Consistency Checking of RM-ODP Specifications

Kenneth Bacławski
College of Computer Science,
Northeastern University, Boston, Massachusetts 02115
kenb@ccs.neu.edu

Mieczysław M. Kokar
Department of Electrical and Computer Engineering,
Northeastern University, Boston, Massachusetts 02115
kokar@coe.neu.edu

Jeffrey Smith
Mercury Computer Systems Inc.
jsmith@coe.neu.edu

Jerzy Letkowski
School of Business, Western New England College, Springfield, MA 01119
jletkows@wnec.edu

Abstract

Ensuring that specifications are consistent is an important part of specification development and testing. In this paper we introduce the ConsVISor tool for consistency checking of RM-ODP specifications. This tool is a category theory based consistency checker for formal specifications in a variety of languages, including both graphical and non-graphical modeling languages. Because RM-ODP supports multiple viewpoints, it is necessary to have a logical framework that can compose viewpoints and that can detect inconsistencies (feature interactions) among multiple viewpoints. The ConsVISor tool composes viewpoints by using the colimit operation of category theory. ConsVISor checks consistency using a combination of theorem proving and model-based reasoning. Some examples of the use of the tool are given.

1 Introduction

Consistency of a formal specification is a fundamental requirement. If a formal specification is inconsistent, then one can logically derive anything, so that from a logical standpoint, the specification is useless. The Reference Model for Open Distributed Processing (RM-ODP) [8] is particularly vulnerable to inconsistency problems because it supports multiple viewpoints. Even if every viewpoint is consistent by itself, it is possible for the combination of the viewpoints to be inconsistent. When inconsistencies arise from incompatibilities among multiple views, the inconsistencies are called *feature interactions*. To ensure global consistency

of an RM-ODP specification, a consistency checking tool must support the composition of multiple viewpoints.

Another important property of RM-ODP is its support for the specification of the dynamics of a system, not just the static model. A consistency checker must therefore support dynamic modeling and must be able to check for reachability and deadlock avoidance.

ConsVISor is a consistency checking tool based on category theory [15, 18]. It has a graphical user interface that supports modeling formalisms such as the General Relationship Model (GRM) [7, 9] and the Unified Modeling Language (UML) [1, 2]. Consequently, ConsVISor also supports graphical specification standards that are described in these languages, such as the Meta Object Facility (MOF), Component Warehouse Model (CWM), CORBA, Software Communication Architecture (SCA), and Software Radio Architecture (SRA). Consistency checking of RM-ODP specifications becomes more important to the UML, and its derivative standards, since one proposed basis for the UML 2.0 Infrastructure is to adopt some of the basic RM-ODP definitions (e.g. objects, actors, etc.) as its core.

In [10] it is argued that formal method based development involving consistency checks and theorem proving is more cost-effective than relying on testing. There are a number of ways to deal with inconsistencies. For instance, in [12] three levels of consistency handling are proposed: ignore, tolerate (defer, circumvent or ameliorate) and resolve.

In the next section we introduce some background in category theory that we will use in our examples, shown in Section 3. We also discuss how the ConsVISor tool is used. A demonstration of the tool is available online at `http://vis.home.mindspring.com/-consvisor.html` We then discuss the underlying methodology used for consistency checking in Section 4, and we end with conclusions and future work.

2 Background

Category theory has been gaining popularity in formal approaches to software engineering. While many formal specification techniques provide the ability to describe the static structure and dynamic behavior of objects, category theory explicitly captures relationships between specifications. The motivation for using a category theory based approach includes:

- Several organizations (e.g. the pUML, OMG subgroups, etc.) have been searching for diagrammatic techniques for formal semantic representations.
- To have tool support for developing domain theories (ontological engineering), code from specifications, specifications from code and specification refinement.
- The graphical and language support for specification composition is more powerful than the include mechanism of languages such as `Z` and is more powerful than simple algebraic specification techniques.
- The general theory of diagrams enables nodes and arcs to be related in a way that preserves the structure and content of the source diagram in the target diagram, making

it possible for diagrams of specifications to be parameterized and instantiated where the result is a diagram, not a single specification [13].

- Category theory is intensional versus extensional - implicit versus explicit [4], i.e. to check a specification against a category theory based form of semantics one would show that the class of models of any instance is within the class of models of the specification, rather than show an explicit association (of classes of models with algebraic specifications in both the source and target specifications).

- Category theory has been described as a potential formal foundation for the emerging UML RT standard [6].

The rest of this section briefly describes the category theory terms used in this paper. Theory-based algebraic specification is concerned with:

1. modeling system behavior using algebras (a collection of values and operations on those values) and axioms that characterize algebra behavior, and

2. composition of larger specifications from smaller specifications. Composition of specifications is accomplished via specification building operations defined by category theory constructs [16].

A theory is the set of all assertions that can be logically proved from the axioms of a given specification. Thus, a specification defines a theory and is termed a theory presentation.

In algebraic specifications, the structure of a specification is defined in terms of sorts, which are abstract collections of values, and operations over those sorts. This structure is called a signature. A signature describes the structure of a solution, but it does not describe the meaning of that structure. To specify semantics, a signature is extended with axioms defining the intended semantics of signature operations. A signature with its associated axioms is a specification.

An algebraic specification as defined above allows one to formally define the internal structure of objects and classes of objects, but does not provide the capability to reason about relationships between objects and classes. To create theory-based algebraic specifications that parallel object-oriented specifications, the ability to define and reason about relationships between theories, similar to those used in object-oriented approaches (inheritance, aggregation, etc.) must be available. Category theory is an abstract mathematical theory used to describe the external structure of various mathematical systems [17] and is used to describe relationships between specifications.

Specification morphisms are the basic instrument for defining and refining specifications. However, we can extend the notion of a specification morphism to allow for the creation of new specifications from a set of existing specifications. Often two specifications derived from a common ancestor specification need to be combined. The desired combination consists of the unique parts of two specifications and some "shared part" common to both specifications (the part defined in the shared ancestor specification). This combining operation is a colimit.

Conceptually, a colimit defines a specification that is the "shared union" of a set of specifications based on the morphisms between the specifications. These morphisms define equivalence classes of sorts and operations. In addition to the specification defined by the colimit, the colimit operation creates a specification morphism from each specification to the specification defined by the colimit.

From morphisms and colimits, we can construct specifications in a number of ways. We can:

1. build a specification from a signature and a set of axioms,
2. form the union of a set of specifications via a colimit,
3. rename sorts or operations via a specification morphism and
4. parameterize specifications.

Many of these methods have been useful in translating object-oriented specifications into theory-based specifications [3, 14].

3 Examples of Feature Interaction

In this section we give some examples of feature interaction inconsistencies that can arise in data modeling.

3.1 Vehicles

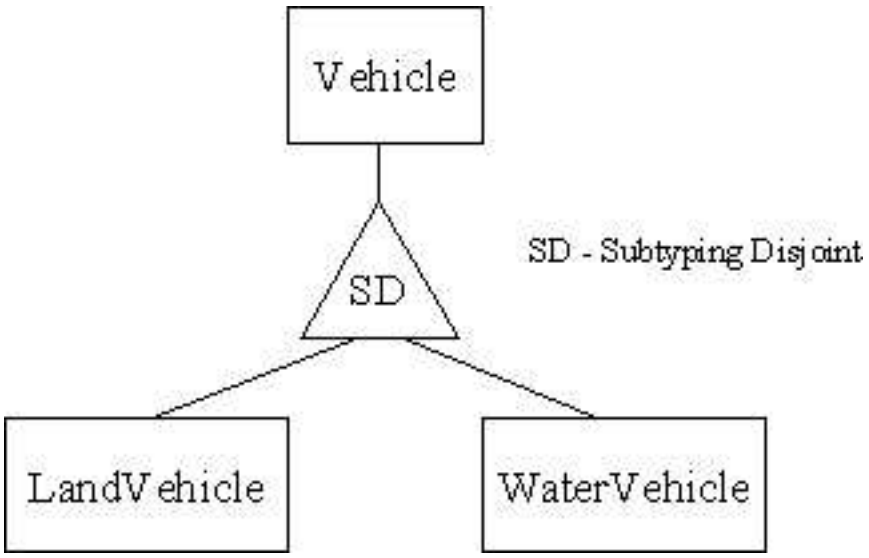

Figure 1: Vehicle Specification

In this example, one models vehicles as being either land vehicles or water vehicles. This view is shown in Figure 1. The vehicle is converted to a collection of facts in many-sorted first-order predicate logic. The following is the vehicle model expressed in terms of such logical facts using a single 3-place predicate called `pv`. The `pv` predicate can be interpreted as asserting that the relationship in the first position relates the object in the second position with the object in the third position.

```
pv(type,vehicle,class).
pv(type,landVehicle,class).
pv(type,waterVehicle,class).
pv(subTypeOf,landVehicle,vehicle).
pv(subTypeOf,waterVehicle,vehicle).
pv(disjointWith,landVehicle,waterVehicle).
```

In the second view, an amphibious vehicle class is introduced as in Figure 2. This adds the following logical facts to the ones above:

```
pv(type,landVehicle,class).
pv(type,waterCraft,class).
pv(type,amphibian,class).
pv(subTypeOf,amphibian,landVehicle).
pv(subTypeOf,amphibian,waterCraft).
```

When the two views are composed, the colimit introduces an additional fact:

```
pv(equivalentTo,waterCraft,waterVehicle).
```

The disjointness in the first view is incompatible with the amphibian subtype introduced in the second view.

Two steps are required to check consistency of these two views.

1. The two views must be composed. This is done by forming the colimit in which the WaterVehicle of the first view is identified with the WaterCraft of the second view.

2. The theorem prover is invoked. In this case, the theorem prover shows that the Amphibian class is inconsistent (i.e., it cannot be instantiated).

The axiom that fails to be satisfied for this example is the following:

```
axiom Disjoint-With is pv?(disjointWith, c1, c2) <=>
  class?(c1) & class?(c2) &
  not (ex(x:Object) (pv?(type, x, c1) & pv?(type, x, c2)))
```

In this axiom, `ex` is the existential quantifier, `&` is the logical `and` operator, and `|` is the logical `or` operator.

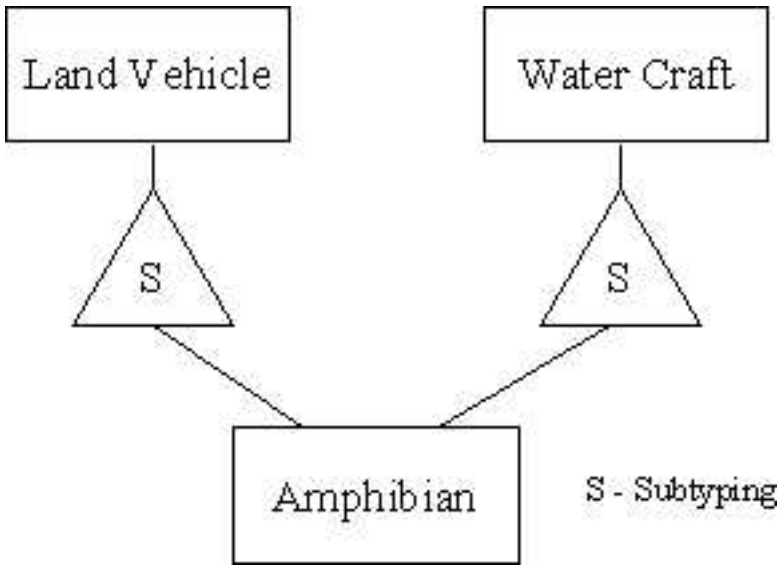

Figure 2: Amphibious Vehicle

3.2 Expressions

In this example, one is modeling an expression consisting of binary or higher operators that can be combined recursively. For example, $(x + y + 5) * (z + 3) * (a + b)$ would be such an expression. This includes operators such as addition and multiplication. In the first view, the notion of Expression is introduced with two disjoint, inclusive subtypes, Elementary and Operation. An elementary expression has no further substructure. It would include constants and variables. An operation is a subexpression that combines operands that are either elementary expressions or other operations.

In the first view, shown in Figure 3, the classes are shown, together with the subtype relationships between them. The second view in Figure 4 shows the operand relationship which connects an operation object with the subexpressions being combined by the operation.

As in the Vehicle example, the views are combined by using a colimit to form a single specification as follows:

```
spec EXPRESSION-ELEMENTARY-OPERATION-COLIMIT is
  import colimit of diagram
    nodes T1: TRIV, T2: TRIV, OP-COMPONENT-EXP,
              ELEMENTARY-OPERATION-SUBTYPE-EXPRESSION
    arcs
      T1 -> OP-COMPONENT-EXP: {e -> Exp},
      T1 -> ELEMENTARY-OPERATION-SUBTYPE-EXPRESSION: {e -> Expression},
      T2 -> OP-COMPONENT-EXP: {e -> Op},
      T2 -> ELEMENTARY-OPERATION-SUBTYPE-EXPRESSION: {e -> Operation},
  end-diagram
end-spec
```

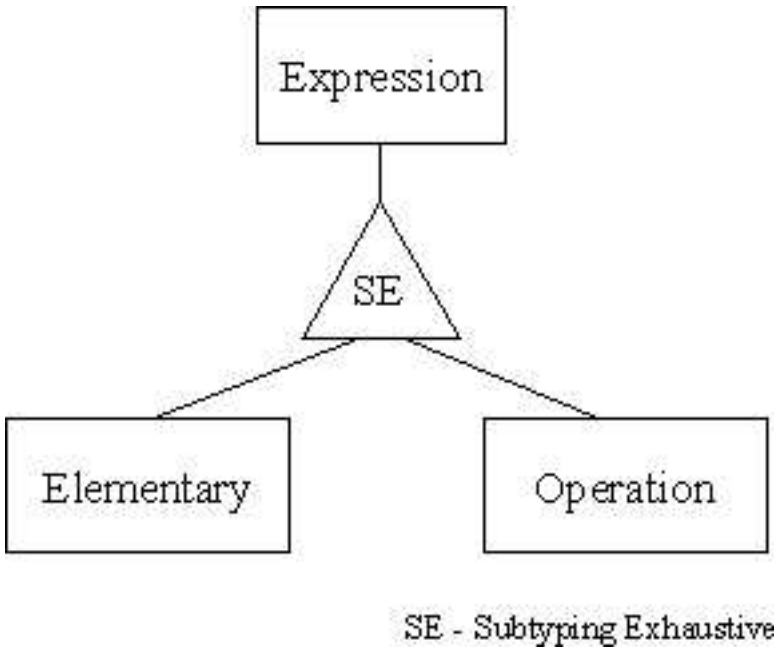

Figure 3: Expression Specification

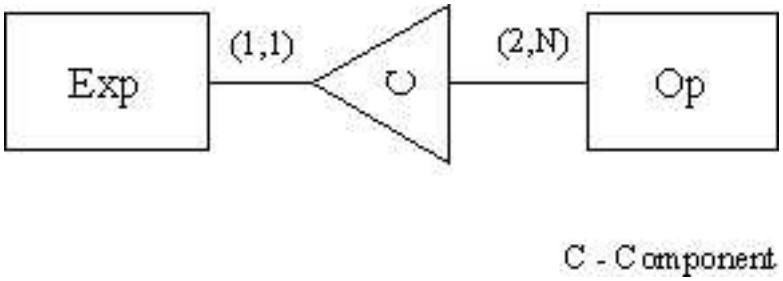

Figure 4: Operands

In the specification above, the Expression class in the first view is identified with the Exp class in the second view. The specification is written in Slang, the language of Specware^{TM} [15].

The resulting specification is not obviously inconsistent. The multiplicity constraints of the relationship between Operation and Expression implies that there are at least twice as many instances of Operation as there are instances of Expression. However, Operation is a subtype of Expression, so every instance of Operation is also an instance of Expression. Using the symbol # to mean *the number of instances*, we have shown that:

$$\#\text{Expression} \geq \#\text{Operation} \geq 2\#\text{Expression}$$

which implies that the Operation class (as well as the Expression class) is either empty or has an infinite number of elements.

The problem with this model is that the cardinality constraints are in the wrong order. Reversing cardinality constraints is a common mistake in data models, and it is much easier to make such a mistake when there are multiple views. The ConsVISor tool will, in this case, warn the user that some of the classes cannot be instantiated.

In Figure 5 we show what the ConsVISor screen looks like for the Expression/Operation example above. ConsVISor uses a Prolog fact and rule base (`base.pl`) for the underlying modeling system, in this case RM-ODP. The model being examined is also converted to Prolog (`expression.pl`) that is combined with `base.pl` and run. The Prolog output is then shown by ConsVISor in the bottom of the window.

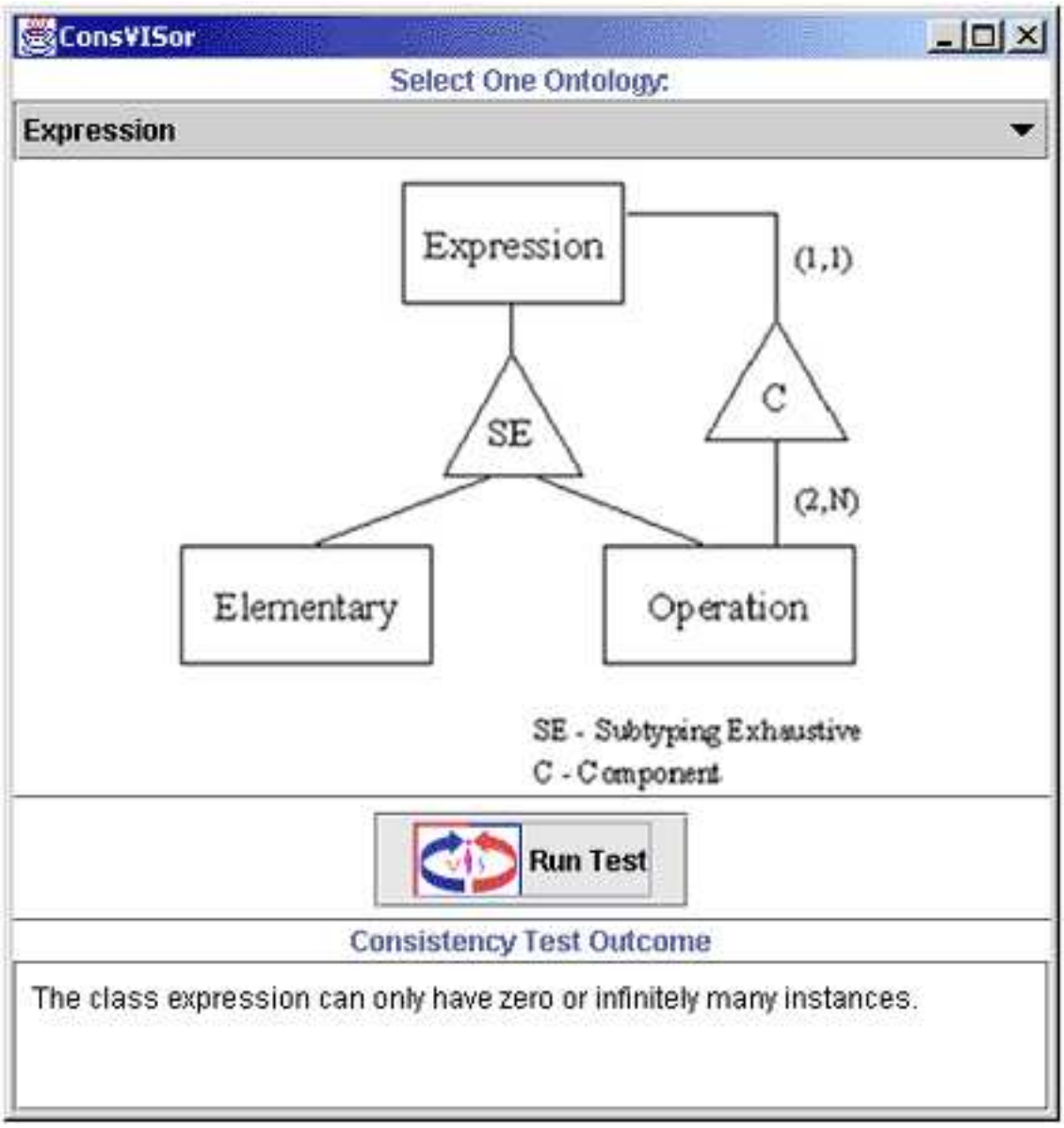

Figure 5: ConsVISor Screen

4 Methods

The ConsVISor tool uses a combination of techniques to perform consistency checking. This is necessary because of the Gödel Incompleteness Theorem which states that the consistency of a nontrivial theory is not expressible within the theory. Therefore, checking consistency is, in general, an undecidable problem. This does not, however, make it impossible. The ConsVISor tool attacks the problem as follows:

1. A model checking component uses Prolog to find syntactic inconsistencies and to perform model checking. If a model can be verified, then the specification is consistent. The model checking component prints diagnostics that can be helpful even when the specification is consistent.

2. A theorem proving component attempts to prove that there are inconsistencies. If there is an inconsistency, then there is a good chance that the theorem prover can find it. Since this algorithm need not terminate, it either finds an inconsistency or it gives up when it runs out of time or space.

5 Conclusions and Future Work

We have introduced the ConsVISor consistency checking tool that can be used to verify consistency of RM-ODP specifications. A demonstration version is now available [11]. We are in the process of introducing new features to ConsVISor. In particular, we are introducing more complex model building capabilities, including the ability to evolve a model progressively through versions beginning with a baseline. We are also planning to add sheaf theory semantics [5] to deal with dynamic systems so that one can check dynamic consistency conditions such as reachability and deadlock avoidance.

References

[1] G. Booch, I. Jacobson, and J. Rumbaugh. *OMG Unified Modeling Language Specification*, March 2000. Available at `www.omg.org/technology/documents/formal/-unified_modeling_language.htm`.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *UML Semantics*, September 1997.

[3] S. DeLoach. *Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specifications*. PhD thesis, Air Force Institute of Technology, WL AFB, OH, June 1996. Ph.D. Dissertation.

[4] A. Evans and S. Kent. Core meta-modeling semantics of UML: The pUML approach. In *UML99 - The Unified Modeling Language: Beyond the Standard, Second International Conference Proceedings*. Springer-Verlag, October 1999. ISBN 3-540-66712-1.

[5] J. Goguen. Sheaf semantics for concurrent interacting objects. *Mathematical Structures in Computer Science*, 2(2):159–191, 1992. citeseer.nj.nec.com/goguen92sheaf.html.

[6] R. Grosu, M. Broy, B. Selic, and G. Stefânescu. What is behind UML-RT? In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 75–90. Kluwer Academic, October 1999. ISBN 0-7923-8629-9.

[7] *Information Technology–Open Systems Interconnection–Management Information Services–Structure of Management Information–Part 7: General Relationship Model (ISO/IEC 10165-7.2)*, August 1993.

[8] *Basic Reference Model for Open Distributed Processing. Use of formal specification techniques for ODP (ISO/IEC JTC1/SC21/WG7 N 753)*, November 1992.

[9] H. Kilov and J. Ross. *Information Modeling: An Object-Oriented Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[10] S. King, J. Hammond, R. Chapman, and A. Pryor. Is proof most cost-effective than testing? *IEEE Transactions on Software Engineering*, 26(8):675–686, 2000.

[11] M. Kokar, J. Letkowski, K. Baclawski, and J. Smith. The ConsVISor consistency checking tool, March 2001. `www.vistology.com/consvisor/`.

[12] B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *IEEE Computer*, pages 24–29, April 2000.

[13] T. Schorsch. *Formal Representation and Application of Software Design Information*. PhD thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, September 1999.

[14] J. Smith. *UML Formalization and Transformation*. PhD thesis, Northeastern University, Boston, MA, December 1999.

[15] *Specware*$^{\text{TM}}$ *User Manual: Specware*$^{\text{TM}}$ *Version Core4*, October 1994.

[16] Y. Srinivas. Algebraic specification: Syntax, semantics, structure. Technical Report 90-15, Department of Information and Computer Science, University of California, Irvine, June 1990.

[17] Y. Srinivas and R. Jüllig. Specware: Formal support for composing software. Technical Report 90-14, Department of Information and Computer Science, University of California - Irvine, 1990.

[18] R. Waldinger et al. *Specware*$^{\text{TM}}$ *Language Manual: Specware*$^{\text{TM}}$ *2.0.3*, March 1998.