manner previously not supported.

The remaining schemas can be unified in the obvious manner.

5.1.5 Example 3 - OSI Management

Our third example involves the application of Z in the ODP information viewpoint to the modelling of OSI Management, which has been investigated by a number of researchers [44, 53]. We show here how unification and consistency checking can be used with such modelling techniques by considering viewpoint specifications of sieve managed objects and their controlling CME agent.

To illustrate some of the techniques we consider two viewpoint specifications of an event reporting sieve object together with a third viewpoint which describes a CME agent and its manipulation of the sieve objects. In this simplified model we have not considered the relationships between managed objects, although a complete presentation would include their specification.

Within ODP, an information object template is modelled by a Z specification. An information object instance is then modelled as a Z specification instance (i.e. a specification complete with initialization of variables), and an ODP action is described by a Z operation.

The variable declarations in a state schema represent the attributes of a managed object. The state schema also describes the state invariant which constrains the values of the attributes. The initialization schema (e.g. *InitSieve*) constrains the initial values of the state schema.

A managed object definition cannot include a *Create* operation, since before it is created a managed object cannot perform any operation, including *Create* itself. However, by including a *Create* operation in the CME agent viewpoint as we do below, we can describe formally the interaction between *Create* and the sieve managed object definition.

We have not considered any particular flavours, or design considerations, to differentiate between the first two viewpoints. Their purpose here is to represent to view of the system from similar standpoints.

Viewpoint 1 : Sieve object

To describe the sieve object, we first declare the types. *SieveConstruct* is used in the event reporting process, its internal structure is left unspecified at this stage, hence it is defined as a given set.

$$[SieveConstruct]$$

The remaining types are declared as enumerated types.

$$\begin{array}{lcl} Operational & ::= & disabled \mid active \mid enabled \mid busy \\ Admin & ::= & locked \mid unlocked \mid shuttingdown \\ Event & ::= & nothing \mid enrol \mid deenrol \\ Status & ::= & created \mid deleted \end{array}$$

Status models the life-cycle of the sieve object, and is used as an internal mechanism to control which operations are applicable at a given point within an object's existence. The state schema defines the attributes of the sieve object, here there are no constraints upon them; and the initialization describes their initial values.

$$\begin{array}{|l} \multicolumn{1}{l}{Sieve} \\ \hline opstate : Operational \\ sico : SieveConstruct \\ adminstate : Admin \\ status : Status \\ \hline \end{array} \qquad \begin{array}{|l} \multicolumn{1}{l}{InitSieve} \\ \hline Sieve \\ \hline opstate = active \\ adminstate = unlocked \\ \hline \end{array}$$

We describe two of the operations available within a sieve object (for a full description of operations see [44]). The first is an operation to delete a sieve. Upon deletion a sieve sends a *deenrol* notification to its environment, and moves into a state where no further operations can be applied.

$$\begin{array}{|l} \multicolumn{1}{l}{Delete} \\ \hline \Delta Sieve \\ notification! : Event \\ \hline status = created \\ notification! = deenrol \\ status' = deleted \\ \hline \end{array}$$

We define a relation *filter* to represent criteria to decide which events to filter out and which to pass on

$$\begin{array}{|l} filter : Event \leftrightarrow SieveConstruct \end{array}$$

and the *Filter* schema represents the operation to perform the filtering.

$$
\begin{array}{|l}
\hline
\textit{Filter} \\
\hline
\Xi Sieve \\
event? : Event \\
notification! : Event \\
\hline
status \neq deleted \\
opstate = active \land adminstate = unlocked \\
(event?, sico) \in filter \Rightarrow notification! = event? \\
(event?, sico) \notin filter \Rightarrow notification! = nothing \\
\hline
\end{array}
$$

Viewpoint 2 : Sieve object

To illustrate some of the unification techniques we now describe a second view of the same sieve object. First of all we declare the types

$$[SieveConstruct]$$

$$
\begin{array}{lll}
Operational & ::= & disabled \mid active \mid enabled \mid busy \\
Admin & ::= & locked \mid unlocked \mid shuttingdown \\
Event & ::= & nothing \mid enrol \mid deenrol \\
Status & ::= & being\_created \mid created \mid deleted
\end{array}
$$

Notice that in this viewpoint *Status* includes an additional value, *being_created.* The state schema and its initialization are then declared.

$$
\begin{array}{|l}
\hline
\textit{Sieve} \\
\hline
opstate : Operational \\
sico : SieveConstruct \\
adminstate : Admin \\
status : Status \\
\hline
opstate \in \{active, disabled\} \\
adminstate \in \{locked, unlocked\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{InitSieve} \\
\hline
Sieve \\
\hline
status = being\_created \\
opstate = active \\
adminstate = unlocked \\
\hline
\end{array}
$$

The change of state of a sieve object from *being_created* to *created* is governed by an internal operation, which can occur spontaneously. This change in *status* allows other operations to be invoked subsequently apart from the *Enrol* operation itself.

$$
\begin{array}{|l}
\hline
\textit{Enrol} \\
\hline
\Delta Sieve \\
notification! : Event \\
\hline
status = being\_created \\
status' = created \\
notification! = enrol \\
\hline
\end{array}
$$

In this viewpoint a relation *newfilter* defines the filtering criteria, and the operation *Filter* performs the filtering.

$$
\begin{array}{|l}
newfilter : Event \leftrightarrow SieveConstruct
\end{array}
$$

$$\begin{array}{|l}
\textit{Filter} \\
\hline
\Xi Sieve \\
event? : Event \\
notification! : Event \\
\hline
status = created \\
opstate = active \land adminstate = unlocked \\
(event?, sico) \notin newfilter \Rightarrow notification! = nothing \\
\hline
\end{array}$$

Viewpoint 3 : CME agent

The final viewpoint is a description of a controlling CME agent. For our purposes here we present a very simplified version of an agent which consists of a number of sieve managed objects. We then show how we can promote the *Delete* operation defined on individual sieve objects, and define a *Create* operation to instantiate sieve objects as required.

This viewpoint has a number of schemas from the other viewpoints as parameters, these are given as empty schema definitions. Upon unification the under-specification of these parameters in this viewpoint will be resolved by the other viewpoint specifications, and thus unification will allow functionality extension of these parameters. The parameters we require are:

$$\begin{array}{|l}
\textit{Sieve} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\textit{InitSieve} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{Delete} \\
\hline
\Delta Sieve \\
\hline
\end{array}$$

We declare types to represent the set of object classes and set of object identifiers respectively. A *CMEagent* is then modelled as a collection of sieve objects, and initially no sieve objects have been created, so the range of *sieves* cannot include a state described by *Sieve*

$$[Class, Id]$$

$$\begin{array}{|l}
\textit{CMEagent} \\
\hline
sieves : Class \times Id \nrightarrow Sieve \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\textit{InitCMEagent} \\
\hline
CMEagent \\
\hline
\nexists Sieve \bullet \theta Sieve \in \operatorname{ran} sieves \\
\hline
\end{array}$$

In order to define CME agent operation, we define a schema $\Phi CMEagent$ which will allow individual object operations to be defined in this viewpoint.

$$\begin{array}{|l}
\Phi CMEagent \\
\hline
\Delta CMEagent \\
\Delta Sieve \\
objectclass? : Class \\
sieveid? : Id \\
\hline
sieves(objectclass?, sieveid?) = \theta Sieve \\
sieves' = sieves \oplus \{sieves(objectclass?, sieveid?) = \theta Sieve'\} \\
\hline
\end{array}$$

An agent operation to delete a specific sieve object can now be defined by promotion of the *Delete* parameter specified in another viewpoint. The other managed object operations are promoted in a similar fashion.

$$DeleteSieve \mathrel{\widehat{=}} (\Phi CMEagent \land Delete) \setminus (\Delta Sieve)$$

Finally the *Create* operation can be defined. Notice this is not part of the sieve specification, so we have preserved the concept that *Create* must occur before any operation in the sieve specification can be applied.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Create}} \\ \hline
\Delta CMEagent \\
\Delta Sieve, \Delta InitSieve \\
objectclass? : Class \\
sieveid? : Id \\ \hline
sieves(objectclass?, sieveid?) \neq \theta Sieve \\
sieves' = sieves \oplus \{sieves(objectclass?, sieveid?) = \theta InitSieve'\} \\ \hline
\end{array}
$$

Unification of Viewpoints

To describe the unification of viewpoints, we decorate with subscripts, so for example $Filter_1$ is the schema *Filter* from the first viewpoint. To unify viewpoints 1 and 2 we first unify the state. The only conflict in the declarations are due to differing types $Status_1$ and $Status_2$. To resolve this conflict, the type *Status* in the unification is taken as the least refinement of $Status_1$ and $Status_2$ (i.e. $Status_1 \cup Status_2$), and state unification is applied to the schema *Sieve*. Hence, in addition to the declarations which are not in conflict, the unification will contain the following:

$$Status ::= being\_created \mid created \mid deleted$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Sieve}} \\ \hline
opstate : Operational \\
sico : SieveConstruct \\
adminstate : Admin \\
status : Status \\ \hline
status \in \{created, deleted\} \Rightarrow true \\
status \in \{being\_created, created, deleted\} \Rightarrow \\
\qquad (opstate \in \{active, disabled\} \wedge adminstate \in \{locked, unlocked\}) \\ \hline
\end{array}
$$

Upon simplification the schema *Sieve* becomes

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Sieve}} \\ \hline
opstate : Operational \\
sico : SieveConstruct \\
adminstate : Admin \\
status : Status \\ \hline
opstate \in \{active, disabled\} \\
adminstate \in \{locked, unlocked\} \\ \hline
\end{array}
$$

In a similar fashion we unify $InitSieve_1$ and $InitSieve_2$, which simplifies to

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitSieve}} \\ \hline
Sieve \\ \hline
status = being\_created \\
opstate = active \\
adminstate = unlocked \\ \hline
\end{array}
$$

The *Delete* and *Enrol* schemas are defined in just one viewpoint. Hence, both these schemas are included in the unification (with adjustments due to the unified state schema *Sieve*). Similarly the unification contains both relations *filter* and *newfilter*.

To unify $Filter_1$ and $Filter_2$ we first adjust $Filter_1$ due to the unified state schema. The predicate part of $Filter_1$ is then

$$status \in \{created, deleted\} \Rightarrow (status \neq deleted \land opstate = active \land adminstate = unlocked)$$

Calculation of the pre-conditions $preFilter_1 \lor preFilter_2$ then simplifies to

$$(status = created \land opstate = active \land adminstate = unlocked)$$

Thus the unification of $Filter_1$ and $Filter_2$ is then given by:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Filter}} \\
\hline
\Xi Sieve \\
event? : Event \\
notification! : Event \\
\hline
status = created \land opstate = active \land adminstate = unlocked \\
(event?, sico) \in filter \Rightarrow notification! = event? \\
(event?, sico) \notin filter \Rightarrow notification! = nothing \\
(event?, sico) \notin newfilter \Rightarrow notification! = nothing \\
\hline
\end{array}
$$

To complete the unification we must unify this specification with the third viewpoint which specified the CME agent. The parameters in the third viewpoint have their functionality extended upon unification. For example, the schema *InitSieve* defined in the third viewpoint is just a parameter from the other viewpoints, and consequently its unification will just be:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitSieve}} \\
\hline
Sieve \\
\hline
status = being\_created \\
opstate = active \\
adminstate = unlocked \\
\hline
\end{array}
$$

The complete unification is then achieved in the obvious manner, by expanding *Sieve*, *InitSieve* and *Delete* and including *Enrol* and *Filter* along with the CME agent operations.

5.2 Consistency Checking of Viewpoint Specifications in Z

The mechanism for unifying two Z specification yields a consistency checking process. A specification is consistent if it does not contain specifications of entities which cannot possibly exist. That is, given a proof system for Z, with a validity relation $\vdash$, a specification is said to be consistent if it is not possible to prove $S \vdash false$. For example, a Z specification of a function will be inconsistent if the predicate part of its axiomatic definition contradicts the fact that it was declared as a function. Another way in which inconsistencies can arise in Z specifications is in the definition of free types. Examples of how such inconsistencies can occur are given in [4, 48, 45]. In general, it is undecidable whether or not a set of axioms given in a Z specification is consistent. [4] discusses sufficient conditions for the consistency of certain combinations of Z paragraphs, in particular axiomatic definitions, given sets and free types.

In addition, consistency usually refers to consistency of the state model, i.e. for a given state there exists at least one possible set of bindings that satisfies the state invariant, [41, 42]. With this consistency condition comes a requirement to prove the Initialisation theorem (see below), which asserts there exists a state that satisfies the initial conditions of the model. Due to an ODP requirement associated with multiple viewpoints, we also require operation consistency, because an ODP conformance statement in Z corresponds to an operation schema(s), [47]. A conformance statement is behaviour one requires at the location that conformance is tested. Thus a given behaviour (i.e. occurrence of an operation schema) conforms if the post-conditions and invariant predicates are satisfied in the associated Z schema. That is, operations defined in two viewpoints are consistent if whenever both operations are applicable, their post-conditions agree. Hence, operations in a unification will be implementable whenever each operation has consistent post-conditions on the conjunction of its pre-conditions.

Thus a viewpoint consistency check in Z involves checking the unified specification for contradictions, and has 5 components: axiom, axiomatic, state and operation consistency in addition to the Initialisation theorem. Assuming the individual viewpoints themselves are consistent, the components then take the following form.

Axiom Consistency : Axioms constrain existing global constants. Hence, to check for consistency of the two viewpoints, axioms from one viewpoint have to be checked against the second viewpoint w.r.t. any terms appearing in the axioms which are defined in the second viewpoint. If an axiom contains no terms appearing in other viewpoints, its consistency checking requirements are discharged.

State Consistency : Consider the general form of state unification given earlier:

$$\begin{array}{|l}
\multicolumn{1}{l}{D} \\
\hline
x : S \cup T \\
\hline
x \in S \implies pred_S \\
x \in T \implies pred_T \\
\hline
\end{array}$$

This state model is consistent as long as both $pred_S$ and $pred_T$ can be satisfied for $x \in S \cap T$.

Axiomatic Consistency : Similar to state consistency.

Operation Consistency : Consistency checking also needs to be carried out on each operation in the unified specification. The definition of operation unification means that we have to check for consistency when both pre-conditions apply. That is, if the unification of A and B is denoted $\mathcal{U}(A, B)$, we have:

$$pre\ \mathcal{U}(A, B) = pre\ A \lor pre\ B, \quad post\ \mathcal{U}(A, B) = (pre\ A \Rightarrow post\ A) \land (pre\ B \Rightarrow post\ B)$$

So the unification is consistent whenever the post conditions agree on the conjunction of the pre-conditions, $(pre\ A \land pre\ B)$.

Initialisation Theorem : The Initialisation Theorem is a consistency requirement of all Z specifications. It asserts that there exists a state of the general model that satisfies the initial state description, formally it takes the form:

$$\vdash \exists\, State \bullet InitState$$

For the unification of two viewpoints to be consistent, clearly the Initialisation Theorem must also be established for the unification.

The following result can simplify this requirement: Let $State$ be the unification of $State_1$ and $State_2$, and $InitState$ be the unification of $InitState_1$ and $InitState_2$. If the Initialisation Theorem holds for $State_1$ and $State_2$, then state consistency of $InitState$ implies the Initialisation Theorem for $State$. In other words, it suffices to look at the standard state consistency of $InitState$.

If, however, $InitState$ is a more complex description of initiality (possibly still in terms of $InitState_1$ and $InitState_2$), the Initialisation Theorem expresses more than state consistency of $Initstate$, and hence will need validating from scratch.

5.2.1 Example 1 - The classroom

State Consistency : The unified state in this example was given by

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Class}} \\
\hline
d : \mathbb{P}\{1,2\} \cup \mathbb{P}\{2,3,4\} \\
\hline
d \in \mathbb{P}\{1,2\} \implies \# d \leq Max \\
d \in \mathbb{P}\{2,3,4\} \implies \# d \geq Min \\
\hline
\end{array}
$$

To show consistency, we need to show that if $d \in \mathbb{P}\{1,2\} \cap \mathbb{P}\{2,3,4\}$, then both $\# d \leq Max$ and $\# d \geq Min$ hold. Suppose the class consisted of just the element 2, i.e. $d = \{2\}$. Both pre-conditions in the unified state, $d \in \mathbb{P}\{1,2\}$ and $d \in \mathbb{P}\{2,3,4\}$, now hold giving the state invariant $Min \leq \# d \leq Max$. Thus the consistency of the viewpoint specifications of the classroom requires that $Min \leq 1 \leq Max$. This type of consistency condition should probably fall under the heading of a *correspondence rule* in ODP, [30], that is a condition which is necessary but not necessarily sufficient to guarantee consistency.

Operation Consistency : In this example, this amounts to checking the operation *Leave* when

$$(p? \in d \cap \{1,2\}) \wedge (p? \in d \cap \{2,3,4\} \wedge \# d > Min + 1)$$

In these circumstances, the two post-conditions are $d' = d \setminus \{p?\}$ and $d' = d \setminus \{p?, 2\}$. These two pre-conditions apply when $p? = 2$ and $2 \in d$. A consistency check has to be applied for all possible values of d. For example, let $d = \{1,2\}$, then $d' = d \setminus \{p?\}$. If further $\# d > Min + 1$, then in addition we have $d' = d \setminus \{p?, 2\}$. These two conditions are consistent (since $p? = 2$) regardless of Max or Min.

Let $d = \{2\}$, then both pre-conditions apply iff $Min < 0$, in which case the post-conditions are $d' = d \setminus \{2\}$ and $d' = d \setminus \{2\}$, and thus consistent.

Hence the two viewpoint specifications are consistent whenever the correspondence rule $Min \leq 1 \leq Max$ holds.

5.2.2 Example 2 - Dining Philosophers

Inspection of the unification in the Dining Philosophers example shows that both state and operation consistency is straightforward (note, however, that with non-object based viewpoint descriptions of this example, consistency checking is a non-trivial task, this points the need for further work on specification styles to support consistency checks). Hence, consistency will follow once we establish the Initialization Theorem for the unification.

The Initialization Theorem for the unification is

$$\vdash \exists \, Table' \bullet InitTable$$

which expands to

$$\begin{aligned}\vdash \exists forks' : 1..N \rightarrow FORK, phils' : 1..N \rightarrow PHIL \bullet \exists InitFORK, InitPHIL \bullet \\ \operatorname{ran} forks' = \{\theta InitFORK\} \wedge \operatorname{ran} phils' = \{\theta InitPHIL\}\end{aligned}$$

Upon simplification this becomes

$$\vdash \exists forks' : 1..N \rightarrow FORK, phils' : 1..N \rightarrow PHIL \bullet \operatorname{ran} forks' = \{Free\} \wedge \operatorname{ran} phils' = \{Thinking\}$$

which clearly can be satisfied. Hence the viewpoint descriptions given for the dining philosophers are indeed consistent.

5.2.3 Example 3 - OSI Management

State Consistency : Consider the state schema *Sieve*. The unified schema across all three viewpoints was given by:

$$\begin{array}{|l}
\textit{Sieve} \\
\hline
opstate : Operational \\
sico : SieveConstruct \\
adminstate : Admin \\
status : Status \\
\hline
status \in \{created, deleted\} \Rightarrow true \\
status \in \{being_created, created, deleted\} \Rightarrow \\
\qquad (opstate \in \{active, disabled\} \wedge adminstate \in \{locked, unlocked\}) \\
\hline
\end{array}$$

From this it can be seen that both predicates $true$ and $(opstate \in \{active, disabled\} \wedge adminstate \in \{locked, unlocked\})$ can be satisfied for $status \in \{created, deleted\} \cap \{being\_created, created, deleted\}$, which is the requirement for consistency for this state schema.

Operation Consistency : Consider the unification of the *Filter* operation schema. From the unification we found that

$$preFilter_1 = preFilter_2 = (status = created \wedge opstate = active \wedge adminstate = unlocked)$$

Thus to show operation consistency we have to show that under this pre-condition, we have

$$\begin{aligned}(((event?, sico) \in filter \Rightarrow notification! = event?) \wedge ((event?, sico) \notin filter \Rightarrow notification! = nothing)) \\ = ((event?, sico) \notin newfilter \Rightarrow notification! = nothing)\end{aligned}$$

It is easy to show that a necessary, but not sufficient, condition for the consistency of this operation is

$$\forall (event?, sico) \in Event \times Event \bullet (event?, sico) \in filter \wedge (event?, sico) \notin newfilter \Rightarrow event? = nothing$$

Thus the consistency of *Filter* requires this condition to be maintained. By giving specifiers explicit notification of which relationships between objects in the viewpoints need preserving, this constraint can then be used by the individual viewpoint specifiers to ensure that any further refinement of the viewpoints do not violate consistency.

Inspection of the unification of the CME agent with the sieve object shows that both state and operation consistency carry over from the state and operation consistency of the unification of viewpoints 1 and 2. Hence, consistency will follow once we establish the Initialization Theorem for the unification of all three viewpoints.

The **Initialization Theorem** for the unification is

$$\vdash \exists\, CMEagent \bullet InitCMEagent$$

which expands to

$$\vdash \exists\, sieves : Class \times Id \rightarrowtail\!\!\!\!\!\rightarrow Sieve \bullet \nexists\, Sieve \bullet \theta Sieve \in \operatorname{ran} sieves$$

Upon simplification this becomes

$$\vdash \exists\, sieves : Class \times Id \rightarrowtail\!\!\!\!\!\rightarrow Sieve \bullet \langle status, opstate, adminstate, sico\rangle \notin \operatorname{ran} sieves$$

The final term describes a set of bindings, and it is clear that such a function *sieves* exists. Hence, the viewpoint descriptions given for the CME agent and sieve objects are indeed consistent.

The consistency checking mechanism works well for small to medium sized Z specifications. For larger specifications additional structure is needed in order that the consistency checking strategy can be scaled up. [17] shows how support for this can be provided by using object oriented variants of Z. These object based methodologies look likely to provide sufficient structure for the consistency checking to remain feasible, this is an area of ongoing research and is discussed briefly below.

5.3 Software Engineering Issues

The previous section elucidated the consistency checking requirements for unified viewpoint specifications. Given these requirements, it is necessary to seek software engineering strategies that make viewpoint decomposition feasible. By feasible we mean it is possible to describe viewpoints which are consistent and that the effort involved in consistency checking is minimised. In this section we explore some of the software engineering consequences of the consistency checking requirements. There are two initial possibilities for how to write viewpoint specifications:

(a) viewpoint description and analysis will work with arbitrary specifications and specification styles;

(b) some style guidelines or further methodology is needed for the process to become feasible.

If one adopts position (a), then there are some issues which need to be addressed:

No encapsulation of state and operations When the state is unified, all operations acting on that state are adjusted to take account of the unified state. Hence, during unification of an operation, two adjustments are made: the first due to the unified state (declarations are updated to take account of the unified state) and the second due to the change in pre- and post-conditions. Therefore, to keep track of the consistency checking requirements the operations need to be encapsulated with the state they affect. Without this consistency checking is possible, but unrealistic for larger examples.

No operation set representation As Zave noted in [56], Z provides no means of representing the operation set of a specification (i.e. the set of operations visible by the environment). The consequences of this for unification is that if an operation schema appears in both viewpoints, then it *has* to be unified, since there is no means to tell whether it was defined (in either viewpoint) for internal structuring purposes only. If there was such information available, then internal structuring schemas could be re-named and just operations in the operational set unified.

Correspondence rules In order to describe the relation between viewpoints, the RM-ODP includes a notion of correspondence rule. Part of their purpose is to identify the commonality between the specifications, and describe any possible renamings between them. Any viewpoint methodology will need to include mappings such as these. The limited structure in an ordinary Z specification makes a succinct naming impossible for correspondences, since, for any non-trivial systems, it is likely that a correspondence will wish to name more than one state/operation.

Viewpoint encapsulation The work of [2] indicates that in a non-object approach a large number of re-namings and re-workings of the viewpoints have to be undertaken *during* the unification process. This appears to be because the boundaries of the problem are not well defined, leading to viewpoint specifiers referencing and defining similar aspects of the same entity. Again this is a manifestation of the lack of encapsulation when defining the area of concern for each viewpoint specifier.

From case studies undertaken and consideration of these issues it seems that viewpoint description without any style guidelines is unlikely to be practical for anything other than small examples. Encapsulation and identity are central to the practical realization of the viewpoints model. Both of these facets could be provided by a number of software engineering methodologies, however, object orientation is an obvious choice. Many of the problems identified above can be resolved if one adopts an object oriented approach.

Encapsulation of state and operation The over-riding advantage of object oriented methods is their encapsulation of state and operation. This will clearly delimit the consistency checking requirements within a unification, with each unified object generating local consistency checking requirements which do not escalate to global consistency checking problems.

Operation set representation Some, although not all, object oriented methodologies in Z provide the ability to specify an operation set, or visibility list, [39, 19], which partitions all the defined operations into disjoint sets of visible and internal operations. If this is provided, then the issue of operation set representation is completely resolved. Even if such a visibility list is not provided by the language used, the encapsulation that comes with object orientation still provides the opportunity for partial resolution of the problem. In an object-based world it is likely that a viewpoint partitioning will include the internal specification of the behaviour of an object in only one of the viewpoints. The other viewpoints will then (possibly) reference objects from viewpoints as parameters, or place constraints on the use of those objects. Hence, in these circumstances the unification of two internal representations is unlikely to occur, and so the issue of operation set representation would not occur. Of course, if the internal specification of an object's behaviour did occur in more than one viewpoint, the need for a visibility list would then arise again.

Correspondence rules Identity is a key property of an object, and will allow correspondence rules to relate suitably complex parts and combination of parts of the viewpoint descriptions in a manner which is not currently supported in Z.

These considerations naturally lead to a choice of an object-based or object oriented language for viewpoint decomposition, where each viewpoint specifies a number of interacting objects. Full OO is not necessarily needed, however, if it is available then OO facilities such as inheritance can be exploited. It is preferable that only one viewpoint specifies the internal representation of a given object, and references to objects from one viewpoint will appear as parameters, either as inheritance within another object, or as an abstraction purely in terms of object or method names. The next section investigates the support available in Z for this approach.

5.4 Using Object Oriented Techniques

The previous section indicated that an object oriented style of specification is particularly suitable for viewpoint descriptions, and indeed the RM-ODP has adopted such an approach. There have been a number of different approaches proposed for providing Z with object oriented facilities. These include the provision of object oriented style guidelines, and extensions to Z to allow fully object oriented specifications. Examples of using Z in an object oriented style include: Hall's style [22, 23]; ZERO [54]; and the ODP architectural semantics [30]. Examples of object oriented extensions to Z include: Object-Z [19]; ZEST [15]; MooZ [39]; OOZE [3]; Schuman & Pitt [46]; and Smith [49]. See [51] for a summary and comparison of several approaches.

Z itself is not object oriented because it does not provided sufficient support for either encapsulation or inheritance. However, it is also possible for Z to be used in an object based fashion, see [42] for a discussion, although there is nothing to keep the specifier within an object based style in contrast to the style guidelines above.

The ODP standardisation initiative requires the use of (near) standardised formal methods, hence the architectural semantics uses Z as opposed to any object oriented variant of that language. However, given that ODP has adopted the object oriented paradigm, there is obvious interest in object oriented variants that can support the required ODP modelling concepts. In particular, Object-Z and ZEST are receiving attention within the ODP community as a specification medium. However, all the object oriented extensions to Z have an unstable definition, or lack a full semantics, or both. Therefore, techniques with a flattening (or approximate flattening) into Z are of considerable interest to our work. By using such a technique we can define unification and consistency checking of viewpoints without compromising the necessity of a standardised formal description technique. Object-Z and ZEST are suitable from this perspective.

Of the Z guidelines the work of Hall is the most general. The style adds no new features to Z, however, there are conventions for writing an object oriented specification. He also provides conventions for modelling classes and their relationships, and, in addition, there is formal support for inheritance through subtyping, [23]. In order to support encapsulation, the RM-ODP has adopted conventions for the use of Z within ODP. Here, encapsulation is achieved by letting each Z specification denote just one object. This achieves the required encapsulation, but clearly any specification of an aggregate of objects or interaction between objects cannot then be modelled within Z. Thus there is clearly a need to extend the framework offered by ODP by considering further style guidelines for the specification of collections of objects.

The unification techniques described above can be used with ZEST for the specification of viewpoints. To do so, use ZEST to describe viewpoints consisting of objects or aggregates of objects. The ZEST can then be flattened to Z, in order to generate the unification of the two viewpoints and to check for consistency. The unification can then easily be re-assembled into a ZEST specification if further object oriented development is required. Other object oriented variants of Z could equally have been used for the basis of this example, in particular, Object-Z would have provided a similar set of facilities as those we have called upon. Although unification is applied by first flattening the ZEST, it is important to note that the benefits of using object orientation are not lost by doing so. The encapsulation can be recovered, and the consistency checking requirements still lie within the boundaries defined by the object encapsulation.

We have undertaken a number of case studies in order to test the hypothesis that object oriented description is the preferable viewpoint specification medium, and our conclusions so far support this claim. The studies involving non-object based descriptions were significantly harder to check for consistency and much harder to specify in an independent fashion in the viewpoints, the specification in [2] is another indication of the difficulty of non-object based viewpoint specifications.

Conversely, the object based viewpoint descriptions were much more successful. When the viewpoints contain only references to objects defined in other viewpoints (as opposed to specifying any of its behaviour) consistency checking is relatively straightforward (although the viewpoints can still be inconsistent). If two viewpoints both contain (partial) descriptions of the same object, then there can be a non-trivial consistency checking process, however, due to encapsulation the boundaries that inconsistency can arise within are well defined.

Examples were undertaken in a number of styles. The style of the object oriented variant chosen did not significantly affect the success or otherwise of the viewpoint specification or unification. There are clear merits in using Z without extended syntax, particularly in the use within ISO initiatives. To that extent, using the work of Hall and Smith have clear advantages. Hall in particular offers formal and well-defined support for inheritance, which is lacking for some other Z object oriented variants. Smith's work at the moment is not sufficiently mature or accepted. The extended syntax approaches have advantages for the developer, who is then not constrained by conventions for embedding object orientation in Z, but only if a clear semantics, and preferably a flattening into Z, can be given.

5.4.1 Relation between Unification and Inheritance

It is important to recognise that unification is a 'horizontal' rather than 'vertical' development activity. By that we mean it is used to check development at a particular stage (consistency checking) or possibly to combine development specifications (unification), rather than a development activity that serves to define the implementation more closely (as in refinement or inheritance). Given that unification is a horizontal activity, one needs to describe the relationship between it and vertical development activities. The relationship between unification and refinement is well known since unification is based upon (least) refinement. We describe here the relation between inheritance and unification.

To do so we need a formal approach to inheritance and subtyping. Hall, [23], contains a discussion of known definitions in terms of both extensional and intensional semantics. Given that we are interested in behaviour of specifications, we shall consider definitions due to intensional semantics here. His intensional meaning of subclass is in terms of subclass instances being valid implementations of the superclass, however, the definition is different from a refinement relation (as one would expect). To exhibit subtyping there must exist a retrieve relation Abs between the superclass and subclass such that the following are true.

S1 $\forall\, Superstate;\ Substate \bullet \mathrm{pre}\, Superop \wedge Abs \Rightarrow \mathrm{pre}\, Subop$

S2 $\forall\, Superstate;\ Substate;\ Substate' \bullet \mathrm{pre}\, Superop \wedge Abs \wedge Subop \Rightarrow (\exists\, Superstate' \bullet Abs' \wedge Superop)$

S3 $\forall\, Substate \bullet (\exists\, Superstate \bullet Abs)$

Only the third rule differs from the rules for refinement, see [23] for justification of this. Hall also compares his definition with those of Cusack [14], Lano & Haughton [34] and Liskov & Wing [38]. We are interested here in the relation between unification and the individual viewpoints. In these circumstances the retrieve relations will be partial functions (and only total if one viewpoint is degenerate). In this case Hall's subtyping imply subtyping in the sense of Cusack, Lano & Haughton and Liskov & Wing (ignoring the history predicates of the latter two). In particular, the rules S1-3 suffice for subtyping in both Hall and ZEST, and we will thus work with this definition.

It is easy to construct examples to show that the unification of two viewpoints is not in general a subtype of each viewpoint. However, this is unsurprising because one viewpoint is only a partial description of an objects behaviour. Instead the natural result to seek is the following:

Theorem 2 *Let P_i, O_i be objects in viewpoint i. Let P_i be a subtype of O_i. Then $\mathcal{U}(P_1, P_2)$ is a subtype of $\mathcal{U}(O_1, O_2)$, where $\mathcal{U}$ is the unification operator between viewpoints.*

Proof

The full proof involves construction of appropriate retrieve relations between $\mathcal{U}(P_1, P_2)$ and $\mathcal{U}(O_1, O_2)$ in a manner similar to the proof that unification is the least refinement, see [16]. The outline of the proof is as follows:

The subtyping rules S1 and S2 between $\mathcal{U}(P_1, P_2)$ and $\mathcal{U}(O_1, O_2)$ are satisfied because unification is the least refinement.

For S3, note that every state in the $\mathcal{U}(O_1, O_2)$ unification appears in either O_1 or O_2 or both. Thus every state in $\mathcal{U}(P_1, P_2)$ is related to some state in $\mathcal{U}(O_1, O_2)$ via the retrieve relation defined for the least refinement. □

It is straightforward to construct examples to show the converse is not true, that is $\mathcal{U}(P_1, P_2)$ being a subtype of $\mathcal{U}(O_1, O_2)$ does not imply that P_i is a subtype of O_i.

The theorem then provides a sound footing for the use of object oriented techniques in viewpoint descriptions. The relationship between unification and multiple inheritance is clearly of importance (especially w.r.t method consistency), and is currently under investigation.

Chapter 6

Conclusion

In conclusion of this deliverable we summarise the results so far and describe the key open problems that remain.

6.1 Summary of results

We have reported on three major areas of investigation:

1. Defining consistency;
2. Consistency checking in LOTOS; and
3. Consistency checking in Z.

6.1.1 Defining consistency

The RM-ODP contains three different definitions of consistency. These definitions have been formalised and related to one another (see chapter 3). The different definitions seem incomparable unless they had been instantiated with a particular FDT. To resolve this deficiency a more general definition of consistency was formulated that can incorporate all three RM-ODP definitions.

In general terms, a number of viewpoint specifications are consistent if and only if a new specification can be found that is a development of all viewpoint specifications. In addition, the new specification is required to be internally valid (i.e. can be implemented). We distinguish between balanced consistency, where one development relation is used in all viewpoints, and unbalanced consistency, where different development relations can be used in different viewpoints.

If a number of viewpoint specifications are consistent with respect to certain development relations, then a common development can be found, which we call unification. Conversely, the existence of an internally valid unification implies consistency of the viewpoint specifications.

Although the definition of consistency is general enough to cope with inter language consistency checking, we have initially considered intra language consistency checking only.

6.1.2 Consistency checking in LOTOS

In chapter 4, we describe techniques to check for binary, balanced consistency of LOTOS specifications with respect to four different development relations: trace preorder, reduction, extension and testing equivalence. Techniques to construct unifications of two LOTOS specifications with respect to these development relations are considered.

In LOTOS it is possible to find necessary and sufficient conditions to show that two specifications are consistent. Once consistency has been established, it is possible to construct a unification. However, the generated unification is usually not the least common development.

6.1.3 Consistency checking in Z

In chapter 5, techniques are given to unify Z viewpoint specifications with respect to the Z refinement relation. It also describes how the internal validity of the derived unification can be verified. Since the derived unification is always the least unification, internal validity of the unification implies the consistency of the original specifications.

The use of object oriented specification techniques was also considered. Initial evidence indicates such techniques can reduce the complexity of the consistency checking process.

6.2 Open problems

This deliverable has outlined a number of issues relating to consistency checking mechanisms for ODP. There clearly remains much work to be done, we discuss briefly here some of the future directions for research in this area.

6.2.1 Inter language consistency checking

It is commonly recognised that different FDTs will be applicable to different viewpoints. In this respect, to be of practical use, inter language consistency checking mechanisms have to be built upon intra language mechanisms. An approach could be envisaged where to check the consistency of two different FDT viewpoint specifications, one specification is translated into the language of the other, after which an intra language mechanism is applied. If this is possible then checking across language boundaries becomes feasible.

Translation is discussed briefly below, languages to be considered include LOTOS, Z and IDLs. The translation between IDLs and Z clearly has practical significance, and we will investigate this as resources allow. The translation between LOTOS and Z would be a fundamentally important result, given the different semantic bases for the two languages, and we are considering to what extent this can be achieved.

6.2.2 Translation

There has been some success in relating formal languages that have similar underlying semantics, e.g. [43, 5]. However, the common semantics used in these approaches is typically very ugly. ODP consistency checking requires translation across FDT families. Some directions that could be pursued to make such translations possible are discussed below.

Syntactic translation Translation based upon a direct relation of syntactic terms in one FDT to terms in another FDT is one possible approach. However, it is difficult to envisage how such an approach could offer a general solution. In particular, a lot of semantic meaning will certainly be lost in such a crude translation of FDTs. Partial syntactic translations may, however, be feasible.

Common underlying semantics Another, more promising, approach is to define a common underlying semantic model for the required FDTs. Specifications could then be translated into the common semantic domain, in which a consistency check can be performed. Such translation could either use the semantics of one of the FDTs as the intermediate semantics or use a third semantics. The former of these is not fully general; for example, Z and LOTOS are so fundamentally different that relating one to the others semantic model is very difficult to envisage. Relating FDTs using a third intermediate form is a more likely approach.

A sufficiently general semantic model has been developed in [57]. One-sorted first-order predicate logic is proposed to capture the semantics of specifications in different formalisms. The proposed semantic model is very general, but it does not necessarily have the same properties as the standard semantics of the FDTs. Also, there are certain features of specification languages which cannot be captured in first-order logic. Nevertheless, the proposed method presents a promising step towards a general technique for consistency checking.

There have been several other attempts to relate different formal languages:

- A link between model based action systems (and thereby Z) and CSP has been made by showing that refinements (forwards and backwards simulation) in an action system are sound and jointly complete with respect to the notion of refinement in CSP [55].
- The requirement for highly expressive intermediate semantics suggests that logical notations may be appropriate. [20] and [7] consider logical characterisations of LOTOS in temporal logic. However, relating temporal logic to the Z first order logic remains an open issue. Categorical approaches and the theory of institutions offer a possible solution [7].
- A final alternative which has the benefit of being ODP specific is suggested by the work of [13]. This work offers a direct denotational semantics for the computational viewpoint language. This semantics could, theoretically, be used to relate different FDT interpretations of the computational viewpoint language. Clearly, this work does not give a complete solution to consistency as the semantics are restricted to a single viewpoint. However, it may be possible to extrapolate this approach to a general solution.

It is clear, though, that a usable translation mechanism is likely to represent a pragmatic, compromise solution. In particular, complete preservation of semantic meaning during translation will not be possible. In addition, different viewpoints describe different sets of features and thus may not be directly translatable between each other.

6.2.3 ODP specific concepts

The work presented in this deliverable has concentrated on a general framework for consistency checking mechanisms and, in particular, we have not considered specific ODP viewpoints. Consideration of the viewpoints will include discussion of the role of the ODP architectural semantics. Specifically, part 4 should provide a basis for relating FDTs. ODP concepts, in particular viewpoint languages, are defined in different FDTs in the architectural semantics. Thus, when relating complete viewpoint specifications in different FDTs these definitions can be used as components of a consistency check.

However, it is important to note that the architectural semantics will only provide a framework for consistency checking. Actual viewpoint language specifications will extend the ODP architectural semantics, which are non-prescriptive by nature, with FDT specific behaviour. There is then a need to combine the framework provided by the architectural semantics with actual consistency checking relationships arising from FDTs. Such cross viewpoint consistency checks will clearly involve correspondence rules, and further work is required on identifying appropriate correspondences between the viewpoints in order to provide formal support for cross viewpoint mappings.

6.2.4 Tool Development

Work on tool development based upon consistency checking mechanisms is clearly important if this work is to move into application areas. We envisage roles for unification tools (in particular for Z), coupled to semi-automatic consistency checkers which will aim to provide support for consistency checks. The extent of such tool development will be considered in the light of available resources.

6.2.5 Object orientation

The ODP viewpoint languages are object based. Current consistency checking and unification algorithms do not take the object oriented nature of the specifications into consideration. Further research is necessary to assess in which way object orientation will effect the consistency checking process. Some experiments with object oriented Z specifications suggest that the encapsulation property of objects will simplify the consistency checking mechanisms.

6.3 Future plans

In this section, we list the activities planned for the future. In the short term, we plan to consider the following tasks:

- Identification and characterisation of ODP correspondence rules;
- Building a unification and consistency checking tool for Z;
- Extending the LOTOS consistency checking techniques to unbalanced and global consistency;
- Starting work on a LOTOS tool for unification and consistency checking;
- Investigation of inter language consistency checking between LOTOS and Z specifications;
- Identification of case studies for intra language consistency checking in LOTOS and Z; and
- Provision of input to RM-ODP Part 1 on consistency checking.

For the long term we envisage the following activities:

- Completion of unification and consistency checking tools for both Z and LOTOS;
- Building a tool for consistency checking and/or translation between Z and LOTOS;
- Performing several case studies on both intra language and inter language consistency checking; and
- Dissemination of final results to the ODP community.

Bibliography

[1] M. Ainsworth, A. H. Cruickshank, L. J. Groves, and P. J. L. Wallis. Formal specification via viewpoints. In J Hosking, editor, *Proc. 13th New Zealand Computer Conference*, pages 218–237, Auckland, New Zealand, 18th–20th August 1993. New Zealand Computer Society.

[2] M. Ainsworth, A. H. Cruickshank, L. J. Groves, and P. J. L. Wallis. Viewpoint specification and Z. *Information and Software Technology*, 36(1):43–51, February 1994.

[3] A. J. Alencar and J. A. Goguen. OOZE: An object oriented Z environment. In P. America, editor, *ECOOP '91 - Object-Oriented Programming*, LNCS 512, pages 180–199. Springer-Verlag, 1991.

[4] R. D. Arthan. On free type definitions in Z. In J. E. Nicholls, editor, *Sixth Annual Z User Workshop*, pages 40–58, York, December 1991. Springer-Verlag.

[5] D. Bert, M. Bidoit, C. Choppy, R. Echahed, J.-M. Hufflen, J.-P. Jacquot, M. Lemoine, N. Lévy, J.-C. Reynaud, C. Roques, F. Voisin, J.-P. Finance, and M.-C. Gaudel. Opération SALSA: Structure d'Accueil pour Spécifications Algébriques. Rapport final, PRC Programmation et Outils pour l'intelligence Artificielle, 1993.

[6] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, 1988.

[7] H. Bowman and J. Derrick. Formalizing conformance in ODP. In R. Wieringa and R. Feenstra, editors, *Working papers of IS-CORE'94, International Workshop on Information Systems - Correctness and Reusability*, pages 357–370, September 1994.

[8] H. Bowman and J. Derrick. Modelling distributed systems using Z. In K. M. George, editor, *ACM Symposium on Applied Computing*, pages 147–151, Nashville, February 1995. ACM Press.

[9] H. Bowman, J. Derrick, and M. Steen. Some results on cross viewpoint consistency checking. In *IFIP International Conference on Open Distributed Processing*. Chapman Hall, 1995.

[10] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification, VIII*, pages 63–74, Atlantic City, USA, June 1988. North-Holland.

[11] E. Brinksma and G. Scollo. Formal notions of implementation and conformance in LOTOS. Technical Report INF-86-13, Dept of Informatics, Twente University of Technology, 1986.

[12] E. Brinksma, G. Scollo, and C. Steenbergen. Process specification, their implementation and their tests. In B. Sarikaya and G. V. Bochmann, editors, *Protocol Specification, Testing and Verification, VI*, pages 349–360, Montreal, Canada, June 1986. North-Holland.

[13] AFNOR cont. *A direct computational language semantics for Part 4 of the RM-ODP*. ISO/IEC JTC1/SC21/WG7 approved AFNOR contribution, July 1994.

[14] E. Cusack. Inheritance in object oriented Z. In P. America, editor, *ECOOP '91 - Object-Oriented Programming*, LNCS 512, pages 167–179. Springer-Verlag, 1991.

[15] E. Cusack and G. H. B. Rafsanjani. ZEST. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 113–126. Springer-Verlag, 1992.

[16] J. Derrick, H. Bowman, and M. Steen. Maintaining cross viewpoint consistency using Z. In *IFIP International Conference on Open Distributed Processing*. Chapman Hall, 1995.

[17] J. Derrick, H. Bowman, and M. Steen. Viewpoints and Objects. In *Ninth Annual Z User Workshop*, Limerick, September 1995. Springer-Verlag. To appear.

[18] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*. Academic Press, 1968.

[19] R. Duke, P. King, G. A. Rose, and G. Smith. The Object-Z specification language version 1. Technical Report 91-1, Software Verification Research Centre, Department of Computer Science, University of Queensland, May 1991.

[20] A. Fantechi, S. Gnesi, and G. Ristori. Compositional logic semantics and LOTOS. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Protocol Specification, Testing and Verification, X*, Ottawa, Canada, June 1990. North-Holland.

[21] J. F. Groote. Transition system specifications with negative premises. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR 90*, LNCS 458, pages 332–341, Amsterdam, 1990. Springer-Verlag.

[22] A. J. Hall. Using Z as a specification calculus for object-oriented systems. In D. Bjorner, C. A. R. Hoare, and H. Langmaack, editors, *VDM '90 VDM and Z - Formal Methods in Software Development*, LNCS 428, pages 290–318, Kiel, FRG, April 1990. Springer-Verlag.

[23] J. A. Hall. Specifying and interpreting class hierarchies in Z. In J. P. Bowen and J. A. Hall, editors, *Eighth Annual Z User Workshop*, pages 120–138, Cambridge, July 1994. Springer-Verlag.

[24] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[25] H. Ichikawa, K. Yamanaka, and J. Kato. Incremental Specification in LOTOS. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Protocol Specification, Testing and Verification X*, pages 183–196, Ottawa, Canada, 1990.

[26] ISO. Information Processing Systems – Open Systems Interconnection – Basic Reference Model, 1984. IS 7498.

[27] ISO. Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour, 1989. IS 8807.

[28] ISO. LOTOS description of the Session Protocol, 1989. ISO/IEC TR9572.

[29] ISO. LOTOS description of the Session Service, 1989. ISO/IEC TR9571.

[30] ITU Recommendation X.901-904 — ISO/IEC 10746 1-4. *Open Distributed Processing - Reference Model - Parts 1-4*, July 1995.

[31] F. Khendek, G. v. Bochmann, and R. Gotzhein. Multiple inheritance in the form of reduction. Technical report, Département d'informatique et de recherche opérationnelle, Université de Montréal, Montreal, Canada, 1992.

[32] F. Khendek and G. von Bochmann. Merging specification behaviours. Technical Report 856, Département d'informatique et de recherche opérationnelle, Université de Montréal, 1993.

[33] S. King. Z and the refinement calculus. In D. Bjorner, C. A. R. Hoare, and H. Langmaack, editors, *VDM '90 VDM and Z - Formal Methods in Software Development*, LNCS 428, pages 164–188, Kiel, FRG, April 1990. Springer-Verlag.

[34] K. Lano and H. Haughton. Reuse and adaption of Z specifications. In J. P. Bowen and J. E. Nicholls, editors, *Seventh Annual Z User Workshop*, pages 62–90, London, December 1992. Springer-Verlag.

[35] G. Leduc. *On the Role of Implementation Relations in the Design of Distributed Systems using LOTOS*. PhD thesis, University of Liège, Liège, Belgium, June 1991.

[36] G. Leduc. A framework based on implementation relations for implementing LOTOS specifications. *Computer Networks and ISDN Systems*, 25:23–41, 1992.

[37] P. F. Linington. Introduction to the Open Distributed Processing Basic Reference Model. In J. de Meer, V. Heymer, and R. Roth, editors, *IFIP TC6 International Workshop on Open Distributed Processing*, pages 3–13, Berlin, Germany, September 1991. North-Holland.

[38] B. Liskov and J. M. Wing. A new definition of the subtype relation. In O. M. Nierstrasz, editor, *ECOOP '93 - Object-Oriented Programming*, LNCS 707, pages 118–141. Springer-Verlag, 1993.

[39] S. L. Meira and A. L. C. Cavalcanti. Modular object oriented Z specifications. In J. E. Nicholls, editor, *Fifth Annual Z User Workshop*, pages 173–192, Oxford, December 1990. Springer-Verlag.

[40] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[41] B. Potter, J. Sinclair, and D. Till. *An introduction to formal specification and Z*. Prentice Hall, 1991.

[42] B. Ratcliff. *Introducing specification using Z*. McGraw-Hill, 1994.

[43] R. Reed, W. Bouma, J. D. Evans, M. Dauphin, and M. Michel. *Specification and Programming Environment for Communication Software*. North Holland, 1993.

[44] S. Rudkin. Modelling information objects in Z. In J. de Meer, V. Heymer, and R. Roth, editors, *IFIP TC6 International Workshop on Open Distributed Processing*, pages 267–280, Berlin, Germany, September 1991. North-Holland.

[45] M. Saaltink. Z and Eves. In J. E. Nicholls, editor, *Sixth Annual Z User Workshop*, pages 223–243, York, December 1991. Springer-Verlag.

[46] S. A. Schuman, D. H. Pitt, and P. J. Byers. Object-oriented process specification. In C. Rattray, editor, *Specification and Verification of Concurrent Systems*, Workshops in Computing, pages 21–70. Springer-Verlag, 1990.

[47] R. Sinnott. *An Initial Architectural Semantics in Z of the Information Viewpoint Language of Part 3 of the ODP-RM*. ISO/IEC SC21/WG7 N915, July 1994. BSI Input document to the ODP Plenary meeting in Southampton.

[48] A. Smith. On recursive free types in Z. In J. E. Nicholls, editor, *Sixth Annual Z User Workshop*, pages 3–39, York, December 1991. Springer-Verlag.

[49] G. Smith. An object-oriented development framework for Z. In J. P. Bowen and J. A. Hall, editors, *Eighth Annual Z User Workshop*, pages 89–107, Cambridge, July 1994. Springer-Verlag.

[50] J. M. Spivey. *The Z notation: A reference manual.* Prentice Hall, 1989.

[51] S. Stepney, R. Barden, and D. Cooper, editors. *Object Orientation in Z.* Workshops in Computing. Springer-Verlag, 1992.

[52] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. On the use of specification styles in the design of distributed systems. *Theoretical Computer Science*, 89(1):179–206, October 1991.

[53] C. Wezeman and A. J. Judge. Z for managed objects. In J. P. Bowen and J. A. Hall, editors, *Eighth Annual Z User Workshop*, pages 108–119, Cambridge, July 1994. Springer-Verlag.

[54] P. J. Whysall and J. A. McDermid. An approach to object oriented specification using Z. In J. E. Nicholls, editor, *Fifth Annual Z User Workshop*, pages 193–215, Oxford, December 1990. Springer-Verlag.

[55] J. C. P. Woodcock and C. C. Morgan. Refinement of state-based concurrent systems. In D. Bjorner, C. A. R. Hoare, and H. Langmaack, editors, *VDM '90 VDM and Z - Formal Methods in Software Development*, LNCS 428, pages 340–351, Kiel, FRG, April 1990. Springer-Verlag.

[56] P. Zave and M. Jackson. Techniques for partial specification and specification of switching systems. In J. E. Nicholls, editor, *Sixth Annual Z User Workshop*, pages 205–219, York, December 1991. Springer-Verlag.

[57] P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411, October 1993.