Towards a Formal Model of Consistency in ODP

Howard Bowman and John Derrick.

Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
(Phone: + 44 227 764000, Fax: +44 227 762811, Email: hb5@ukc.ac.uk and jd1@ukc.ac.uk.)

March 17, 1994

Abstract

The ODP (Open Distributed Processing) development model is a natural progression from OSI. Multiple viewpoints are used to specify complex ODP systems. The conformance assessment methodology for ODP defines the relationships between specifications and implementations that must be proved in order that conformance to ODP systems can be asserted. These relationships include transformation, refinement, conformance and consistency. This paper develops a formal interpretation of these concepts. In particular the paper defines a framework for checking the *consistency* of different specifications of the same ODP system. This framework is essential if FDT's are to be successfully employed in the development of ODP systems. In the second part of the paper, we present examples of consistency checking in both LOTOS and RAISE.

Keywords: Open Distributed Processing, Conformance Assessment, Formal Methods, Specification, Implementation, Consistency.

1 Introduction

The ODP (Open Distributed Processing) standardisation initiative is a natural progression from OSI, broadening the target of standardization from the point of interconnection to the end-to-end system behaviour. The ODP framework, development of which began in earnest in 1987, with the initiation of work on the ODP Reference Model (ODP-RM), can be said to be reaching a level of relative maturity [?]. In particular, Parts 2 and 3 of the reference model are currently Committee Drafts being balloted for progression to Draft International Standard. A goal of the ODP architecture as described in [?] is to enable the construction of potentially global computing systems that may both cross many administrative boundaries and utilize a variety of systems and communications technology, including that which currently exists and that which may be provided for the purpose in the future.

Those working on the ODP reference model have avoided the mistake made when defining the OSI reference model of leaving consideration of conformance until later. Instead the meaning of conformance to ODP has been built into the ODP reference model from the start. It is therefore important to develop a conformance assessment *methodology*, and define the place within that of active and passive testing, and consistency checking. Furthermore, a conformance assessment methodology sufficient to meet the needs of ODP should be general and powerful enough to be used for conformance assessment of OSE profiles and of the full complexity of OSI network management.

One of the cornerstones of the ODP framework is a model of multiple viewpoints which enables different participants to observe a system from a suitable perspective and at a suitable level of abstraction [?, ?]. There are five separate viewpoints presented by the ODP model: Enterprise, Information, Computational, Engineering and Technology. Requirements of, and statements about, an ODP system can be made from any of these viewpoints and, therefore, conformance assessment is separately relevant to each viewpoint. This means that a methodology to address conformance assessment in ODP will have a very broad range and consequently will be very widely applicable.

However, while it has been accepted that the viewpoint model greatly simplifies the development of system specifications and offers a powerful mechanism for handling diversity within ODP, the practicalities of how to make the approach work are only beginning to be explored. In particular, one of the consequences of adopting a multiple viewpoint approach to development is that descriptions of the same or related objects can appear in different viewpoints and must co-exist. Furthermore, different notations are likely to be used in different viewpoints. *Consistency* of specifications across viewpoints thus becomes a central issue, both in the development and the conformance assessment process. However, the actual mechanism by which consistency can be checked and maintained is only just being addressed [?, ?, ?]. Many of the ODP consistency concepts have not been formalized. In particular, the mathematical properties of specifications co-existing in different viewpoints and potentially in different languages need to be clarified.

Work underway in Peter Linington's Open Distributed Processing research group at the University of Kent at Canterbury aims to respond to these issues. The initial step in this research has been the development of a formal model of consistency within ODP. We strongly believe that a formal approach to the problem of consistency checking should be employed. In particular, the ability to reason rigorously about the specifications under consideration will greatly aid the development of proofs of mutual consistency. This is especially important given the increasing role that formal methods are playing within ODP. Part 4 of the ODP-RM outines requirements for applying formal description techniques in the specification of ODP systems. Languages under investigation include LOTOS, Estelle, SDL, Z, Object-Z and RAISE. We illustrate our formal model of consistency checking using two of these candidate languages: RAISE and LOTOS.

The concepts and terminology surrounding the consistency checking issue are not firmly fixed. In fact, concepts are frequently misinterpreted within this domain. Thus, the role of this paper can be seen to be two fold. Firstly, it will clarify a coherent set of terminology surrounding the consistency checking issue. Implicit in this will be the highlighting of the exact position that consistency fits into the present ODP model. Then secondly, we will focus directly on the consistency issue and present a first step towards the definition of a formal theory of consistency between specifications.

Thus, the paper is structured as a description and definition of general ODP concepts and principles, leading up to a precise definition of consistency and related concepts. The paper first discusses the general ODP model of product development (in Section 2) and then in Section 3 the theory of conformance assessment suggested by the ODP framework is developed. A formal interpretation of this conformance assessment process is developed in Section 4. A discussion of the framework is given in Section 5, followed by some simple examples of consistency relationships in formal description techniques in Section 6. Finally, Section 7 contains some concluding remarks.

2 Preliminaries: The ODP Development Process

Work on conformance assessment for Open Distributed Systems (ODP) has identified a number of issues in the development process and the conformance assessment process by which confidence in a product or an instance of a product can be gained [?]. The purpose of this paper is to give formal interpretations of some of these notions.

Product development extends from the initial requirements to the final product that fulfills those requirements. Initially the requirements are clarified to produce a specification.

The development process then focuses on this specification and is responsible for the generation of a number of subsequent specifications using one of a number of types of step ("transformations") including *translation* and *refinement*.

Specifications are expressed in some natural or formal language. Translation produces a specification with the same meaning in a (possibly) different language. Refinement, on the other hand, produces a specification with new details that serve to define the product more closely (note that refinement can occur across language boundaries and thus, there can be an element of translation in the process).

Once these steps have resulted in sufficient precision an example of the product is realized (a "real implementation"), based on the final ("implementation") specification.

The development process and the transformations implicit in it are shown in Figure 1. The product then enters use, it should then meet the needs expressed in the document defining its requirements. Conformance assessment is the process used in order to give confidence that a product does meet its requirements.

3 Conformance assessment within ODP

The conformance assessment process determines whether a product satisfies (ie conforms to) a given specification. In this way we can obtain a measure of confidence that a product satisfies the requirements that the specification has been derived from.

The relationships between specifications and real implementations that are relevant to conformance assessment were identified in [?]. These are divided into two groups:

(i) relationships between specifications and real implementations (conformance); and,

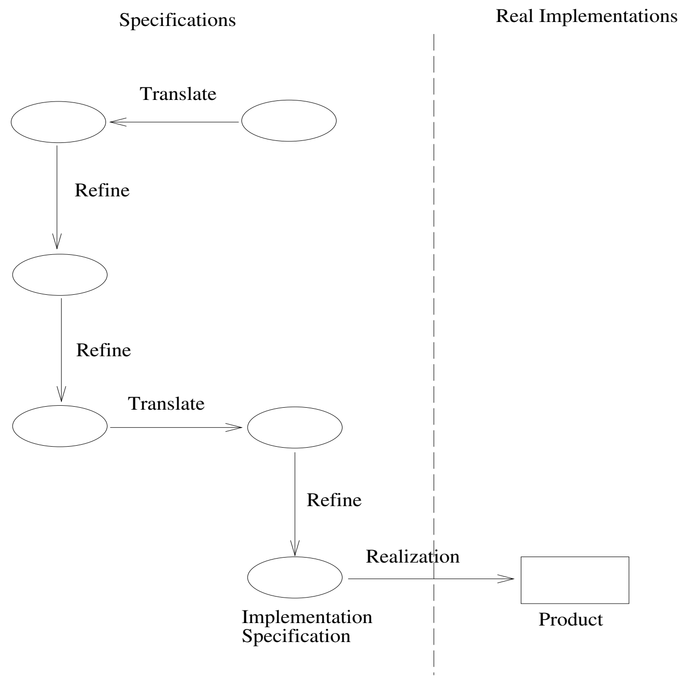


Figure 1: Example Product Development Relationships

(ii) relationships between specifications alone (compliance, refinement, consistency and internal validity).

We outline their informal definition as given in [?]:

Conformance is a relation between a specification and a real implementation, such as an example of a product. It holds when specific properties in the specification are met by the implementation. Conformance assessment is the process through which this relation is determined.

Compliance is a relation between two specifications, A and B. B complies wth A when specification A makes requirements which are all fulfilled by specification B.

One specification is a *refinement* of another when all the products that could conform to the refinement could also conform to the specification from which it was refined. Compliance and refinement are in fact very closely related concepts and in terms of this paper we will not distinguish them. However, within the full ODP model refinement is associated with the design process, while compliance is a more general term for relating specifications in the abstract.

ODP systems are specified from different viewpoints. Consistency between the specifications is needed as it is important that the properties of one specification do not contradict those of another. Consistency is a relation between two specifications that holds when it is possible for at least one example of a product to exist that can conform to both of the specifications.

A specification is *internally valid* when there are no conflicts between its properties and those implicit properties required of the specification; and when there is at least one example product that could conform to it. Properties, such as freedom from deadlock, which are always required to hold in specifications are called implicit properties. However, the full value of distinguishing such properties is still not clear. A further discussion of this issue is made in Section 4.4.

Conformance assessment is the determination of these relationships either by testing (conformance) or by specification checking (compliance, refinement, consistency and internal validity).

3.1 Conformance assessment: testing

A specification defines aspects of a real implementation's behaviour, thus an implementation can be tested to confirm the presence of the required behaviour.

Testing is the process of providing the stimuli to the implementation and confirming the expected observable outcomes for each appropriate test. In ODP specific points of conformance are identified in the specification [?]. At these conformance points the expected observable outcome following the application of a known stimuli can be prescribed.

3.2 Conformance assessment: specification checking

The need for specifications of ODP systems by a number of viewpoints is well documented [?, ?]. The presence of a number of viewpoint specifications complicates the conformance assessment process.

The study [?] identified a number of mappings between specifications that occur in the development and the conformance assessment process. Transformations between specifications during the development process included translation and refinement, with specifications being mapped onto real implementations by realization or animation. Checks between specifications during the conformance assessment process included refinement checking, while checks between a real implementation and a specification included only testing.

In addition there are transformations between specifications during the development process (unification) and an additional check between specifications during the conformance assessment process (consistency). The following table summarizes the different mappings that result.

Mapping specification to	Development Process	Conformance Process
Specification	translation, refinement, unification	internal validity checks, refinement checks, consistency checks
Real Implementation	animation	testing

The specification of the same ODP system from different viewpoints means that there must be some way to combine different viewpoint specifications at some stage during the development process (assuming the final implementation is to have a single specification). This combination is referred to as *unification*.

Simply combining specifications is not possible unless common notions in the different specifications are first *normalized* by identifying them and associating them with the same term. An example of defining common notions would be highlighting the correspondence between variables in the two specifications. The appropriate correspondences must normally be supplied explicitly by experts who understand the intent behind the different specifications.

Once such normalization has taken place it is possible that the resulting specification will have no possible implementation. This is because the viewpoint specifications have placed contradictory

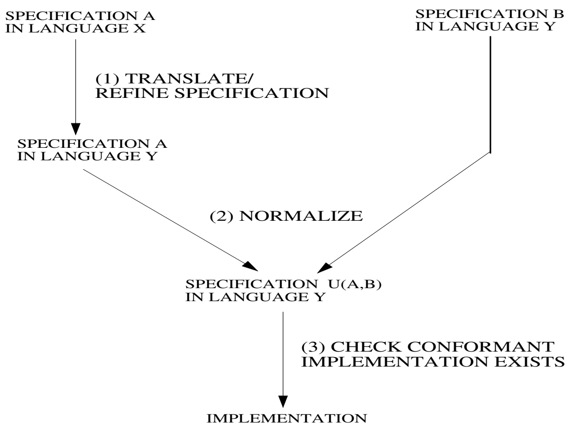


Figure 2: Checking Consistency

requirements on an implementation. The development procedure must ensure that the specifications developed are consistent.

Specifications from different viewpoints may be checked for consistency by applying transformations (eg translation and refinement) to each of them as necessary into a common language, perform a normalization of their universe of discourse and then show that at least one implementation of the unified specification is possible. This process, in the case where the common language chosen is one of the original specification languages is shown, in Figure 2. In this diagram $\mathcal{U}(A, B)$ signifies the unification of specifications A and B. In addition, transformation (3) is the actual consistency checking step.

A consistency check is defined in [?] by fixing the language of one of the specifications and transforming the other specifications to that language. It then requires the resulting specifications to be behaviourally compatible. That is, an ODP object displaying behaviour conforming to one specification should be able to be replaced by an ODP object with behaviour conforming to the other. Thus an object that conforms to both specifications must be found.

Unification (transformation and normalization) of specifications is useful, therefore, both in the assessment of consistency and in the production of a single implementation from many specifications.

4 Formal Interpretation

In this section a general mathematical framework is developed for reasoning about the conformance assessment process. To do so we consider relations which are supposed to express formally the notions of conformance, refinement and consistency.

ODP systems are specified from a number of different viewpoints, each viewpoint specification is specified in a language. Let L_i ($i \in \mathbb{N}$) be a collection of languages, and let $Spec_i = Spec(L_i)$ be the set of specifications written in that language. Define

$$SPEC = \bigcup_i Spec_i$$

SPEC includes the text of all possible specifications of ODP systems (including inconsistent ones).

In practice, however, an ODP specification would consist of specifications from different viewpoints and a number of additional mappings showing the relationships between constructs used in the different viewpoint languages. However for the purposes of the initial abstract model developed here, it suffices to consider the flat domain SPEC.

4.1 Conformance

Conformance is a relationship between specifications and potential real implementations. Let Imp denote the collection of real implementations. Note that Imp is the class of products or instances of a product, and thus is a very different type of object to $SPEC$. Most theories of conformance do not make this distinction, but to reason correctly about the ODP development and assessment process we believe it is important to do so. Beyond noting that $SPEC \cap Imp = \emptyset$, we can say little about the structure of Imp.

The formal conformance relation, denoted *conf* in the sequel, is intended to express the notion of an implementation conforming to a specification. Thus

$$conf \subseteq SPEC \times Imp$$

Since the link between a product and its model remains informal by nature, it can never be proved that a specification and implementation are related by *conf*. The best we can do is to assert that $(S, I) \in conf$ on the basis of conformance testing carried out. If however conformance testing indicates that the behaviour of the implementation I contradicts that required by S, then we can say that $(S, I) \notin conf$.

Previous notions of conformance have considered the formal conformance relation to be a subset of $SPEC \times SPEC$, [?, ?, ?, ?]. The theory of equivalence and refinement remains much the same what ever approach is taken, indeed Section 4.2 contains analogous definitions and results to those defined by the non-transitive conformance relation $\underline{\text{imp}} \subseteq SPEC \times SPEC$ in [?].

4.2 Equivalence and Refinement

The transformations taking place during the ODP development process include translation and refinement. A specification S_1 is a translation of a specification S_2 if they are equivalent, denoted $S_1 \equiv S_2$.

Definition 1 $S_1 \equiv S_2$ *iff* $\{I : S_1\ conf\ I\} = \{I : S_2\ conf\ I\}$

Intuitively, two specifications are equivalent iff they determine exactly the same set of valid implementations, as defined by *conf* (this is depicted in Figure 3). Equivalence should certainly preserve semantics. That is, if two specifications have the same underlying semantic interpretation, denoted $\mathcal{M}[S_1] = \mathcal{M}[S_2]$, then they should be regarded as equivalent. This implies that if $S_1\ conf\ I$ and $\mathcal{M}[S_1] = \mathcal{M}[S_2]$ then $S_2\ conf\ I$.

It is obvious that equivalence is reflexive, symmetric and transitive. Therefore $\equiv$ is an equivalence relation whatever *conf* is.

Refinement is a development activity which restricts the set of valid implementations of a specification. The refinement relation is denoted by $\sqsubseteq$.

Definition 2 $S_1 \sqsubseteq S_2$ *iff* $\{I : S_2\ conf\ I\} \subseteq \{I : S_1\ conf\ I\}$

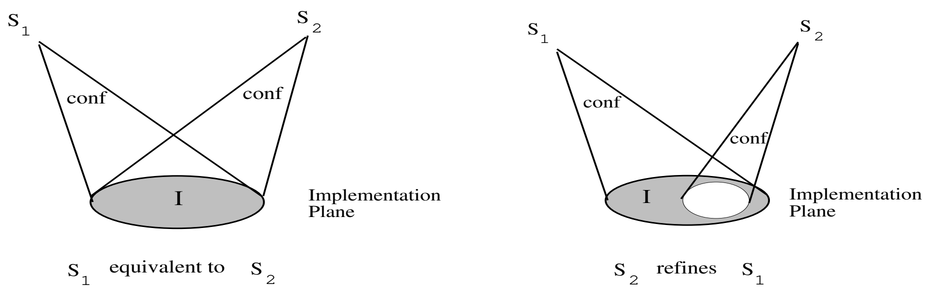


Figure 3: Equivalence and Refinement

This definition ensures that any implementation that conforms to S_2 will also conform to specification S_1, see Figure 3.

Proposition 1
(i) $\sqsubseteq$ is a pre-order (ie reflexive and transitive)
(ii) $S_1 \equiv S_2$ iff $S_1 \sqsubseteq S_2$ and $S_2 \sqsubseteq S_1$ (ie, $\sqsubseteq$ is a partial order with respect to equivalence)

The two relations $\sqsubseteq$ and *conf* are further related, as shown by the next proposition (where $\circ$ denotes relation composition).

Proposition 2 $(\sqsubseteq \circ\, conf) = conf$

Proof
The result follows from the following two arguments:
(i) Let $S_1(\sqsubseteq \circ conf)I$. Then there exists S_2 with $S_1 \sqsubseteq S_2$ and $S_2 conf I$. By the definition of *conf*, $S_1 conf I$.
(ii) $Id \subseteq \sqsubseteq$ implies that $conf \subseteq (\sqsubseteq \circ conf)$. □

We now consider some properties of the relations stronger than refinement. These will characterize refinement.

Proposition 3 *For all R, we have $R \subseteq \sqsubseteq$ iff $(R \circ\ conf) \subseteq\ conf$*

Proof
$(\leftarrow)$ This follows from the facts that: $R \subseteq \sqsubseteq$ implies that $R \circ\ conf \subseteq (\sqsubseteq \circ conf)$, and $\sqsubseteq \circ\ conf = conf$.

$(\rightarrow)$ We argue by contradiction. Consider any relation R such that $R \subseteq \sqsubseteq$ doesn't hold. We show that $R \circ\ conf \subseteq\ conf$ doesn't hold.

By hypothesis, there exist P and Q with PRQ and $\neg(P \sqsubseteq Q)$. Since $P \sqsubseteq\ Q$ doesn't hold, there exists an implementation I with $QconfI$ and $\neg(PconfI)$. Hence there exist I, P, Q, such that $(PRQ) \wedge (QconfI) \wedge \neg(PconfI)$. Hence $R \circ\ conf \subseteq\ conf$ doesn't hold. □

By restricting to reflexive relations stronger than refinement, the next proposition and corollary easily follow:

Proposition 4 *For all R, we have $Id \subseteq R$ implies that $(R \subseteq \sqsubseteq)$ iff $(R \circ\ conf = conf)$*

Corollary 1 *Refinement is the least relation R such that $R \circ conf = conf$.*

4.3 Normalization and Unification

Normalization as a process takes two specifications defined over the same language, and produces a normalized version which is a combination of the two specifications, again with respect to (wrt) to the same language. The first attempt would be to regard normalization defined over the language L_i as a function $N : Spec \times Spec \longrightarrow Spec$ where $Spec = Spec_i$.

However it is possible to normalize two specifications in many different ways, all that we should be concerned about is that all normalizations should be equal wrt to equivalence, $\equiv$. Hence normalization defined over the language L_i is a function $N : Spec \times Spec \longrightarrow Spec^\bullet$ where $Spec^\bullet = Spec/\equiv$ in the usual notation is the set of equivalence classes of $Spec$ wrt $\equiv$. The equivalence class of a specification S wrt $\equiv$ is denoted [S], however as is common convention we will denote classes by representatives without loss of generality when no ambiguity arises.

Properties 1 *A normalization function must satisfy the following properties:*

(i) $N(T_1, T_2) = N(T_2, T_1)$ - *commutativity*
(ii) $N(T_1, N(T_2, T_3)) = N(N(T_1, T_2), T_3)$ - *associativity*
(iii) $T_1, T_2 \sqsubseteq N(T_1, T_2)$ - *common refinement*
(iv) If $T_1 \sqsubseteq T_2$ *then* $N(T_1, T_2) = T_2$

Normalization should clearly be commutative. It should also be associative, since the normalization of three specifications should be independent of how that normalization is achieved. Common refinement expresses the fact that implementations conforming to $N(T_1, T_2)$ should also conform to the original specifications T_1, T_2. Note that this does not guarantee that there exists a conforming implementation, only that if one does exist then it must conform in the manner shown.

A normalization function N and a specification T_1, induce a function $N_{T_1} : Spec \longrightarrow Spec^\bullet$ given by $N_{T_1}(T_2) = N(T_1, T_2)$. It is then clear that we can consider this as a function $N_{T_1} : Spec^\bullet \longrightarrow Spec^\bullet$, since if $T_2 \equiv T_3$ then $N(T_1, T_2) = N(T_1, T_3)$. This expresses our intuition that if we translate in different ways then the normalization should be the same. We can thus view normalization as a function $N : Spec^\bullet \times Spec^\bullet \longrightarrow Spec^\bullet$.

Proposition 5 *Any normalization function N satisfies the following properties:*

(i) $N(T, T) = T$ - *idempotency*
(ii) $N_{T_1}(N_{T_1}(T_2)) = N_{T_1}(T_2)$ - *(by associativity)*
(iii) N is not one-one.

Proof
The first two are obvious. For the third, it is sufficient to exhibit two pairs of specifications (T_1, T_2), (T_3, T_4) with $N(T_1, T_2) = N(T_3, T_4)$ but $T_1 \not\equiv T_3$, $T_2 \not\equiv T_4$. The example

$$T_1 = [x = 1, y = 2], \quad T_2 = [z = 3], \quad T_3 = [x = 1], \quad T_4 = [y = 2, z = 3]$$

suffices since $N(T_1, T_2) = N(T_3, T_4) = [x = 1, y = 2, z = 3]$. □

Unification can now be expressed formally in terms of equivalence and normalization.

Definition 3 *Let $S_1 \in Spec_1, S_2 \in Spec_2$. Then $S_3 \in Spec_3$ is the unification of S_1 and S_2 (written $S_3 = \mathcal{U}(S_1, S_2)$) if there exists specifications $T_1, T_2 \in Spec_3$ such that $S_1 \equiv T_1$, $S_2 \equiv T_2$ and $S_3 = N(T_1, T_2)$.*

Thus unification is the process of transforming specifications from (potentially) different viewpoints into a common language, and then normalizing in order to identify the commonality between the two specifications.

Proposition 6 *For arbitrary specifications and any unification the following hold:*

(i) $\mathcal{U}$ *is commutative and associative*
(ii) $S_1, S_2 \sqsubseteq \mathcal{U}(S_1, S_2)$ *- common refinement*
(iii) $\mathcal{U}(S, S) = S$

Proof
These follow easily from analogous properties of normalization. □

4.4 Consistency and Validity

Consistency is a relation, denoted $\mathcal{C}$ in the sequel, between two specifications defined in arbitrary viewpoints. Thus $\mathcal{C} \subseteq SPEC \times SPEC$. Informally two specifications are consistent if their unification can be implemented. Formally we make the definition:

Definition 4 *Let* $S_1 \in Spec_1, S_2 \in Spec_2$*. Then* $(S_1, S_2) \in \mathcal{C}$ *iff there exists an implementation* I *and a specification language* L_3 *such that* $S_1 \equiv T_1$*,* $S_2 \equiv T_2$ *and* $N(T_1, T_2) conf I$ *for some* $T_1, T_2 \in Spec_3$*.*

Proposition 7 *Consistency is a symmetric relation, but it is neither reflexive nor transitive.*

Proof
The first is obvious. To see that $\mathcal{C}$ is not reflexive we just need to consider an inconsistent specification, for example $S = [x = 0, x = 1]$. Then consistency amounts to showing that there exists an implementation I with $S conf I$. But this is clearly impossible.

To see that $\mathcal{C}$ is not transitive we will find specifications S_1, S_2, S_3 with $(S_1, S_2) \in \mathcal{C}$, $(S_2, S_3) \in \mathcal{C}$ but $(S_1, S_3) \notin \mathcal{C}$. An appropriate example is:

$$S_1 = [x = y, x \neq 0], \quad S_2 = [x = z, x \neq 0], \quad S_3 = [y = -z, x = z]$$

□

ODP specifications will be defined from a number of viewpoints, consistency checks need to be applied to an arbitrary number of specifications and not just two, so we extend the definition of consistency to the following.

Definition 5 *Three specifications* S_1, S_2, S_3 *are (globally) consistent if* $(S_1, \mathcal{U}(S_2, S_3)) \in \mathcal{C}$*. The extension of consistency to an arbitrary finite number of specifications is done in the obvious manner.*

In order to show that this definition is well-founded, we shall prove the following proposition.

Proposition 8 *Let* $S_1 \in Spec_1, S_2 \in Spec_2, S_3 \in Spec_3$*. Then*

$$(S_1, \mathcal{U}(S_2, S_3)) \in \mathcal{C} \quad \textit{iff} \quad (S_2, \mathcal{U}(S_1, S_3)) \in \mathcal{C} \quad \textit{iff} \quad (S_3, \mathcal{U}(S_1, S_2)) \in \mathcal{C}$$

Proof
Suppose that $(S_1, \mathcal{U}(S_2, S_3)) \in \mathcal{C}$. Then there exists a language L_4 and an implementation I such that $S_1 \equiv T_1, \mathcal{U}(S_2, S_3) \equiv T_4$ and $N(T_1, T_4) conf I$ for some $T_1, T_4 \in Spec_4$.

Let $U = \mathcal{U}(S_2, S_3)$. Then $U = N(T_2, T_3)$ for some $T_2, T_3 \in Spec_4$ with $S_2 \equiv T_2$ and $S_3 \equiv T_3$. By definition $U \equiv T_4$. Thus

$$N(T_1, N(T_2, T_3)) \ conf \ I$$

Since N is associative, $N(N(T_1, T_2), T_3) \ conf \ I$ where $S_3 \equiv T_3$ and $N(T_1, T_2) = \mathcal{U}(S_1, S_2)$. Hence $(S_3, \mathcal{U}(S_1, S_2)) \in \mathcal{C}$. The other cases are proved in a similar fashion. $\square$

The relation between arbitrary (or global) consistency and pairwise consistency can be seen in the following proposition.

Proposition 9
1. Global consistency of three or more specifications implies pairwise consistency.
2. Pairwise consistency does not imply global consistency.

Proof
For the first let $S_1 \in Spec_1, S_2 \in Spec_2, S_3 \in Spec_3$ be consistent specifications. Then there exists a language L_4 and an implementation I such that $S_1 \equiv T_1, \mathcal{U}(S_2, S_3) \equiv T_4$ and $N(T_1, T_4) conf I$ for some $T_1, T_4 \in Spec_4$. Since $\mathcal{U}(S_2, S_3) \equiv T_4 \sqsubseteq N(T_1, T_4)$ we have $\mathcal{U}(S_2, S_3) conf I$. Hence $(S_2, S_3) \in \mathcal{C}$. The other cases are similar.

To show that pairwise consistency does not imply global consistency it is enough to find an example where $(S_1, S_2), (S_2, S_3), (S_1, S_3) \in \mathcal{C}$, but $(S_1, \mathcal{U}(S_2, S_3)) \notin \mathcal{C}$. An appropriate example is

$$S_1 = [x = y, x \neq 0], \quad S_2 = [x = z], \quad S_3 = [y = -z]$$

$\square$

Internal validity is a property of single specifications. Informally a specification is internally valid when "there are no conflicts between its properties and those implicit properties required of the specification; and when there is at least one example product that could conform to it" [?].

When the ODP development process is viewed formally, all requirements of a system should be captured in the initial specification by the process of requirements capture. In our formal interpretation we will not consider any requirements beyond those present in the specification. With that interpretation a specification is internally valid when there is at least one example product that could conform to it. This is just the property that a specification is consistent with itself.

Definition 6 *A specification S is internally valid if $(S, S) \in \mathcal{C}$.*

The following is an obvious consequence of this definition:-

Proposition 10 *$\mathcal{C}$ is reflexive if we restrict the set of specifications that it is defined over to be those that are internally valid.*

5 Discussion of the Framework

The framework presented here makes a distinction between specifications and implementations. Other approaches to conformance and the definition of implementation relations have not made such a distinction [?, ?, ?, ?]. For example the relation conf in LOTOS is a non-transitive relation which has been taken as the formal basis for conformance testing in [?]. In fact the theory of equivalence and refinement remains much the same whether or not one enforces this distinction. We have done so here, so that we can later unify the theory of conformance testing into that of conformance assessment. Our work could be extended to combine the *conf* relation presented here with non-transitive implementation relations such as conf in LOTOS.

Although the conformance relation *conf* is the basic relation in our abstract framework, in actual practice, it is the definition of refinement that is important. Different languages have different notions of refinement. It is improbable to expect that they will all satisfy Definition 2 that we took as the basic condition of refinement. For example we might allow refinements to satisfy the weaker condition:

$$S_1 \sqsubseteq S_2 \text{ implies that } \{I : S_2 \ conf \ I\} \subseteq \{I : S_1 \ conf \ I\}$$

Likewise when defining consistency of two specifications we require the production of an actual implementation which conforms to the unification of the two specifications. In practice in a development situation we might show that there exists an implementation specification IS which is the refinement of the unification, and show that the specification IS is consistent by conformance testing.

6 Examples

The following two subsections illustrate the mathematical framework we have presented in this paper. The examples considered are very simple. Any more realistic illustrations of the framework are beyond the scope of this paper. The first subsection presents examples of consistency and refinement in the Raise specification language [?], while the second subsection illustrates the framework using examples in LOTOS [?, ?]. The Raise examples are concerned with consistency arising from specification of data properties, while the LOTOS examples consider the consistency between behavioural specifications. Both the sets of examples are restricted to consideration of consistency between two specifications which are in the same language. Thus, our examples do not consider the important issue of cross language translation, although, our framework has explored this issue.

6.1 RAISE

The RAISE Specification Language (RSL) is a (wide-spectrum) language which allows specification in three different paradigms; declarative, imperative and concurrent. The language is expression-oriented, with a 'pure' functional core. On top of this are added expressions which can read or write to variables, and take input from and give output to communication channels. Sufficient checks (or imprecations) are made to ensure that side-effects and communications are restricted to appropriate parts of the language, *axioms* are expected not to have side-effects, for instance.

The formal implementation relation defined in RAISE has been chosen to ensure that an implementation can be substituted for its specification. A specification in RSL is a collection of *objects* or *schemes*, each of which is built from a class expression. Basic class expressions in RSL correspond to theory presentation (signature + axioms) in algebraic specification languages.

A class expression signifies a theory and denotes a class of models. One class expression implements another if every provable consequence of the later is a provable consequence of the former. This is equivalent to sub-classing of models (under a technical assumption). If we denote the implementation relation by *impl*, and provable consequence by $\vdash$, we have

$$S_1 \ impl \ S_2 \quad \text{if} \quad S_1 \vdash \phi \rightarrow S_2 \vdash \phi$$

Define *impl* for class expressions by saying that $class\_expr_1$ *impl* $class\_expr_2$ if both the following hold:

- $class\_expr_2$ statically implements $class\_expr_1$
- The theory of $class\_expr_1$ is provable in $class\_expr_2$

Some simple examples illustrate these concepts. Consider the following schemes:

S = **class type** T **value** x,y : T **axiom** x$\neq$ y **end**
S_1 = **class type** T = **Int value** x : T = 1, y : **Int** = 2 **end**
S_2 = **class value** x : **Int** = 1, y : **Int** = 2 **end**
S_3 = **class type** T = **Int value** x : T = 1, y : T = 1 **end**

S_1 implements S; its signature includes that of S and it satisfies the theory of S that x is different from y.

S_2 does not statically implement S (it defines no type T). S_2 is implemented by S_1.

It is clear that the RSL implementation relation satisfies the requirements of a refinement relation on our framework.

6.1.1 Consistency checking in RAISE

Let S, S_1, S_2 and S_3 be the schemes defined above. Simple examples of consistency checking can be found by considering combining the schemes via normalization. Note that this is sufficient for unification since we do not cross a language boundary.

1. S_1 implements S. So the normalization of the two, $N(S, S_1)$, is just S_1.

2. S_1 implements both S and S_2. In fact S_1 is the greatest upper bound, wrt *impl*, which implements both S and S_2. As one would expect in these circumstances, $N(S_1, S_2) = S_1$.

3. As an example of two inconsistent specifications we consider the normalization of S and S_3. Clearly here, if the values x and y refer to the same entities in both schemes, then these schemes are inconsistent. This can be seen since their normalization is

S_4 = **class type** T = **Int value** x : T = 1, y : T = 1 **axiom** x$\neq$ y **end**

The fact that this is inconsistent can be deduced from the fact that:

$$S_4 \vdash x = y \qquad S_4 \vdash x \neq y$$

6.2 LOTOS

Illustration of our framework can also be presented in the LOTOS specification language (background to LOTOS can be found in [?, ?]). In particular, by way of contrast to the previous examples which concentrated on consistency resulting from data relationships, here we will illustrate consistency resulting from behavioural properties.

The mechanism highlighted in this paper for checking the consistency of two specifications involves first normalizing the two specifications into a combined form and then showing that a 'conformant' implementation exists. There are two elements to normalization, a translation in order to enforce the correspondence between terms in the two specifications and the actual process of combining the two specifications. In terms of LOTOS behaviours we can illustrate the first of these two elements in terms of the renaming of events. For example, if we had the following two (LOTOS like) specifications of a trivial drinks machine:

DM_1 := 100pence ; tea ; stop

and

DM_2 := 1pound ; tea ; stop

then there is clearly a correspondence between the event 100pence in DM_1 and 1pound in DM_2. Thus, we would perform suitable translations in order to reflect this correspondence. For example, we might translate the two machines to:

DM_1 := coin ; tea ; stop

and

DM_2 := coin ; tea ; stop

Now when we combine the two specifications the correspondence between the first events in the two behaviours will be explicit. Let us now give some examples of checking consistency between LOTOS behaviours. Consider then the following, often not very sensible, drinks machines, where we assume that all necessary translations have been made to reflect correspondences between events:-

DM_3 := coin; tea ; stop [] tea; coin; stop

DM_4 := coin; stop ||| tea; stop

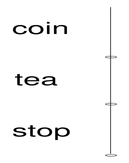
DM_5 := coin; tea; stop

DM_6 := coin; coin; stop

It should be clear that DM_3 and DM_4 are consistent, in fact they are equivalent. In addition, DM_5 is a refinement of both of DM_3 or DM_4 and is also consistent with both. However, DM_6 is inconsistent with all of the other three drinks machines. We can illustrate these consistency relationships, by composing behaviours together in parallel (such that common events are synchronised) and then determining whether any ambiguities result. For example, the normalization of DM_3 and DM_5, $N(DM_3, DM_5)$, is the following behaviour:-

(coin;tea;stop[]tea;coin;stop) |[coin,tea]| (coin;tea;stop)

The two behaviours can be seen to be consistent, ie $(DM_3, DM_4) \in \mathcal{C}$, since the behaviour of $N(DM_3, DM_5)$:-



does not contain any ambiguities. In contrast, $N(DM_5, DM_6)$:

(coin;tea;stop) |[coin,tea]| (coin,coin,stop)

will yield a deadlock state, as follows:-

$$\begin{array}{c} \big\downarrow \text{coin} \\ \text{tea;stop |[tea,coin]| coin;stop} \end{array}$$

and thus, $(DM_5, DM_6) \notin \mathcal{C}$.

Thus, we take the existence of an unambiguous common behaviour to imply that a common implementation exists. However, the existence of a deadlock state in the common behaviour suggests that an implementation which is consistent with both specifications does not exist. This does not actually completely satisfy our definition of consistency, since it may still be the case that the common behaviour is not conformant to the target product. We have overlooked this requirement, since consideration of conformance to physical products is realistically beyond the scope of such simple examples.

In summary then, these very simple examples suggest that consistency checking in LOTOS takes the following general form. Normalization involves translating event names in order to reflect correspondences between terms and then combining behaviours using the general parallel operator, such that common events are synchronised. Consistency checking then involves considering whether the resultant behaviour contains ambiguities, characteristically deadlocks.

7 Conclusion

This paper has made a first step towards the development of a formal theory of consistency between specifications. We believe that consideration of this issue is timely. In particular, there is an urgent need for a formal understanding of consistency within the Open Distributed Processing setting.

Due to the limited scope of this paper we have only been able to illustrate our framework with very simple examples. However, we have investigated the consistency properties arising from a number of more realistic specifications. In particular, we are involved in ongoing work on checking the consistency of the existing LOTOS and Z specifications of the ODP trader. The ultimate objective of our work is to develop automated techniques for consistency checking which can be used within the ODP product development framework.

Although the work presented in this paper is at an early stage of development we believe it makes a valuable first step towards the development of a theory of consistency checking within the Open Distributed Processing framework.