INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
A Formal Model of Software Subsystems

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Stephen Edwards, B.S., M.S.

* * * * *

The Ohio State University

1995

Dissertation Committee:

Dr. Bruce W. Weide
Dr. William F. Ogden
Dr. Spiro Michaylov

Approved by

[Signature]

Adviser
Department of Computer and Information Science
© Copyright by
Stephen Edwards
1995
To my wife, Diane Hodge, for all of her support and hard work,

and my parents, Hilary and Shirley Edwards,

for their infinite love and patience.
ACKNOWLEDGEMENTS

My sincerest thanks go to my advisor, Bruce Weide. Without your effective guidance, continual interest, and strong support, finishing this research would not have been possible. I am also grateful to the other current and former members of my reading committee, Bill Ogden, Spiro Michaylov, and Timothy Long. Their constructive suggestions not only strengthened the research, they greatly improved the readability of the final result.

I also wish to thank the past and present members of the Reusable Software Research Group at The Ohio State University. Not only have the group members been a true pleasure to work with, but they have patiently put up with numerous and sometimes confusing discussions as I worked my way through the critical issues. Their comments have likewise improved the presentation of my work, and I would especially like to thank Joseph Hollingsworth, Wayne Heym, and Murali Sitaraman.

In addition, I wish to thank my parents, Hilary and Shirley Edwards, for getting me my first computer and starting me on this long but interesting path. Thanks also go to my sister, Leanne Sampson, for her support and encouragement. Finally, I am greatly indebted to my wife, Diane Hodge, for her understanding, tolerance, and sympathy during this period, without which I would not have been able to finish this work.
VITA


1988 ............................................. B.S. Electrical Engineering, California Institute of Technology.


1992 ............................................. M.S. Computer and Information Science, The Ohio State University.

1991–present ............................... Graduate Research Associate, The Ohio State University.

Publications

Research Publications


**Instructional Publications**


**Fields of Study**

Major Field: Computer and Information Science

Studies in:

- Software Engineering
  - Prof. Bruce W. Weide
- Formal Methods in Programming Languages
  - Prof. William F. Ogden
- Information Retrieval and Databases
  - Prof. Gary Perlman
# Table of Contents

**DEDICATION** ................................. ii

**ACKNOWLEDGEMENTS** ..................... iii

**VITA** ........................................ iv

**LIST OF TABLES** ............................... xii

**LIST OF FIGURES** ............................. xiii

---

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Introduction .................................. 1</td>
</tr>
<tr>
<td></td>
<td>1.1 Why Conventional Languages Fail ....... 3</td>
</tr>
<tr>
<td></td>
<td>1.2 Correcting the Deficiency ............... 5</td>
</tr>
<tr>
<td></td>
<td>1.3 Contributions ................................ 6</td>
</tr>
<tr>
<td></td>
<td>1.4 Organization ................................ 7</td>
</tr>
<tr>
<td>II</td>
<td>Understanding Software: Mental Models ....... 11</td>
</tr>
<tr>
<td></td>
<td>2.1 Mental Models ................................ 11</td>
</tr>
<tr>
<td></td>
<td>2.2 The Inadequacy of Conventional Languages .... 18</td>
</tr>
<tr>
<td></td>
<td>2.2.1 Software Interfaces Fail to Support Mental Models .... 18</td>
</tr>
<tr>
<td></td>
<td>2.2.2 Software Implementations Fail to Support Mental Models .... 20</td>
</tr>
<tr>
<td></td>
<td>2.2.3 Why Comments Alone are not Enough ...... 21</td>
</tr>
<tr>
<td></td>
<td>2.2.4 Conventional Programming Languages Do Not Support Mental Models .... 24</td>
</tr>
<tr>
<td></td>
<td>2.3 ACTI Addresses the Inadequacy ............. 25</td>
</tr>
</tbody>
</table>

vii
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5.9</td>
<td>Concrete Template Environments ($CTE$)</td>
<td>106</td>
</tr>
<tr>
<td>4.5.10</td>
<td>Interpretation Mapping Template Environments ($IMTE$)</td>
<td>107</td>
</tr>
<tr>
<td>4.6</td>
<td>Abstract Instances ($AI$)</td>
<td>107</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Variable Environment Signatures ($VES$)</td>
<td>111</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Operation Environment Signatures ($OES$)</td>
<td>112</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Abstract Template Environment Signatures ($ATES$)</td>
<td>112</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Concrete Template Environment Signatures ($CTES$)</td>
<td>113</td>
</tr>
<tr>
<td>4.7</td>
<td>Concrete Templates ($CT$)</td>
<td>113</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Concrete Template Signatures ($CTS$)</td>
<td>115</td>
</tr>
<tr>
<td>4.8</td>
<td>Abstract Templates ($AT$)</td>
<td>116</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Abstract Template Signatures ($ATS$)</td>
<td>117</td>
</tr>
<tr>
<td>4.9</td>
<td>Specification Adornment Environments ($SAE$)</td>
<td>118</td>
</tr>
<tr>
<td>4.9.1</td>
<td>Specification Adornment Variable Environments ($SAVE$)</td>
<td>121</td>
</tr>
<tr>
<td>4.9.2</td>
<td>Specification Adornment Environment Invariants ($SAEI$)</td>
<td>122</td>
</tr>
<tr>
<td>4.9.3</td>
<td>Specification Adornment Instance Environments ($SAIE$)</td>
<td>123</td>
</tr>
<tr>
<td>4.9.4</td>
<td>Specification Adornment Template Environments ($SAITE$)</td>
<td>123</td>
</tr>
<tr>
<td>4.9.5</td>
<td>Specification Adornment Templates ($SAT$)</td>
<td>124</td>
</tr>
<tr>
<td>4.9.6</td>
<td>Specification Adornment Template Signatures ($SATS$)</td>
<td>125</td>
</tr>
<tr>
<td>4.10</td>
<td>Interpretation Mappings ($IM$)</td>
<td>126</td>
</tr>
<tr>
<td>4.10.1</td>
<td>Type Interpretation Mappings ($TIM$)</td>
<td>129</td>
</tr>
<tr>
<td>4.10.2</td>
<td>Variable Interpretation Mappings ($VIM$)</td>
<td>131</td>
</tr>
<tr>
<td>4.10.3</td>
<td>Operation Interpretation Mappings ($OIM$)</td>
<td>131</td>
</tr>
<tr>
<td>4.10.4</td>
<td>Abstract Template Interpretation Mappings ($ATIM$)</td>
<td>133</td>
</tr>
<tr>
<td>4.10.5</td>
<td>Concrete Template Interpretation Mappings ($CTIM$)</td>
<td>135</td>
</tr>
<tr>
<td>4.11</td>
<td>Interpretation Mapping Templates ($IMT$)</td>
<td>136</td>
</tr>
<tr>
<td>4.11.1</td>
<td>Interpretation Mapping Template Signatures ($IMTS$)</td>
<td>138</td>
</tr>
<tr>
<td>4.12</td>
<td>Basis Environments ($BE$)</td>
<td>138</td>
</tr>
<tr>
<td>4.13</td>
<td>Operations on Subsystems</td>
<td>139</td>
</tr>
<tr>
<td>4.13.1</td>
<td>Abstracting an Environment</td>
<td>140</td>
</tr>
<tr>
<td>4.13.2</td>
<td>Reinterpreting an Environment</td>
<td>142</td>
</tr>
<tr>
<td>4.13.3</td>
<td>Cutting Down an Environment by an Interface</td>
<td>142</td>
</tr>
<tr>
<td>4.13.4</td>
<td>Derivation of Instances by Difference</td>
<td>144</td>
</tr>
<tr>
<td>4.13.5</td>
<td>Composing Interpretation Mappings</td>
<td>145</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The Four Kinds of Subsystems ............................... 34</td>
</tr>
<tr>
<td>2</td>
<td>Matrix of Viewpoints ....................................... 256</td>
</tr>
<tr>
<td>3</td>
<td>Matrix of Check List Items versus Previous Models ........ 297</td>
</tr>
<tr>
<td>4</td>
<td>Matrix of Check List Items versus Previous Models (continued) ... 298</td>
</tr>
<tr>
<td>5</td>
<td>Matrix of Check List Items versus Previous Models (continued) ... 299</td>
</tr>
</tbody>
</table>

xii
# List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The Temperature Controls on a Refrigerator</td>
</tr>
<tr>
<td>2</td>
<td>A Plausible Mental Model of Temperature Control</td>
</tr>
<tr>
<td>3</td>
<td>The Refrigerator's Actual Temperature Control Mechanism</td>
</tr>
<tr>
<td>4</td>
<td>The Partial.Map Facility Interface</td>
</tr>
<tr>
<td>5</td>
<td>The Partial.Map Facility Specification</td>
</tr>
<tr>
<td>6</td>
<td>The Partial.Map Facility Specification (continued)</td>
</tr>
<tr>
<td>7</td>
<td>The Meaning of the Partial.Map Facility Specification</td>
</tr>
<tr>
<td>8</td>
<td>The Model of the Type Partial.Map</td>
</tr>
<tr>
<td>9</td>
<td>The Model of the Partial.Map Operation Define</td>
</tr>
<tr>
<td>10</td>
<td>Subsystems Contain Types, Variables, and Operations</td>
</tr>
<tr>
<td>11</td>
<td>Reflecting the Context of Partial.Map.Facility</td>
</tr>
<tr>
<td>12</td>
<td>Subsystems Have Context</td>
</tr>
<tr>
<td>13</td>
<td>Subsystems Have Nested Structure</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>18</td>
<td>The Specification of Communal.PMap.Facility</td>
</tr>
<tr>
<td>19</td>
<td>The Specification of Communal.PMap.Facility (continued)</td>
</tr>
<tr>
<td>20</td>
<td>The Meaning of the Communal.PMap.Facility Specification</td>
</tr>
<tr>
<td>21</td>
<td>Subsystems Separate Specification Adornments from Program Code</td>
</tr>
<tr>
<td></td>
<td>Elements</td>
</tr>
<tr>
<td>22</td>
<td>A Generic Specification: Communal.PMap.Template</td>
</tr>
<tr>
<td>23</td>
<td>A Generic Specification: Communal.PMap.Template (continued)</td>
</tr>
<tr>
<td>24</td>
<td>The Meaning of Communal.PMap.Template</td>
</tr>
<tr>
<td>25</td>
<td>Templates Are Functions from Subsystems to Subsystems</td>
</tr>
<tr>
<td>26</td>
<td>The Details of an Abstract Instance</td>
</tr>
<tr>
<td>27</td>
<td><em>IM</em> Shows How to &quot;Interpret&quot; Specification A as Satisfying Speci-</td>
</tr>
<tr>
<td></td>
<td>fication B</td>
</tr>
<tr>
<td>28</td>
<td>A Subsystem S and the Outer Environment That Will Provide Its</td>
</tr>
<tr>
<td></td>
<td>Context</td>
</tr>
<tr>
<td>29</td>
<td><em>IM</em> Shows How OUTER Can Be Interpreted as the Context for S</td>
</tr>
<tr>
<td>30</td>
<td>Instantiating Communal.PMap.Template in RESOLVE</td>
</tr>
<tr>
<td>31</td>
<td>A More Detailed View of the Instantiation</td>
</tr>
<tr>
<td>32</td>
<td>Formal Notation for Subsystems</td>
</tr>
</tbody>
</table>

xiv
<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>An Initial Basis Environment $BE$</td>
<td>163</td>
</tr>
<tr>
<td>34</td>
<td>$BE$: A Simplified View</td>
<td>164</td>
</tr>
<tr>
<td>35</td>
<td>$BE$ After Instantiating Communal_PMap_Template</td>
<td>166</td>
</tr>
<tr>
<td>36</td>
<td>The Communal_PMap_Template Concept</td>
<td>168</td>
</tr>
<tr>
<td>37</td>
<td>The Communal_PMap_Template Concept (continued)</td>
<td>169</td>
</tr>
<tr>
<td>38</td>
<td>The Communal_PMap_Template Concept (continued)</td>
<td>170</td>
</tr>
<tr>
<td>39</td>
<td>The PARTIAL_FUNCTION_THEORY_TEMPLATE Mathematics</td>
<td>171</td>
</tr>
<tr>
<td>40</td>
<td>The PARTIAL_FUNCTION_THEORY_TEMPLATE Mathematics (continued)</td>
<td>172</td>
</tr>
<tr>
<td>41</td>
<td>PARTIAL_FUNCTION_THEORY_TEMPLATE is a Specification Adornment Template</td>
<td>173</td>
</tr>
<tr>
<td>42</td>
<td>PARTIAL_FUNCTION_THEORY_TEMPLATE's Parameter</td>
<td>173</td>
</tr>
<tr>
<td>43</td>
<td>PARTIAL_FUNCTION_THEORY_TEMPLATE's Resulting Specification Adornment Instance</td>
<td>174</td>
</tr>
<tr>
<td>44</td>
<td>Communal_PMap_Template is an Abstract Template</td>
<td>175</td>
</tr>
<tr>
<td>45</td>
<td>Communal_PMap_Template's Parameter</td>
<td>176</td>
</tr>
<tr>
<td>46</td>
<td>Communal_PMap_Template's Resulting Abstract Instance</td>
<td>177</td>
</tr>
<tr>
<td>47</td>
<td>Communal_PMap_Template's Resulting Abstract Instance (continued)</td>
<td>178</td>
</tr>
<tr>
<td>48</td>
<td>An Interpretation Mapping for Communal_PMap_Template's Parameter</td>
<td>180</td>
</tr>
<tr>
<td>49</td>
<td>The Unordered_Stack Realization Header</td>
<td>182</td>
</tr>
<tr>
<td>50</td>
<td>Unordered_Stack's Parameter</td>
<td>183</td>
</tr>
<tr>
<td>Page</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>51</td>
<td>The D.R.Pair.Facility Context-Bound Concrete Instance</td>
<td>185</td>
</tr>
<tr>
<td>52</td>
<td>The Stack.Facility Context-Bound Concrete Instance</td>
<td>185</td>
</tr>
<tr>
<td>53</td>
<td>The Unordered.Stack Realization Body</td>
<td>187</td>
</tr>
<tr>
<td>54</td>
<td>The Unordered.Stack Realization Body (continued)</td>
<td>188</td>
</tr>
<tr>
<td>55</td>
<td>The Unordered.Stack Realization Body (continued)</td>
<td>189</td>
</tr>
<tr>
<td>56</td>
<td>The Unordered.Stack Realization Body (continued)</td>
<td>190</td>
</tr>
<tr>
<td>57</td>
<td>The Unordered.Stack Realization Body (continued)</td>
<td>191</td>
</tr>
<tr>
<td>58</td>
<td>Unordered.Stack's Resulting Concrete Instance</td>
<td>192</td>
</tr>
<tr>
<td>59</td>
<td>An Interpretation Mapping for Unordered.Stack's Parameter</td>
<td>194</td>
</tr>
<tr>
<td>60</td>
<td>The UStack.To.CPMap Interpretation Mapping</td>
<td>197</td>
</tr>
<tr>
<td>61</td>
<td>The UStack.To.CPMap Interpretation Mapping (continued)</td>
<td>198</td>
</tr>
<tr>
<td>62</td>
<td>The Meaning of the Result of UStack.To.CPMap in ACTI</td>
<td>199</td>
</tr>
</tbody>
</table>
CHAPTER I

Introduction

In interacting with the environment, with others, and with the artifacts of technology, people form internal, mental models of themselves and of the things with which they are interacting. These models provide predictive and explanatory power for understanding the interaction. These statements hardly need be said, for they are consistent with all that we have learned about cognitive processes and, within [the book *Readings in Human-Computer Interaction*], represent the major underlying conceptual theme. Nonetheless, it does not hurt to repeat them and amplify them, for the scope of the implications of this view is larger than one might think.

— Donald Norman [49, p. 241]

As Donald Norman indicates, the human-computer interaction (HCI) community has benefited from exploring the implications of mental models. This perspective has aided their ability to create more usable, understandable user interfaces for computer programs, as exemplified by the popular “desk top” metaphor.

Researchers also have used the concept of mental models to examine how programmers design new software [23][64][14] and understand existing software [32][41]. The next logical step, as it was for the HCI community, is to ask how focusing on mental models can help us create software that is more usable and understandable to programmers. This is particularly important in the coming age of “software reuse,”
where software parts will be written with the goal of being utilized by different programmers in quite disparate contexts. In this dissertation, we turn our attention to the question of how to help create software that is more comprehensible, with interesting results.

In particular, asking questions about how programming languages support the formation of effective mental models leads one to conclude that conventional programming languages are *inadequate* for constructing large, sophisticated software systems that are "understandable." Furthermore, there is not a generally-accepted theory of what the building-blocks of software systems are, or how they are envisioned as contributing to the understandability of the software comprising them.

This dissertation explains the inadequacy of conventional programming languages in supporting the formation of effective mental models. It then presents a new model of component-based software that provides concrete support for recording critical information about each software structure, information that can form the basis for a programmer's own mental model of that structure. This new model, termed the ACTI model, for "Abstract and Concrete Templates and Instances," is both mathematically formal and language-independent. It captures and formalizes the underlying conceptual view of software architecture embedded in modern module-structured languages while simultaneously providing support for forming mental models. As a result, it can serve as a general-purpose theory of the nature of software building-blocks and their compositions.
1.1 Why Conventional Languages Fail

In short, modern programming languages have evolved from their predecessors with the primary purpose of describing instructions to computers. They were never designed to help explain to people the meaning of the software that they can describe. This has led to two critical problems with programming languages today: modules are considered to be purely syntactic constructs with no independent meaning, and those parts of programs that are deemed meaningful (usually procedures, in imperative languages) have a "hierarchically constructed" meaning.

First, most modern programming languages have some construct that is intended to be the primary "building-block" of complex programs. This building-block may be called a "module," a "package," a "structure," or a "class." Unfortunately, these constructs are rarely given meaningful semantic denotations. Conventional wisdom in the computer science field indicates that these constructs are primarily for grouping related definitions, controlling visibility, and enforcing information hiding. For example, when speaking of module-structured languages like Ada or Modula-2, Bertrand Meyer states:

In such languages, the module is purely a syntactic construct, used to group logically related program elements; but it is not itself a meaningful program element, such as a type, a variable or a procedure, with its own semantic denotation. [46, p. 61]

In this view, there is no way for one to make such building-blocks contribute directly to the understandability of the software comprising them. While object-oriented languages usually give a stronger meaning to the notion of a "class," they also fail
to provide any vision of how the meaning of individual classes can contribute to a broader understanding of the software systems in which they are embedded.

Second, those program elements that are given a real semantic denotation are often given a meaning that is "hierarchically constructed," or synthesized. In other words, the meaning of a particular program construct, say a procedure, is defined directly in terms of its implementation—a procedure "means" what the sequence of statements implementing it "means." The meaning of its implementation is defined in terms of the meanings of the lower-level procedures that it calls. Thus, a procedure's meaning is "constructed" from the meanings of the lower-level program units it depends on, and the meanings of those lower-level units in turn depend on how they are implemented, and so on.

This simple synthesis notion of how meaning is defined bottom-up is adequate from a purely technical perspective. It is also very effective when it comes to describing the semantics of layered programming constructs. Unfortunately, it is at odds with the way human beings form their mental representations of the meaning of software parts, as described in Chapter II.

The result of these two features of programming languages is that they are inadequate for effectively communicating the meaning of a software building-block to people (programmers, in particular). The semantic denotations of programming constructs in current languages only relate to how a program operates. They fail to capture what a program is intended to do at an abstract level, or why the given implementation exhibits that particular abstract behavior. In order to address these
concerns, it is necessary to assign meaning to software building-blocks, to separate the abstract description of a software part's intended behavior from its implementation, and to provide a mechanism for explaining why the implementation of the part achieves behavior consistent with that abstract description.

1.2 Correcting the Deficiency

The ACTI model is centered around the notion of a "software subsystem," a generalization of the idea of a module or a class that serves as the building-block from which software is constructed. A subsystem can vary in grain size from a single module up to a large scale generic architecture. ACTI is designed specifically to capture the larger meaning of a software subsystem in a way that contributes to human understanding, not just the information necessary to create a computer-based implementation of its behavior.

ACTI is not a programming language, however. It is a formal, theoretical model of the structure and meaning of software subsystems. It has two features that specifically address the inadequacies described in Section 1.1:

1. In ACTI, software subsystems (building-blocks) have an intrinsic meaning; they are not just syntactic constructs used for grouping declarations and controlling visibility. This meaning encompasses an abstract behavioral description of all the visible entities within a subsystem.
2. The meaning of a software subsystem is not, in general, hierarchically constructed. In fact, it is completely independent of all the alternative implementations of the subsystem.

Thus, ACTI provides a mechanism for describing what a subsystem does, not just how it is implemented. The meaning provided for a subsystem is a true abstraction—a "cover story" that describes behavior at a level appropriate for human understanding without explaining how the subsystem is implemented. Further, ACTI provides a formally defined mechanism, called an interpretation mapping, that captures the explanation of why an implementation of a subsystem will give rise to the more abstractly described behavior that comprises the meaning attributed to the subsystem—in short, an explanation for why the cover story works.

1.3 Contributions

The primary contribution of this research to the field of computer science is the development of the ACTI model. As a model of software structure, ACTI is unique because:

1. It promotes the creation of large, sophisticated software systems that are understandable. This is achieved by both supporting the formation of mental models of software parts, and addressing the inherent human (cognitive) limitations in dealing with those mental models.

2. It gives a real semantic denotation to "modules," which are the building-blocks from which software is composed. This denotation identifies just what "mod-
ules” should be so that they are meaningful, allow sophisticated composition, and present simple (not synthesized) conceptual models.

3. It identifies “interpretation mappings” as the mechanism that allows one to clearly represent why an abstraction (a specification with a simple model) correctly describes the composite behavior of a complex combination of lower-level software parts (an implementation).

4. It is language-independent, and unifies the concepts behind object-oriented programming and more traditional module-based programming. Thus, it solves the problem of understandable software composition across module- and class-based languages.

5. It solves the problems normally associated with “inheritance” [8] by separating “programming by difference” from type-to-type and module-to-module subtyping relationships.

Because of these achievements, ACTI can serve as a general-purpose theory of the nature of software building-blocks and their compositions.

1.4 Organization

Chapter II begins with a discussion of mental models and how programmers understand software. The topic of mental models leads to an explanation of the inadequacy of conventional programming languages, which provide no real aid to forming effective mental models of software parts. Chapter II concludes by explaining how ACTI
addresses this inadequacy. The discussion in Chapter II can be appreciated by anyone with a general background in computer science.

Next, Chapter III provides an explanatory introduction to the ACTI model of software subsystems. This introduction is presented at an informal, intuitive level, using an extended code example written in RESOLVE (Section A.3). Chapter III concentrates on what is contained in an ACTI subsystem, and examines a typical software component specification. While Chapter III is aimed at a general computer science audience, a basic familiarity with the RESOLVE language [66] will aid in getting the most out of the example.

Chapter IV then provides a complete formal definition of the ACTI model of software subsystems. The definition is centered around the four types of ACTI subsystems—abstract instances, concrete instances, abstract templates, and concrete templates—which can be used to provide semantics for module or class specifications, implementations, generics, and so on. ACTI is defined in formal mathematical terms, so a certain degree of mathematical maturity and familiarity with the concepts of denotational semantics is very helpful in obtaining a full appreciation of the model.

Given the formal definition of ACTI, Chapter V continues the example introduced in Chapter III to illustrate some of the more advanced aspects of ACTI. While Chapter III focuses on one subsystem, Chapter V looks more at the relationships between subsystems. It demonstrates the use of interpretation mappings between abstract and concrete instances, and explains the semantics of module-level program statements (such as instantiation of a parameterized component) in terms of the ACTI model.
In addition to familiarity with the formal definition presented in Chapter IV, the reader will need a good understanding of the RESOLVE language to fully appreciate Chapter V.

Chapter VI brings the main body of the dissertation to a close by summarizing the work, discussing the contributions, and pointing toward future research.

For the reader interested in the details of the development of the ACTI model, Appendix A discusses the research approach. Included is an examination of four modern programming languages chosen for the broad spectrum of software construction philosophies they embody: OBJ, RESOLVE, Eiffel, and Standard ML. Appendix A gives summaries of these languages and discusses the issues they raise for the software structure modeling problem.

Given this collection of languages, the next step is to compare and contrast the extent to which each addresses the intuitive notions about software architecture and composition that practitioners have observed. For example, the separation of specifications from implementations, the possibility of multiple implementations for one specification, and the use of generics for parameterized programming are all structuring techniques that practitioners have used in software systems. In order to perform a comparison based on such criteria, however, it is necessary to capture these "intuitive notions" in a more rigorous form. Appendix B addresses this problem.

Finally, Appendix C analyzes ACTI using the notions formalized in Appendix B to show how it measures up against the requirements derived from previous work. This appendix also shows how each of the example languages in Appendix A is "subsumed"
by the new language-independent ACTI model, or alternatively, how the structures in each of the languages can be interpreted within the ACTI mathematical spaces.

Taken as a whole, this dissertation follows the path that Norman advocates software professionals should follow, this time in the realm of software engineering rather than HCI:

People's mental models are apt to be deficient in a number of ways, perhaps including contradictory, erroneous, and unnecessary concepts. As designers, it is our duty to develop systems and instructional materials that aid users to develop more coherent, usable mental models. [49, p. 244]
CHAPTER II

Understanding Software: Mental Models

This chapter examines mental models and the question of how programmers understand software. This examination leads to an indictment of conventional programming languages for failing to provide real aids to the formation of effective mental models of software parts. Finally, the chapter concludes with an overview of how ACTI proposes to address this deficiency.

2.1 Mental Models

As pointed out by Norman [49, p. 241], people form internal mental models of the things they interact with. These models help a person to understand interactions with other people or things, in two ways [49, p. 241]:

1. A mental model allows one to predict the behavior of the person or thing with which the interaction takes place.

2. A mental model allows one to explain why the behavior arises.

Both of these benefits are important for a person to understand how to interact effectively with another person, a physical device, or a piece of complex software.
Hence, a mental model that is *effective* is one that provides sufficient predictive and explanatory power, and which a person can reasonably internalize and use to understand an interaction.

For the purposes of this dissertation, we are concerned with a person’s interactions with a software system. Following Norman’s terminology [49], *target system* will be used to refer to the software system with which a person is interacting. The *system image* is the entire visible “programmer interface” to the software component seen by a(No)ther software professional. It may include a system specification, complete source code, manuals and instructions accompanying the software, and even the way the software behaves and responds under operating conditions. While much of Norman’s work is related to the “user” interface that an entire system presents to an end user, here we are concerned with the “software” interface that a software subsystem presents to a programmer developing or maintaining a system.

Mental models evolve naturally through interaction with the target system [49, p. 241]. Over time, people reformulate, modify, and adapt their mental models whenever these models fail to provide reasonable predictive or explanatory power. For most purposes, the models need not be completely accurate, and usually they are not, but they must be functional. Norman documents the following general observations about mental models:

1. Mental models are incomplete.
2. People’s abilities to simulate or mentally execute their models are severely limited.
3. Mental models are unstable: People forget the details of the system they are using, especially when those details (or the whole system) have not been used for some period.

4. Mental models do not have firm boundaries: similar devices and operations get confused with one another.

5. Mental models are “unscientific”: People maintain “superstitious” behavior patterns even when they know they are unneeded because they cost little in physical effort and save mental effort.

6. Mental models are parsimonious: Often people do extra physical operations rather than the mental planning that would allow them to avoid those actions . . . .

[49, p. 241]

These observations indicate that mental models are inherently limited. These limitations stem from human cognitive limitations, a person’s previous experiences with similar systems, and even misleading system images [49, p. 241]. As Norman points out:

In making things visible [in the system image], it is important to make the correct things visible. Otherwise people form explanations for the things they can see, explanations that are likely to be false. . . . People are very good at forming explanations, at creating mental models. It is the designer's task to make sure that they form the correct interpretations, the correct mental models: the system image plays the key role. [50, p. 198]

As an example, Norman uses the temperature controls on an older refrigerator [50, pp. 14–17]. Figure 1 shows the basic temperature control layout on such machines. There are two dials located in the fresh food compartment of the refrigerator: one marked “Freezer,” and the other marked “Fresh Food.” Norman points out that this particular arrangement and labeling of the controls suggests a basic mental model of how the refrigerator should be operated.
Figure 1: The Temperature Controls on a Refrigerator

Figure 2 illustrates that model. Without any prior experience, an average person would likely assume that the temperatures in the two compartments of the refrigerator are independent, and each compartment's temperature is directly controlled by its corresponding dial. Such a mapping would be ideal; whenever the temperature needs to be changed in either compartment, it is perfectly clear exactly how to change it.
Figure 2: A Plausible Mental Model of Temperature Control

As Norman illustrates, however, refrigerators are not actually constructed that way. Because of the cost of the cooling subsystem, it is significantly more expensive for refrigerators to be built with two. As a result, refrigerators (especially older ones) use a single cooling unit to cool both compartments, as shown in Figure 3.
Unfortunately, the actual operation of such a machine is at odds with the “naive” mental model suggested by the temperature controls. Instead, the two controls are actually dependent on each other, one controlling the thermostat setting of the cooling unit, the other controlling a diverter regulating how the cooled air is split between the
two compartments. Even if one wants to change the temperature in one compartment of the refrigerator, it is not clear exactly how to go about it, because the controls are interdependent.

Fortunately, with modern technology and a little design forethought it is possible to remedy the situation. Instead of using the temperature dials on the refrigerator to directly control the cooler and diverter, the dials can be read by a low-cost microprocessor-based controller. This controller can then interpret the dial settings as desired temperature goals for the two refrigerator compartments, and modify the cooler thermostat and diverter settings accordingly. This is a nice solution all the way around: the person using the refrigerator can keep a simple, effective mental model; the manufacturer can keep costs down by using a single cooling unit; and the mapping between the person's mental model and the actual operation of the machine is made straightforward through the use of an intermediate microprocessor "interpreter."

Given this information, how can one support the formation and maintenance of effective mental models for complex software systems? Norman points out that the designer of the software already has a conceptual model of the system that is (presumably) accurate, consistent, and complete [49, p. 241][50, pp. 189-190]. The goal, in the best of all possible worlds, is to ensure that the system image is completely consistent with the conceptual model of the designer, and that from this system image the user forms a mental model consistent with the designer's conceptual model. Further, the system image should help alleviate the inherent human limitations of mental models. A well-designed system image can help to make the user's mental
model more complete, to record details so the user's model has firm boundaries, and to present those details in such a way that the user's model is more stable.

2.2 The Inadequacy of Conventional Languages

Section 2.1 indicates that to support understandable software, one must support the formation of effective mental models of that software, and help to overcome the natural limitations of human mental models. Unfortunately, hierarchically constructive representations of software as typified by common programming languages (e.g., Ada, C++, etc.) do not provide any such support.

To see why modern programming languages fail to provide adequate support for mental models, one can look at the typical system image presented by the source code. Modern programming languages typically provide two mechanisms for describing software artifacts: interfaces and implementations. Neither provides effective support for forming or maintaining effective mental models.

2.2.1 Software Interfaces Fail to Support Mental Models

In modern programming languages, software interfaces (Ada package specifications, C++ header files, and so on) only describe the syntactic features of a software artifact. This information is important for determining whether separate artifacts can be composed in a "grammatically" correct way, but tells one little about the meanings of the artifacts, either separately or in combination.

In terms of helping the "client" form an effective mental model of a software part, a syntactic interface merely provides names for the externally visible features of the
part, plus minor syntactic information about how these features may be combined. Norman argues persuasively that such “constraints” can be powerful aids in making devices easier to use [50, pp. 60–62, 81–86]. Unfortunately, this level of information is only sufficient for simple interfaces: the shape of Lego blocks constrains how they can be combined, for example. In general, syntactic constraints are useful when there are few useful action sequences, but a very large number of primitive actions to choose from—constraints drastically reduce the number of reasonable choices. If the user is unsure or “forgets” what action comes next, constraints help point the way.

For complex software components, however, there are often no limits on the number of useful action sequences (e.g., sequences of procedure calls exported by a subsystem). Syntactic constraints help a person to understand which ones are well-formed, but give no guidance in understanding how to select among alternatives, or how to infer the meaning of any particular sequence. What the interface lacks is visibility of the meaning and functionality provided by each type, operation, or exported entity.

Norman gives a useful example of a telephone which is very difficult to operate [50, pp. 17–23]. While there are clear constraints and affordances showing how to operate the controls of the phone (e.g., buttons invite one to press them), in empirical studies people could not easily make use of the majority of the phone’s advanced features, such as conference calling, automatic call-back, and even the “hold” capability. Because the behavior of the phone was so much more complex than its simple control interface, nothing about the controls themselves made the advanced capabilities of the phone
visible. Where complex functionality is hidden by a much simpler syntactic interface, there is apt to be trouble [50, p. 22].

As a result, syntactic constraints are insufficient for interfaces that require complex interactions, which is exactly the case for software parts. Syntactic interface descriptions also fail to alleviate any of the shortcomings of human mental models, unless visibility of the meaning and functionality of a software part is exposed by other means.

2.2.2 Software Implementations Fail to Support Mental Models

If a software component’s interface description does not support forming an effective mental model, perhaps the component’s implementation will. Since an implementation describes the actual inner workings of a software part, however, one could only argue that it supports an effective mental model if the inner mechanism itself were an effective mental model. If this were normally the case, abstraction would be unnecessary.

Alternatively, one might argue that the structure of the implementation “suggests” an abstract view of that implementation. In all modern software languages, however, implementations are constructive representations of the inner workings of the corresponding software part. A constructive representation is one in which a complex software artifact is directly represented as a composition or interconnection of lower-level pieces. For an implementation to suggest a useful mental model, the programmer would have to possess mental models for the lower-level pieces from which
the implementation is constructed. Mentally composing these would then have to suggest a mental model for the implementation.

Unfortunately, the inherent limitations of mental models described in Section 2.1 make it difficult for people to perform such acts reliably. Indeed, empirical studies of the breakdowns that software professionals experience during design activities indicate exactly this: merging mental models for subsolutions into a coherent model of a complete solution is particularly troublesome [23, p. 77]. This is due in part to the growing complexity of such a mental construct and the attendantly taxing cognitive burden of simulating or evaluating it, as systems get even moderately large.

Together with Norman’s observations that mental models tend to be incomplete, lack firm boundaries, and are unstable, this evidence strongly suggests that “mental composition” of mental models in a hierarchical fashion will exacerbate their inherent limitations. As a result, implementations simply do not provide any reasonable support for forming effective mental models of complex software subsystems.

2.2.3 Why Comments Alone are not Enough

When trying to provide simple mental models for software components, the question of the effectiveness of comments arises. Thorough commenting of a component interface can allow a software designer to document the conceptual model of the part and pass this model on to other professionals using that part. In addition to the possibility of informal written descriptions of conceptual models, comments also allow the inclusion of formally defined behavioral specifications. Entire specification languages, such as
ANNA [43], have been designed to support complete or partial formal specifications via structured comments.

Unfortunately, this is really only enough to capture (most often, informally) the external behavior advertized in a component’s specification. It is analogous to the user’s mental model of the refrigerator pictured in Figure 1. To truly embrace the perspective instigated by mental models, one must also provide support for forming mental models of the implementation of a software component, how that implementation works, and why it gives rise to the abstract description in its specification. Software implementations typically rely on more information to operate correctly than is represented in the source code alone in a conventional language [40, pp 70–74][2, pp. 47–48]. Consider an abstract data type, where an implementation level “representation invariant” or its equivalent is proposed to capture some of this information. This, in addition to a model of the representation of the exported data type and behavioral descriptions of all internally declared operations, makes up a maintainer’s mental model of a component implementation. In the refrigerator analogy, this would be an abstract view of how the refrigerator actually operates, as shown in Figure 2 (including a microprocessor, if present).

Further, however, one must also account for the maintainer’s mental model of why that implementation conforms to its specification—the basis for explaining why the behavior of the inner, more detailed mental model can be interpreted as being consistent with that predicted from the specification’s more abstractly described cover story. This information is usually associated with an “abstraction function,” “ab-
stract relation,” or “type correspondence” that explains the relationship between the representation of a given data type and its abstract specification. Similar information must be recorded about the relationship between module-level state variables in the implementation and those visible in the specification. For a refrigerator, the analogy here would be an explanation of why the implementation meets the abstract model of how the controls operate—e.g., an explanation of the way in which the microprocessor translates dial settings to cooler and diverter settings.

Certainly it is possible to lay out a comprehensive plan for how to document all of this information via structured, formal or informal comments. Once all of this information is captured in the software component itself, it is even possible to start asking questions about whether the component is being used correctly:

- Does this implementation actually conform to that specification?
- From the client’s point of view, is she using the given component “correctly” (i.e., in accordance with its professed abstract model)?
- Will a proposed maintenance change introduce new defects by violating the expectations of the remainder of the implementation, as expressed in the representation invariant(s)? Will it invalidate the reasons why the implementation currently conforms to the specification?

Unfortunately, to lay out this comprehensive commenting plan, one must go to a great deal of trouble to provide a firm foundation for it. Indeed, one must understand all of the information that must be represented, and have a firm grasp of the theoretical
underpinnings of how valid answers to questions like those above are generated. In doing this, the researcher steps far beyond the idea of “simply commenting the code,” and runs head on into the problems addressed (formally) in the remainder of this dissertation.

2.2.4 Conventional Programming Languages Do Not Support Mental Models

Because neither software interfaces nor bare implementations provide support for forming effective mental models or for overcoming the natural limitations of these models, one must conclude that modern programming languages alone are inadequate for constructing large, complex software systems that are understandable. If one considers supplementary documentation, manuals, design diagrams, and so on as part of the system image, then perhaps it is still possible to provide a clear, consistent conceptual model for the user to absorb. But note that much of this conceptual model is absent from the source code itself. Chapter IV presents a model of software that includes this necessary information and that does support understandability even if the source code is the sole provider of the system image. A significant advantage of this approach is that a uniform semantics for the source language makes it possible to treat the kinds of questions in Section 2.2.3 in a formal way.

Also, the discussion in Section 2.2.2 leads one to an important observation: People cannot easily juggle and compose mental models of components to form new models of a whole. As a result, any proposed model of software that purports to support understandability must provide for each software building-block to be meaningful on
its own, possessing a clear conceptual model people can readily assimilate in isolation. This same capability must also be provided for any combination of building-blocks (e.g., a subsystem or system); each must be meaningful on its own, independent from any of the lower-level pieces of which it is composed. This is an important feature of ACTI.

This suggests the need for new "syntactic slots" in programming languages, where information that is known to be helpful in conveying mental models can be provided as part of the software itself. In addition to providing a way to express this information, such language devices also suggest to software designers and documenters that such information is needed.

2.3 ACTI Addresses the Inadequacy

ACTI, which stands for "Abstract and Concrete Templates and Instances," is a mathematically formal, language-independent model of software that is described in Chapter IV. It allows a software part, either a specification or implementation, to be given a useful, abstract meaning completely independent of any other software part. This meaning can be a direct representation of the designer's conceptual model or abstract view of the software part, and as such can form a concrete basis for the user's mental model. In addition, this meaning is not constructive in how it is represented; it is not described in terms of the lower-level building blocks from which the software part is constructed.
ACTI allows relational "computation" over arbitrary mathematical domains. This means that the abstract meanings of software parts can be expressed in the simplest conceptual terms, and the complete power of mathematics is available for describing the most accessible conceptual models.

The meaning of a software part in ACTI is an abstract model of the part that completely captures all of its relevant behavioral characteristics. As such, it conveys much more information to the client than a syntactic interface description alone.

Because each software part has an explicit meaning associated with it, and this meaning is (by design) also the mental model that the software writer intends for the client to absorb, ACTI provides direct support for mental model formation.

Because the meaning of each software part is described right along with the part itself, the client always has a complete, stable representation of the intended mental model for the part, and this representation is recorded so she can always refer to it when in doubt. This provides active support for addressing the natural limitations of mental models.

Of course, one cannot claim that the conceptual model chosen by the designer is one that clients will relate to and internalize easily. Just as specification tools merely allow one to write a specification, but in no way ensure that a "good" one will be produced, ACTI allows one to express a conceptual model as an important aspect of a software part, without ensuring that the designer will choose a good one.

This is in contrast to conventional programming languages, which do not even support effective mental models; even if the designer has a conceptual model he thinks
is good, there is no "official" place for it to be written down. Conventional languages just have no way to represent such information, and it must be relegated to supplementary materials (documentation, manuals, comments, etc.). As a result, ACTI supports the designer by giving a place to provide a complete, unambiguous, stable description of the intended mental model. Similarly, it supports the programmer by providing this intended model for reference, both when the client is first trying to understand the component, and later when support is needed in overcoming his or her own mental model's natural limitations. While this does not ensure an easy-to-assimilate mental model will be provided by the component's designer, to the extent there is one, support is offered in ways that do not exist in conventional languages.
This chapter provides an explanatory introduction to the ACTI model of software subsystems. The basic concepts of ACTI are introduced at an informal, intuitive level, using an extended code example written in RESOLVE (Section A.3). While this chapter is aimed at a general computer science audience, a basic familiarity with the RESOLVE language [66] will aid in getting the most out of the example.

3.1 General Guiding Characteristics

In developing ACTI, there were four overriding concerns:

1. Supporting the formation of effective mental models of software parts, and addressing the natural limitations of such mental models.

2. Defining ACTI in mathematical terms, rather than simply developing a new language syntax.

3. Maintaining independence from any particular programming language.
4. Ensuring ACTI is comprehensive, in that it captures the underlying conceptual view of software architecture embedded in modern module-structured languages. Refer to Appendix B for a description of how these intuitive notions were captured.

While simultaneously achieving all four of these goals was challenging, the goals themselves provided the rationale for the three key decisions that shaped ACTI: basing it on denotational semantics; embracing the phase distinction between compilation and execution; and giving subsystems real semantic denotations.

First, ACTI is based on denotational semantics. Because one of the goals for ACTI is achieving a model of software structure that is independent of any specific programming language, the question of how one can present such a construction without reference to a particular program syntax naturally arises. This same problem also exists in the study of programming language semantics, where the relationship between a program's syntactic form and its abstract meaning is of central concern. In fact, one approach to programming language semantics, the denotational approach, is uniquely suited for the task at hand. Robinson briefly describes the core approach used in denotational semantics as follows:

In the denotational philosophy inspired by Strachey the program, or program fragment, is first given a semantics as an element of some abstract mathematical object, generally a partially ordered set, the semantics of the program being a function of the semantics of its constituent parts; properties of the program are then deduced from a study of the mathematical object in which the semantics lives. [55, p. 238]
By pursuing this philosophy, one is naturally led to consider the various mathematical objects which make up the "semantic world" of a particular programming language. This dissertation uses this approach and extends it one step farther, looking for the mathematical spaces which make up a "semantic world" broad enough that it encompasses the notions of software architecture, module structuring, and parameterized programming that underlie modern programming languages. Further, to support understandable software, these spaces must also encompass the notions necessary to support mental models.

Thus, the entities comprising ACTI are intended to represent the meanings of program structures such as modules, implementations, and specifications. These meanings are described and presented via mathematical notation rather than a particular programming language notation. This is a direct consequence of goals (2) and (3) above.

Second, ACTI was constructed with respect for a strict separation between the two distinct phases observed by programming languages: the static phase of determining whether programs or program fragments are well-typed, well-formed, and meaningful (usually the compilation phase); and the dynamic phase of evaluation or execution of programs or program fragments that have passed through the static phase (usually the execution phase). This phase distinction [48, p. 1] is present in all programming languages, although no two languages seem to agree on exactly where to draw the boundary. Nevertheless, observance of the phase distinction during language design can play a critical role in achieving efficient execution for a typed language [47, p. 26].
Further, the phase distinction is important from the point of view of maintaining language independence. The static phase is concerned expressly with determining whether a purported program is well-typed, well-formed, and meaningful, which is clearly a language-specific task. As a result, ACTI was developed to model programming concepts that appear in the execution phase, rather than the compilation phase: Given that a particular module definition is well-formed, how can its meaning be modeled?

Third, ACTI is built on the idea that “modules” and other program structuring constructs have true semantic denotations during the execution phase. This is in direct support of goal (1). This is contrary to some languages with module-like structuring mechanisms, such as Ada or Modula-2, which consider them to be purely syntactic grouping constructs.

The languages described in Appendix A vary on this point. RESOLVE, for example, does not include concept or realization structures in its denotational semantics. Standard ML does include structure meanings in its dynamic semantics, but the role of signatures (module interfaces) is almost entirely restricted to the static semantics. Eiffel includes module-like information in its semantics only because it identifies the notion of module with the notion of type [46, p. 61]. OBJ is the only language of the four to include useful semantic denotations of both module implementations and specifications completely in the dynamic phase of its semantics [21, p. 172–182].

As of the publication of this research. There is, however, a notion of “module validity” in the language’s proof system. For more information on RESOLVE’s denotational semantics, see [29]. Also of interest are [33], [12], [10], [13], and [11].
is the author's opinion that this is a direct result of the fact that OBJ allows module
specifications to describe behavior, in addition to syntax, and that the relationships
between specifications and implementations (OBJ views) are distinct, named entities
in the language.

Further, the detailed requirements for ACTI presented in Appendix B include
allowing specifications to describe the abstract behavior of subsystems. They also
advocate incorporating checkable specification-to-specification and specification-to-
implementation relationships in a model. As a result, it makes sense to search for
useful semantic denotations for module-like structures in ACTI, even if it is restricted
to capturing meaning during the execution phase.

In addition, if ACTI is to achieve goal (4), it must necessarily include the notion
of reusable software subsystems, where a subsystem can vary in grain size from a
single module up to a large scale generic architecture. As a result, ACTI is built
around the idea of a software subsystem as the primary building-block. In effect, a
subsystem is a generalized program module that may encompass an arbitrarily com-
plex interconnection of smaller subsystems, packaging them up as a single, composite
building-block.

The four goals presented here, and the three major decisions they led to, have
profoundly affected the nature of ACTI. The remainder of this chapter describes the
result of following them to their logical conclusion, searching for a model that is as
simple as possible, but which still reflects the sophistication and capabilities of current
programming languages.
3.2 Deriving ACTI

As described in Section 3.1, an early decision was made to use the techniques of denotational semantics for expressing the new model. Thus, the process for developing ACTI began with a careful study of the formal definition of Standard ML’s semantics, described by Milner et al. [48], along with the literature on RESOLVE semantics and verification. Some of RESOLVE’s denotational semantics are described by Wayne Heym [29], and fragments of the techniques used therein appear elsewhere [12, 10, 13, 11]. A natural outcome of this is that the resulting model shares many similarities with the both of these sources. ACTI was then refined to gradually incorporate all of the properties described in the check list in Appendix B.

3.3 Abstract and Concrete Templates and Instances

The ACTI model is centered around four different classes of subsystems:

1. **Abstract** subsystems, or subsystem specifications that are implementation-neutral and that describe the structural and behavioral properties of a subsystem interface.

2. **Concrete** subsystems, or subsystem implementations that provide representations or realizations of their features.

3. **Template** subsystems, or generic subsystems, which are parameterized and can be used to generate any member of an entire class of related subsystems.

4. **Instance** subsystems, which are not parameterized like templates.
These terms are all taken from Weide et al. [51, p. 23], and the same ideas appear in the 3C model [62]. The name “ACTI” is an acronym derived from these four terms: “Abstract and Concrete Templates and Instances.”

<table>
<thead>
<tr>
<th>Subsystem Varieties</th>
<th>Abstract (Specification)</th>
<th>Concrete (Implementation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template</td>
<td>RESOLVE concept</td>
<td>RESOLVE realization</td>
</tr>
<tr>
<td></td>
<td>SML functor signature</td>
<td>SML functor</td>
</tr>
<tr>
<td>Instance</td>
<td>Ada package spec.</td>
<td>Ada package body</td>
</tr>
<tr>
<td></td>
<td>SML signature</td>
<td>SML structure</td>
</tr>
<tr>
<td></td>
<td>Eiffel class interface</td>
<td>Eiffel class impl.</td>
</tr>
</tbody>
</table>

This view of the world allows software subsystems to be partitioned along two orthogonal dimensions, as shown in Table 1. The distinction between “abstract” and “concrete” embodies the separation between a specification or interface, and an implementation or representation. The distinction between “template” and “instance” allows one to talk about both generic modules, and the product of fixing (binding) the parameters of such a generic module: an instance module.

Table 1 gives examples of some programming language structures that might typify each of the four kinds of subsystems. The intuition behind these subsystem varieties are:
Abstract Instance—A disembodied subsystem specification or interface description. There is no implementation associated with anything defined in the specification.

Concrete Instance—A subsystem that provides implementations for its types and operations. All of the defined types and operations in the subsystem are represented and/or implemented.

Abstract Template—A subsystem-to-subsystem function that, when applied to its argument, which is some abstract instance, will generate another abstract instance. Effectively, an abstract template is a form of generic subsystem specification.

Concrete Template—A subsystem-to-subsystem function that, when applied to its argument, which is some concrete instance, will generate another concrete instance. Thus, a concrete template is a form of generic subsystem implementation.

While there are many details in the ACTI definition presented in this chapter, a strong intuitive grasp of these distinctions gives one an effective understanding of the heart of ACTI.

3.4 An Intuitive View of Subsystems

Before seeing the formal definition of the mathematical spaces comprising ACTI, it helps to see a simplified, intuitive picture of what the model captures. The deno-
tional approach to programming language semantics is centered around defining a semantic function $S$ from programs to their meanings, which are functions from states to states:

$$ S : Program \rightarrow (State \rightarrow State) $$

(3.1)

It is typical to consider a $State$ to simply be a mapping from names to values (e.g., variable names to their corresponding values) [52, p. 174].

Subsystems, or more properly, abstract and concrete instances, provide a structuring mechanism for organizing the name-to-value mapping in a $State$. In a programming language with support for subsystems, subsystem-level statements, such as subsystem declarations, instantiations, or compositions, manipulate this "structured" state space at a higher level than assignment statements. Thus, a subsystem can be viewed as a "bundle" of name-to-value associations within the overall program state, a bundle that can be manipulated as a whole.

In order to explore exactly how subsystems can be viewed as structured collections of name-to-value mappings, this section introduces an example software component specification written in RESOLVE (Section A.3, p. 232). The different aspects of an ACTI subsystem are then incrementally presented using this concrete code example.

### 3.4.1 The Partial.Map.Facility Example

The example we use in this chapter is a component defining a "partial map." Partial map is another name for a data structure known variously as an associative map, a dictionary, or a symbol table. It is used to store associations between pairs of key values and data values. Partial maps get their name from the fact that they are
facility Partial_Map_Facility

context
  global context
    facility Standard_Character_String_Facility
    facility Standard_Integer_Facility
    facility Standard_Boolean_Facility

interface

type Partial_Map

operation Define ( alters m : Partial_Map
  consumes d : Character_String
  consumes r : Integer
  )

operation Undefine ( alters m : Partial_Map
  preserves d : Character_String
  produces d.copy : Character_String
  produces r : Integer
  )

operation Und6fine_Any ( alters m : Partial_Map
  preserves d : Character_String
  produces r : Integer
  )

operation Is-Defined ( preserves m : Partial_Map
  preserves d : Character_String
  ) : Boolean

operation Size ( preserves m : Partial_Map
  ) : Integer

end Partial_Map_Facility

Figure 4: The Partial_Map_Facility Interface
mappings, or functions, from key values to data values—given a particular key, the partial map can be used to look up the associated data value. Such structures are "partial" because they do not record a data value for every possible key, so like a partial function, they only map a limited set of keys to data values.

Figure 4 shows the syntactic interface to a program unit named Partial_Map_Facility that exports a type Partial_Map. This program unit is written in a variant of RESOLVE\(^2\), and is adapted from a generic version with fully specified behavior [9, pp. 31–32]. For the reader interested in how a RESOLVE implementation for this unit would look, one is available [2, pp. 42–45].

The Partial_Map type provided by the Partial_Map_Facility records associations from Character_String keys to Integer data values. Thus, a partial map object of this type is like a partial function from character strings to integers, recording a particular collection of associations. These types have been chosen for simplicity in this example; any two types could conceivably be used here.

A RESOLVE facility is simply a stand-alone module, much like an Ada package. The syntactic interface to the Partial_Map_Facility shown in Figure 4 gives approximately the same kind and amount of information one would see in an Ada package specification, a C++ class header, or a Standard ML signature. As described in Section 2.2, this is an inadequate basis from which to form a mental model.

Figures 5 and 6 show a complete specification of Partial_Map_Facility, including a formal description of the behavior of each operation. This specification is model-

\(^2\)The only changes to the RESOLVE syntax are the addition of the facility keyword so that a non-generic component can be described, and the omission of formal specifications in Figure 4.
facility Partial.Map.Facility

context
global context

| facility Standard.Character.String.Facility |
| facility Standard.Integer.Facility          |
| facility Standard.Boolean.Facility         |

(c)

interface

type Partial.Map is modeled by

set of (
  d : math[Character.String]
  r : math[Integer]
)

exemplar m

constraint

for all d : math[Character.String],
  r1, r2 : math[Integer]
  where ((d, r1) is in m and
     (d, r2) is in m)
  (r1 = r2)

(a)

initialization

ensures m = empty_set

operation Define ( __________________________________

alters m : Partial.Map

consumes d : Character.String

consumes r : Integer

) (b)

requires not there exists r : math[Integer]
  ((d, r) is in m)

ensures m = #m union {(#d, #r)}

Figure 5: The Partial.Map.Facility Specification
operation Undefine (  
    alters m : Partial_Map  
    preserves d : Character_String  
    produces d_copy : Character_String  
    produces r : Integer  
  )  
  requires there exists r : math[Integer]  
                   ((d, r) is in m)  
  ensures (d, r) is in #m and  
            m = #m - {(d, r)} and  
            d_copy = d  

operation Undefine_Any_One (  
    alters m : Partial_Map  
    preserves d : Character_String  
    produces r : Integer  
  )  
  requires m ≠ empty_set  
  ensures (d, r) is in #m and  
            m = #m - {(d, r)}  

operation Is_DEFINED (  
    preserves m : Partial_Map  
    preserves d : Character_String  
  ) : Boolean  
  ensures Is_DEFINED iff  
        there exists r : math[Integer]  
        ((d, r) is in m)  

operation Size (  
    preserves m : Partial_Map  
  ) : Integer  
  ensures Size = |m|  

end Partial_Map_Facility

Figure 6: The Partial_Map_Facility Specification (continued)
based [9, p. 29–30], in that the behavior of each operation is described in terms of a mathematical model of the types of values involved. In this case, a Partial.Map object is modeled as a set of ordered key-value pairs, as shown in box (a) of Figure 5. The behavior of each operation is described with respect to this model, possibly adding, removing, or otherwise manipulating a specific key-value pair within the set held in a particular partial map object, as shown in box (b) for the operation Define that adds an association.

While the mathematical model of the type Partial.Map allows the designer to precisely describe the intended behavior of each operation in this component specification, it is also a useful basis for any client programmer's mental model. By conceiving of a partial map as simply a set of ordered pairs representing key-value associations, any programmer reading this specification has enough information to make effective use of any component that conforms to it. More importantly, this "cover story" insulates such a programmer from the actual implementation details involved. Whether the type Partial.Map is implemented using a hash table, a binary search tree, a linear list, or any other mechanism, the client programmer can understand and reason about the resulting behavior using only the abstract cover story.

3.4.2 Subsystems Contain Types, Variables, and Operations

The Partial.Map.Facility introduced in Section 3.4.1 is what programmers would normally consider a "module specification." It provides an abstract description of the program elements exported by a software part, including their behavior.
In ACTI's world of subsystems, the denotation of this facility specification is an abstract instance. That is, it is a subsystem specification that does not contain any implementation information. Because software professionals are so accustomed to the idea of separating a component's specification from its implementation, the running example in the remainder of this chapter will focus on abstract instances. Once the reader has a comfortable understanding of abstract instances, it is easier to generalize to the other kinds of subsystems in ACTI.

Intuitively, one would expect a subsystem to encompass name-to-value associations for types, variables, and operations, since one normally finds those kinds of definitions within the traditional module or package constructs of conventional languages. For an abstract instance subsystem, the value associated with each name is a mathematical model of the properties of the corresponding programming element. Figure 7 schematically presents the meaning of the Partial.Map.Facility shown in Figures 5 and 6.

In Figure 7, Partial.Map.Facility is split into three separate name-to-value mappings: one for mapping type names to type models, one for mapping variable names to variable models, and one for mapping operation names to operation models. Since there are no program variables declared in Partial.Map.Facility, its variable name-to-value mapping is empty.

The type name-to-value mapping contains a single name, for the type Partial.Map. This type's name is associated with (Model of Partial.Map), which is depicted informally in Figure 8. Formally, the model of the type Partial.Map is described
Partial_Map_Facility $\rightarrow$ **Exported Behavior**

<table>
<thead>
<tr>
<th>Types</th>
<th>{ Partial_Map $\rightarrow$ \langle Model of Partial_Map $\rangle$ }</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>{}</td>
</tr>
<tr>
<td>Operations</td>
<td>{ Define $\rightarrow$ \langle Model of Define $\rangle$, Undefine $\rightarrow$ \ldots, Undefine.Any_Hue $\rightarrow$ \ldots, Is_Defined $\rightarrow$ \ldots, Size $\rightarrow$ \ldots, Initialize $\rightarrow$ \ldots, Finalize $\rightarrow$ \ldots, Swap $\rightarrow$ \ldots }</td>
</tr>
</tbody>
</table>

Figure 7: The Meaning of the Partial_Map_Facility Specification
\[ \langle \text{Model of Partial.Map} \rangle \equiv \left( \begin{array}{c} \text{Domain} \\
\text{of} \\
\text{Values} \\

\text{Constraint} : \\
\end{array} \right) \left( \begin{array}{c} \text{The CPO of all sets of} \\
\text{ordered pairs of the form} \\
\text{math[Character.String],} \\
\text{math[Integer]}. \\

\text{Only sets containing ordered} \\
\text{pairs with unique values in} \\
\text{their first component are} \\
\text{allowable.} \\
\end{array} \right) \]

Figure 8: The Model of the Type Partial.Map

in Figure 5 as box (a). This type model consists of two parts: the mathematical domain from which the values of the type come, and a constraint over that domain restricting which values are acceptable. Informally, the second part of the type model, the constraint, is often referred to as a “type invariant.”

The operation name-to-value mapping in Figure 7 contains eight entries, one for each of the operations provided by Partial.Map.Facility. This includes the three operations RESOLVE implicitly declares with each type declaration: Initialize, Finalize, and Swap [9]. For simplicity, only the model of the Define operation is shown in Figure 7, although each of the operations has a similar value associated with its name. Figure 9 informally presents the \( \langle \text{Model of Define} \rangle \), which corresponds to the specification assertions shown in box (b) of Figure 5.

The model of an operation consists of three parts. The first is the parameter profile, which records the types and order of the parameters to the operation. The second is an abstract representation of the operation’s precondition, indicating what argument values are allowable when invoking the operation. The third is an abstract
representation of the operation's postcondition, which can be thought of as a relation between the input arguments and the output arguments.

In the abstract, then, an abstract instance can contain types, variables, and operations, all of which are associated with abstract models. This is shown in Figure 10, where an abstract instance $AI_1$ is presented as a collection of name-to-value mappings for each of these kinds of entities.

### 3.4.3 Subsystems Have Context

In Figure 5, Partial\_Map\_Facility is not defined in isolation. It has a global context section, where it imports three other facilities from RESOLVE's standard library. In this case, these imported facilities merely specify programming types one would normally consider built-in to a language: Character\_String, Integer, and Boolean. Even in this simple case, however, the global context of the partial map facility shows that subsystems are not defined in isolation, but within a specific context or environ-
Abstract Instance

\[ AI_1 \rightarrow \]

Exported Behavior

\[
\begin{align*}
\text{Types} & : \begin{cases} 
T_1 & \mapsto \langle \text{Model of } T_1 \rangle \\
T_2 & \mapsto \langle \text{Model of } T_2 \rangle \\
\vdots & \mapsto \ldots
\end{cases} \\
\text{Variables} & : \begin{cases} 
V_1 & \mapsto \langle \text{Model of } V_1 \rangle \\
V_2 & \mapsto \langle \text{Model of } V_2 \rangle \\
\vdots & \mapsto \ldots
\end{cases} \\
\text{Operations} & : \begin{cases} 
O_1 & \mapsto \langle \text{Model of } O_1 \rangle \\
O_2 & \mapsto \langle \text{Model of } O_2 \rangle \\
\vdots & \mapsto \ldots
\end{cases}
\end{align*}
\]

Figure 10: Subsystems Contain Types, Variables, and Operations

ment (see Section A.1 for a complete discussion of the terms concept, content, and context used in the 3C Model).

The “context” associated with a subsystem, such as Partial.Map.Facility, represents all of the expectations that subsystem has of its environment, and thus captures all of its external dependencies. This context includes all such dependencies on types, variables, operations, or other subsystems outside of Partial.Map.Facility.

As a result, every ACTI abstract instance contains an explicit interface to its context that completely expresses all of the instance’s expectations and assumptions about its environment. This context interface, which is roughly analogous to fixed context [9, p. 29] (as opposed to parametric context), is expressed as simply another
### Context

<table>
<thead>
<tr>
<th>Concrete Instances</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char.Str.Fac</td>
<td>Types: {Character.String $\mapsto \ldots$}</td>
</tr>
<tr>
<td>Int.Fac</td>
<td>Types: {Integer $\mapsto \ldots$}</td>
</tr>
<tr>
<td>Bool.Fac</td>
<td>Types: {Boolean $\mapsto \ldots$}</td>
</tr>
</tbody>
</table>

### Exported Behavior

- **Types**: {Partial.Map $\mapsto \langle Model\ of\ Partial\ .Map \rangle$}
- **Variables**: {}
- **Operations**:
  - Define $\mapsto \langle Model\ of\ Define \rangle$
  - Undefined $\mapsto \ldots$
  - Undefined.Any.One $\mapsto \ldots$
  - Is_DEFINED $\mapsto \ldots$
  - Size $\mapsto \ldots$
  - Initialize $\mapsto \ldots$
  - Finalize $\mapsto \ldots$
  - Swap $\mapsto \ldots$
- **Invariant**: true

---

**Figure 11**: Reflecting the Context of `Partial.Map_Facility`
abstract instance—this time describing the component’s interface to its environment, instead of the client programmer’s interface to the component itself.

Figure 11 schematically shows the meaning of Partial_Map_Facility, including its context interface. Partial_Map_Facility’s global context description in box (c) of Figure 5 indicates that it depends on three external specifications: Standard_Character_String_Facility, Standard_Integer_Facility, and Standard_Boolean_Facility. The Context section in Figure 11 shows that the Partial_Map_Facility abstract instance expects its environment to provide three subsystem implementations, here abbreviated to Char_Str_Fac, Int_Fac, and Bool_Fac, respectively. The names that appear in the context of this abstract instance are intentionally different from those of the standard RESOLVE library units. This is because in ACTI, a subsystem’s context describes requirements that subsystem makes on its environment; the context does not state how those requirements are filled—i.e., which actual units in the environment will be chosen to meet the requirements.

Deciding which external subsystems will be used to meet Partial_Map_Facility’s context requirements is called binding its context. In RESOLVE, the global context description in box (c) of Figure 5 both describes the fixed context of Partial_Map_Facility and binds it to specific RESOLVE facilities in the programming environment. For clarity, ACTI distinguishes these two aspects.

Figure 11 also shows that an abstract instance has an invariant. Just as types have invariants that capture properties shared by all objects of that type, abstract instances have invariants that capture properties shared by all subsystems conforming to that
Abstract Instance

\[ AI_1 \] \[ \quad \longrightarrow \quad \]

<table>
<thead>
<tr>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exported Behavior</strong></td>
</tr>
<tr>
<td><strong>Types</strong> : ( { T_1 \mapsto \langle \text{Model of } T_1 \rangle, T_2 \mapsto \langle \text{Model of } T_2 \rangle, \ldots } )</td>
</tr>
<tr>
<td><strong>Variables</strong> : ( { V_1 \mapsto \langle \text{Model of } V_1 \rangle, V_2 \mapsto \langle \text{Model of } V_2 \rangle, \ldots } )</td>
</tr>
<tr>
<td><strong>Operations</strong> : ( { O_1 \mapsto \langle \text{Model of } O_1 \rangle, O_2 \mapsto \langle \text{Model of } O_2 \rangle, \ldots } )</td>
</tr>
<tr>
<td><strong>Invariant</strong> : ( \ldots )</td>
</tr>
</tbody>
</table>

Figure 12: Subsystems Have Context

abstract instance. In the case of `Partial_Map_Facility`, the invariant is simply \textbf{true}, since there is no visible state in this subsystem and it therefore has no properties that might vary at run-time.

Figure 12 shows the more general case. An abstract instance always has an explicit context interface, although it may be empty. Also, an abstract instance always has an invariant for capturing constant properties that all subsystems conforming to the abstract instance must have, and must maintain throughout their lives at run-time.
3.4.4 Subsystems Have Nested Structure

In order for subsystems to capture collections of interconnected software parts, subsystems can be nested, containing other subsystems. This has already been shown in Figure 11, where the requirements for three external subsystems are nested within the single abstract instance representing the context of Partial.Map.Facility.

For example, it is possible that Partial.Map.Facility could have “sub”-facilities nested within it, giving further structure to the specification. This is distinct from the notion of Partial.Map.Facility being implemented using subsystems—instead, its specification is presented in terms of a collection of sub-specifications. Such nested constructs allow a specification designer to compartmentalize large abstract instances into conceptual chunks that are easier to digest. Figure 13 shows that any abstract instance $AI_1$ may contain other nested abstract instances.

3.4.5 Subsystems Have Specification Adornments

The requirements for ACTI advocate the separation of specification information adorning a subsystem from the actual program elements that give rise to executable artifacts (e.g., code, data). Intuitively, “specification adornment” information is auxiliary information useful for expressing the conceptual model of a software building-block; while it helps in describing the building-block, it does not contribute to the operational behavior of the software. ACTI provides for separating out this information by mirroring the basic subsystem structure to provide room for specification adornments.
### Abstract Instance

\[ AI_1 \rightarrow \]

<table>
<thead>
<tr>
<th>Context</th>
<th>Exported Behavior</th>
</tr>
</thead>
</table>
| Types   | \[
T_1 \mapsto \langle \text{Model of } T_1 \rangle
\]
|         | \[
T_2 \mapsto \langle \text{Model of } T_2 \rangle
\]
|         | \[
\vdots
\]
| Variables | \[
V_1 \mapsto \langle \text{Model of } V_1 \rangle
\]
|         | \[
V_2 \mapsto \langle \text{Model of } V_2 \rangle
\]
|         | \[
\vdots
\]
| Operations | \[
O_1 \mapsto \langle \text{Model of } O_1 \rangle
\]
|         | \[
O_2 \mapsto \langle \text{Model of } O_2 \rangle
\]
|         | \[
\vdots
\]
| Invariant | \[
\ldots
\]
| Nested Abst. Instances | \[
AI_2 \mapsto \]
|         | \[
AI_3 \mapsto \]
|         | \[
\vdots
\]

Figure 13: Subsystems Have Nested Structure
To illustrate this, we can expand the `Partial.Map.Facility` specification to add some specification adornment definitions. Figures 14, 15, and 16 present a new version of the `Partial.Map.Facility` specification.

Since we have informally described a partial map object as a "partial function" from keys to values (character strings to integers, in this case), the new `Partial.Map.Facility` specification introduces a declaration of a mathematical type `PARTIAL_FUNCTION` to capture this idea. This definition appears in box (a) in Figure 14, and explains that a `PARTIAL_FUNCTION` is a set of ordered pairs of character strings and integers with the function property, i.e., where all character strings in the set are unique. The `PARTIAL_FUNCTION` type is a purely mathematical construct, and is only used for formal specification purposes—it is not a programming type of which clients can declare program variables.

Along with the `PARTIAL_FUNCTION` type, a purely mathematical function `DEFINED_IN` has also been added to the `Partial.Map.Facility`, in box (b) of Figure 14. Intuitively, `DEFINED_IN` determines whether a given character string is part of the domain of a given partial function.

The addition of specification adornment definitions provides subsystem designers with a way to build tools for describing their conceptual models. These definitions are custom building-blocks for mathematical modeling, and help to conceptually chunk complex mathematical models into pieces that are easier for client programmers to assimilate independently. Eventually, we will be able to generalize the `PARTIAL_FUNCTION` adornment definitions presented here and move them completely
facility Partial_Map_Facility

calendar

global context

facility Standard_Character_String_Facility
facility Standard_Integer_Facility
facility Standard_Boolean_Facility

local context

math subtype PARTIAL_FUNCTION is

| set of ( |
| d : math[Character_String] |
| r : math[Integer] |
| ) |

exemplar m

constraint

for all d : math[Character_String],
    r1, r2 : math[Integer]
where ((d, r1) is in m and
    (d, r2) is in m)
    (r1 = r2)

math operation DEFINED_IN ( |

| m : PARTIAL_FUNCTION |
| d : math[Character_String] |
| ) : boolean |

explicit definition

there exists r : math[Integer]
    ((d, r) is in m)

interface

type Partial_Map is modeled by \textbf{PARTIAL_FUNCTION}(c)

exemplar m

initialization

ensures m = empty_set

Figure 14: An Alternative Specification of Partial_Map_Facility
operation Define (  
    alters m : Partial.Map  
    consumes d : Character.String  
    consumes r : Integer  
  )  
  requires not DEFINED_IN (m, d)  
  ensures m = #m union {(#d, #r)}

operation Undefine (  
    alters m : Partial.Map  
    preserves d : Character.String  
    produces d_copy : Character.String  
    produces r : Integer  
  )  
  requires DEFINED_IN (m, d)  
  ensures (d, r) is in #m and  
            m = #m - {(d, r)} and  
            d_copy = d

operation Undefine_Any_One (  
    alters m : Partial.Map  
    produces d : Character.String  
    produces r : Integer  
  )  
  requires m ≠ empty_set  
  ensures (d, r) is in #m and  
            m = #m - {(d, r)}

operation Is_Defined (  
    preserves m : Partial.Map  
    preserves d : Character.String  
  ) : Boolean  
  ensures Is_Defined iff DEFINED_IN (m, d)

Figure 15: An Alternative Specification of Partial.Map_Facility (continued)
operation Size (  
    preserves m : Partial.Map  
  ) : Integer  
ensures Size = |m|

end Partial.Map.Facility

Figure 16: An Alternative Specification of Partial.Map.Facility (continued)

out of this particular facility so that they may be reused in many specifications.  
For now, however, we can examine how the addition of these definitions affects the  

Figure 17 shows the meaning of the new version of Partial.Map.Facility, including its specification adornments. Intuitively, specification designers might need  
to define new mathematical types, mathematical variables, or mathematical operations  
to support the construction of complex conceptual models. Further, we can group these specification adornment definitions together into a distinct environment  
so that they are not accidentally confused\(^3\) with the exported program-level behavior  
provided by a subsystem. As with abstract instances, these specification adornment  
environments can be nested within each other so that the designer can compartmentalize adornment definitions as appropriate.

Figure 17 shows that the new version of Partial.Map.Facility includes two  
specification adornment definitions: one for a type, and one for an operation. The  
name of the specification adornment type, PARTIAL_FUNCTION, is associated with an  
\(^3\)Confused by a human reader, not by the formal semantics.
**Figure 17: The Meaning of the Alternative Partial.Map.Facility Specification**
abstract model of that mathematical type, corresponding to box (a) in Figure 14. This mathematical type has the same model that was presented in Figure 8, namely a set of order pairs of character strings and integers, where all character strings in the set are unique. The name of the specification adornment operation, DEFINED_IN, is similarly associated with an abstract model of its value, corresponding to box (b) in Figure 14.

Given these specification adornment definitions, we can use them to define the mathematical model of the type Partial.Map. This is easy in this case, since we have constructed the definition of the type PARTIAL_FUNCTION to exactly coincide with our conceptual view of the type Partial.Map. As shown in box (c) of Figure 14, it is easy to say that the conceptual model for the programming type Partial.Map is the same as that of the mathematical type PARTIAL_FUNCTION.

To further demonstrate the use of specification adornment definitions, we can examine a variation on the basic partial map specification. Figures 18 and 19 present the specification for Communal_PMap_Facility. The details of many of the operations have been elided here for brevity.

This new facility specification defines a "communally bounded" Partial.Map. That is, there is a bound placed on the aggregate size of all objects of type Partial.Map considered together. The specific bound is given here by the specification adornment operation MAX_TOTAL_SIZE, defined as the constant 1024, which places an upper limit on the total number of associations that can be stored in all Partial.Map objects combined. A purely mathematical state variable, total.size, has also been
facility Communal_PMap_Facility

class
global context
    facility Standard_Character_String_Facility
    facility Standard_Integer_Facility

local context

math subtype PARTIAL_FUNCTION is
    set of (
        d : math[Character_String],
        r : math[Integer]
    )

exemplar m

constraint
    for all d : math[Character_String],
        r1, r2 : math[Integer]
    where ((d, r1) is in m and
        (d, r2) is in m)
    (r1 = r2)

math operation DEFINED_IN (m : PARTIAL_FUNCTION, d : math[Character_String]) : boolean
    explicit definition
        there exists r : math[Integer]
        ((d, r) is in m)

math operation MAX_TOTAL_SIZE : integer
    explicit definition
        1024

Figure 18: The Specification of Communal_PMap_Facility
state variables

<table>
<thead>
<tr>
<th>total.size : integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint</td>
</tr>
</tbody>
</table>
| 0 \leq \text{total.size} \leq \text{MAX}_\text{TOTAL}_\text{SIZE} 
  (a)                |
| initialization       |
| ensures \text{total.size} = 0 |

interface

type Partial.Map is modeled by PARTIAL_FUNCTION
exemplar m
initialization
ensures m = empty_set

operation Define ( 
  alters m : Partial.Map
  consumes d : Character.String
  consumes r : Integer
)

referenced state variables

alters total.size

requires not DEFINED_IN (m, d) and

\text{total.size} < \text{MAX}_\text{TOTAL}_\text{SIZE}

ensures m = #m union \{(d, #r)\} and

\text{total.size} = \#\text{total.size} + 1

... 

end Communal.PMap.Facility

Figure 19: The Specification of Communal.PMap.Facility (continued)
added to track the aggregate size of all active partial map objects (Figure 19, box (a)). The pre- and postconditions of the operations provided by the facility have been modified to reflect how they depend on and change total.size.

If one places bounds on the size of single objects, the implementer has the freedom of using fixed-size representations for program objects, which often permits more efficient implementations. If such a bound is assured, the amount of memory allocated for the representation of one program object is no longer dynamic, and memory management costs are correspondingly reduced. Similarly, bounding the "communal" size of all objects of a given type allows the implementer the freedom to use a common, fixed-size pool of resources to implement all of the objects of the corresponding type, making the creation or destruction of individual objects very efficient. These strategies for implementing data abstractions are called shared realizations [11, p. 288], since the resources used to implement the abstraction are shared among the individual data objects involved.

Figure 20 shows the meaning of the Communal_PMap_Facility in in ACTI terms. Here, the definitions of MAX_TOTAL_LENGTH and total.size have been added as additional specification adornments. Note that total.size is not a program variable that clients of Communal_PMap_Facility can manipulate. Instead, it is purely a specificationational device used to abstractly model the shared state within any implementation of Communal_PMap_Facility, so that the behavior of the exported operations can be properly described. For this reason, it is treated as a specification adornment variable, and not an exported program variable.
### Context

<table>
<thead>
<tr>
<th>Nested</th>
<th>Concrete</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char_Str_Fac</td>
<td>Int_Fac</td>
<td>Bool_Fac</td>
</tr>
</tbody>
</table>

### Specification Adornment

<table>
<thead>
<tr>
<th>Types</th>
<th>Variables</th>
<th>Operations</th>
<th>Invariant</th>
<th>Nested Spec. Adorn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARTIAL_FUNCTION</td>
<td>total_size = Z</td>
<td>DEFINED_IN, MAX_TOTAL_LENGTH</td>
<td>0 ≤ total_size ≤ MAX_TOTAL_LENGTH</td>
<td>{}</td>
</tr>
</tbody>
</table>

### Exported Behavior

<table>
<thead>
<tr>
<th>Types</th>
<th>Variables</th>
<th>Operations</th>
<th>Invariant</th>
<th>Abstract Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partial_Map</td>
<td>Define, Undefine, Undefine.Any.ONE, Is_DEFINED, Size, Initialize, Finalize, Swap</td>
<td>true</td>
<td>{}</td>
<td></td>
</tr>
</tbody>
</table>

---

Figure 20: The Meaning of the Communal_PMap_Facility Specification
Figure 20 also shows the abstract model associated with a variable, in this case a specification adornment variable. The abstract model associated with the variable `total.size` is simply its type, `Z`, the mathematical domain of integers. Program

<table>
<thead>
<tr>
<th>Abstract Instance</th>
<th>Context</th>
<th>Specification Adornment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AI_1</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Types              | {⋯}    |
| Variables          | {⋯}    |
| Operations         | {⋯}    |
| Invariant          | {⋯}    |

\[
\{ S_{A_1} \mapsto \quad , \quad S_{A_2} \mapsto \quad , \quad \ldots \}
\]

| Nested Spec. Adorn. | {⋯}    |

<table>
<thead>
<tr>
<th>Exported Behavior</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>{⋯}</td>
<td></td>
</tr>
<tr>
<td>Variables</td>
<td>{⋯}</td>
<td></td>
</tr>
<tr>
<td>Operations</td>
<td>{⋯}</td>
<td></td>
</tr>
<tr>
<td>Invariant</td>
<td>{⋯}</td>
<td></td>
</tr>
</tbody>
</table>

\[
\{ A_{I_2} \mapsto \quad , \quad A_{I_3} \mapsto \quad , \quad \ldots \}
\]

| Nested Abst. Instances | {⋯}    |

Figure 21: Subsystems Separate Specification Adornments from Program Code Elements
variables have similar abstract models, consisting simply of the name of the type delineating the domain of values that variable may take on.

Communal_PMap_Facility also now has a non-trivial invariant as is shown in Figure 21. This invariant property captures the constraints placed on the specification adornment definitions in the facility, which were originally written down in box (a) of Figure 19.

This gives us a more refined picture of the structure of abstract instances in ACTI, as depicted in Figure 21. Here, one can see the intuitive structure of an abstract instance as being partitioned into three parts: the context, the specification adornments, and the exported behavior. The context is expressed as another abstract instance, and the structure of the specification adornment definitions mirrors that of the exported behavioral description.

3.4.6 Templates—Generic Subsystems

So far, a fairly detailed picture of abstract instances has been constructed at an intuitive level. Now it is possible to generalize this basic information to other kinds of ACTI subsystems. In this section, we consider abstract templates, which are parameterized subsystems.

A parameterized subsystem can be viewed as a function that takes one subsystem as a parameter and produces another as a result. In Ada, parameterized program units are known as generics; in C++, they are known as templates; in Standard ML, they are known as functors; and in RESOLVE, they are known as concepts (although RESOLVE realizations are also parameterized). Figures 22 and 23 present a
concept Communal_PMap_Template

context
global context
   facility Standard_Boolean_Facility
   facility Standard_Integer_Facility

parametric context

\[
\begin{array}{l}
\text{type D.Item} \\
\text{type R.Item} \\
\text{constant max.total.size : integer} \\
\quad \text{restriction} \\
\quad \quad \text{max.total.size > 0}
\end{array}
\]

local context

math subtype PARTIAL_FUNCTION is
   set of (d : math[D.Item], r : math[R.Item])

exemplar \( m \)
constraint for all d : math[D.Item], r1, r2 : math[R.Item]
   where ((d, r1) is in \( m \) and (d, r2) is in \( m \))
   (r1 = r2)

math operation DEFINED_IN (m : PARTIAL_FUNCTION, d : math[D.Item])
  ): boolean

explicit definition
   there exists r : math[R.Item] ((d, r) is in \( m \))

Figure 22: A Generic Specification: Communal_PMap_Template
state variables
  total_size : integer

constraint
  0 ≤ total_size ≤ max.total.size

initialization
  ensures total_size = 0

interface

type Partial.Map is modeled by PARTIAL_FUNCTION

exemplar m

initialization
  ensures m = empty.set

operation Define ( 
  alters m : Partial.Map
  consumes d : D.Item
  consumes r : R.Item
)

  referenced state variables
  alters total_size
  requires not DEFINED_IN (m, d) and
  total_size < max.total.size
  ensures m = #m union {(#d, #r)} and
  total_size = #total_size + 1

end Communal_PMap_Template

Figure 23: A Generic Specification: Communal_PMap_Template (continued)
parameterized version of the communally bounded partial map specification called \texttt{Communal.PMap.Template}.

The \texttt{Communal.PMap.Template} does not describe a partial map between \texttt{Character.Strings} and \texttt{Integers}, but describes a partial map between any given domain type and range type. It is \textit{parametric} in these two types, here called \texttt{D.Item} and \texttt{R.Item}. Further, \texttt{Communal.PMap.Template} is parametric in the upper bound placed on the aggregate size of all the partial map objects it provides, through its integer-valued \texttt{max.total.size} parameter. This \textit{parametric} context is shown in box (a) of Figure 22.

Before a client programmer can use the \texttt{Communal.PMap.Template} subsystem in a program, she must choose specific values for these parameters and \textit{instantiate} the template. In many ways, instantiating a template with specific parameter values is like applying a function to its arguments. In this case, applying the \texttt{Communal.PMap.Template} to specific arguments will generate a new subsystem—an abstract \textit{instance}. If the client chooses to use \texttt{Character.String} as the value for the \texttt{D.Item} parameter, \texttt{Integer} as the value for the \texttt{R.Item} parameter, and 1024 as the value for the \texttt{max.total.size} parameter, the resulting abstract instance will be just like the \texttt{Communal.PMap.Facility} of Section 3.4.5. Now, however, any values that meet the restrictions specified in Figure 22, box (a), can be chosen, and several instances of the template can be generated using different parameter combinations.

Figure 24 shows \texttt{Communal.PMap.Template} as a function from a parameter abstract instance $AI_1$ to a result abstract instance $AI_2$. $AI_1$ is a bundle of all the parameters
Figure 24: The Meaning of Communal_PMap_Template
to the template, in this case the D.Item and R.Item types and the max.total.size variable. After choosing specific values for these parameters, the client programmer can apply the template to produce a new abstract instance, \( A_l \). This new instance will export a Partial.Map type that can store associations between D.Item and R.Item elements.

Figure 25 illustrates the general case, where an arbitrary abstract template is viewed as a function from abstract instances to abstract instances. Figures 24 and 25 also show that subsystems can contain nested templates, just as they contain nested instances.

3.4.7 Interpretation Mappings

In addition to the four kinds of subsystems, ACTI also includes the notion of an interpretation mapping. An interpretation mapping is an explanation of why one subsystem can be "interpreted" in terms of another. This is a generalization of "is-a" relationships or behavioral subtyping between subsystems.

For example, suppose that in addition to the Communal.PMap.Facility abstract instance, there were also a Container.Facility abstract instance that described the exported behavior of any subsystem that provides a data type that can be used as a "container" to hold other data objects. One might expect the abstract model of a general-purpose container type to be a multiset. Perhaps Container.Facility provides a Size operation to query the size of any container object.

If this were the case, it would be possible to define an interpretation mapping that explains how Communal.PMap.Facility can be interpreted as a Container.Facility.
Figure 25: Templates Are Functions from Subsystems to Subsystems
This involves explaining what types and operations in Communal.PMap.Facility will be identified with those present in Container.Facility. It also involves explaining the relationships between the abstract models of each type involved, and also explaining the relationships between variables and specification adornment definitions in the two subsystems.

An interpretation mapping is a construct for recording this information. Such a mapping can be used to explain why one specification (abstract instance) can be considered a behavioral subtype of another, or to explain why an implementation (concrete instance) conforms to a given specification. The notion of an interpretation mapping borrows heavily from Goguen's definition of views in OBJ [21]. ACTI also includes interpretation mapping templates to describe parameterized interpretations. These features are formally defined in Chapter IV.

### 3.4.8 Subsystem Summary

Sections 3.4.2 through 3.4.6 have incrementally shown the basic conceptual structure of abstract instances and abstract templates. Figure 26 summarizes this structure by showing all of the parts of an abstract instance.

To understand ACTI at an intuitive level, the only remaining distinction to be made is between abstract instances and concrete instances. Because abstract instances (or templates) describe behavior, while concrete instances (or templates) describe representations or implementations, the difference between the two does not lie in their structure, but in the kinds of values they associate with names. Abstract instances associate names with models of behavior, while concrete subsystems
<table>
<thead>
<tr>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Specification Adornment</strong></td>
</tr>
<tr>
<td>Types</td>
</tr>
<tr>
<td>Variables</td>
</tr>
<tr>
<td>Operations</td>
</tr>
<tr>
<td>Invariant</td>
</tr>
</tbody>
</table>

**Spec. Adorn. Instances**

| $SAI_1 \mapsto \square$ |

| Spec. Adorn. Templates | $SAT_1 \mapsto (\square \rightarrow \square)$ |

**Exported Behavior**

| Types | $\{\ldots\}$ |
| Variables | $\{\ldots\}$ |
| Operations | $\{\ldots\}$ |
| Invariant | $\ldots$ |

**Abstract Instances**

| $AI_1 \mapsto \square$ |

| Concrete Instances |

| $CI_1 \mapsto \square$ |

**Interpretation Mappings**

| $IM_1 \mapsto \uparrow$ |

**Abstract Templates**

| $AT_1 \mapsto (\square \rightarrow \square)$ |

| Concrete Templates |

| $CT_1 \mapsto (\square \rightarrow \square)$ |

**Int. Mapping Templates**

| $IMT_1 \mapsto (\square \rightarrow \uparrow)$ |

---

Figure 26: The Details of an Abstract Instance
associate names with both models of behavior and actual values (for variables) or implementations (for operations). Otherwise, both kinds of instances have a basic structure that is exactly parallel—the structure shown in Figure 26. This figure provides a schematic representation of the true structure of an instance. Both abstract and concrete instances have this layout.

At the level depicted in these figures, abstract and concrete components look the same; the intuitive difference is that abstract components map variable and operation names to behavioral descriptions (variables to types, and operations to pre- and postcondition-like profiles), while concrete instances also map them to actual values (variables to variable values, and operations to computed functions).

Interestingly, a concrete instance (intuitively, a collection of named representations and implementations) contains the same sorts of information that one would also like to capture in a program state. Not only can it contain the name-to-value associations for types, variables, and operations, but by subsystem nesting it can also contain the name-to-value associations for other abstract or concrete instances and abstract or concrete templates. As a result, one can think of the program state, or execution environment, for the dynamic semantics of a program language as being an extremely large concrete instance subsystem. ACTI carries out this identification, interchangeably using the terms Environment and Concrete Instance to refer to exactly the same kinds of mathematical objects.

In fact, one can carry this identification a step farther. For languages that include some notion of a compiler-maintained library of previously compiled units (e.g., RE-
SOLVE and Ada), one can use the space of Environments or Concrete Instances as the basis for defining the semantic denotation of such program libraries.

### 3.5 Chapter Summary

The ACTI software model was developed with four goals in mind: supporting effective mental models, having a formal definition, maintaining programming language independence, and ensuring that it is comprehensive. As a result, ACTI is based on the mathematics behind denotational semantics. It also respects the strict separation between the compilation and execution phases of programming. Most importantly, however, it is build on the idea that module-like program constructs have independent semantic denotations during the execution phase.

The ACTI model is centered around four different classes of subsystems: abstract instances, concrete instances, abstract templates, concrete templates. This chapter presented the example of a software component exporting a Partial.Map abstract data type (ADT). Through this example, the basic facets of abstract and concrete instances were shown. These include exported behavior, such as types, variables, and operations; context, an explicit definition of external dependencies; invariants, at both the level of both types and subsystems; specification adornment definitions; and nested subunits. In addition to instance subsystems, ACTI includes templates, which are functions from instances to instances that represent parameterized subsystems. Interpretation mappings are provided as a mechanism for explaining how the behavior described in one subsystem can be “interpreted as” conforming to the ab-
stract behavioral description in another subsystem. All of these subsystem varieties are formally defined in Chapter IV.
CHAPTER IV

The ACTI Definition

This chapter presents the formal definition of ACTI. For the reader interested in the development of ACTI, Appendices A through C document the relevant background material, the method used to develop it, its requirements, and brief comparisons with four existing programming languages.

4.1 Foundational Concepts

The ACTI model is defined as a collection of mathematical spaces, together with a collection of operations on those spaces. To simplify the formal definition of the model, it relies on several foundational concepts. These basic concepts are central to understanding ACTI and how to use it, and they are described in this section.

4.1.1 Complete Partial Orders

Because ACTI is defined using the concepts of denotational semantics, the reader should be familiar with the idea of a complete partial order. A partial order is a pair \((D, \sqsubseteq)\) of a set \(D\) and a relation \(\sqsubseteq\) on \(D\) that is reflexive, antisymmetric, and transitive. Briefly, a complete partial order (CPO) is a partial order \((D, \sqsubseteq)\) where the following conditions hold:

75
1. The set $D$ has a "least" element with respect to $\sqsubseteq$. This least element is denoted by $\bot_D$, or simply $\bot$ (which is read "bottom").

2. For every chain $S$ of elements in $D$ related by $\sqsubseteq$, there exists a least upper bound of the chain, denoted $\sqcup S$.

For formal definitions of chains, least upper bounds, and other concepts related to CPOs, consult a text on denotational semantics (e.g., [42]).

Because CPOs play such a central role in denotational semantics, it is not surprising that all of the mathematical spaces in ACTI are CPOs. More formally, the universe of discourse in which ACTI is defined is the category $\textbf{CPO}$ of CPOs; all of the mathematical spaces defined in ACTI are objects in this category.

For the purposes of this dissertation, the defined mathematical spaces can be appreciated with a limited knowledge of CPOs, including two important facts. First, any ordinary set of values can be turned into a simple CPO; and second, the collection of all CPOs is closed under Cartesian product and function formation operations.

An ordinary set $S$ can be turned into a flat CPO [42, p. 74–75]. To do this, pick any value that is not in the set $S$; call this element $\bot_S$. Now it is easy to form the set $S_\bot = S \cup \{\bot\}$. Then the relation $\sqsubseteq_S$ can be defined:

$$\forall a, b \in S : a \sqsubseteq_S b \iff (a = \bot_S \text{ or } a = b)$$

Clearly, $(S_\bot, \sqsubseteq_S)$ is a CPO.

Next, if $(D_1, \sqsubseteq_{D_1}), \ldots, (D_n, \sqsubseteq_{D_n})$ are all CPOs, then $(D_1 \times \cdots \times D_n, \sqsubseteq_\times)$ is also a CPO [42, p. 75]. Here, the relation $\sqsubseteq_\times$ is the relation formed by point-wise application
of the component $\subseteq_{D_i}$ relations. Similarly, the least element of the new CPO is $\bot_{D_1} \times \cdots \times \bot_{D_n}$.

Finally, if $(D, \subseteq_D)$ and $(E, \subseteq_E)$ are both CPOs, then $((D \to E), \subseteq_{(D \to E)})$ is also a CPO [42, p. 75]. The least element in the new CPO is the function that maps every element in $D$ to $\bot_E$. Once again, the relation $\subseteq_{(D \to E)}$ is formed by point-wise comparison of the range values of two functions across all possible domain values.

In the examples above, $\bot$ and $\subseteq$ have been subscripted to indicate which sets they are intended to operate on, for clarity. Typically, however, these symbols are written without subscripts where their intended meaning is clear from context. This convention will be followed in the remainder of this chapter.

The majority of the mathematical spaces in ACTI are constructed using cartesian product formation or function formation, so it should be clear at each point how the individual spaces are interpreted as CPOs.

### 4.1.2 Name-to-Value Mappings

Section 3.4 discusses how run-time program states can be viewed in general terms as name-to-value mappings. Similarly, subsystems are collections of name-to-value associations that can be manipulated as a whole.

Within this dissertation, the term *mapping* will be used as another name for a function $f : D \to E$ from one CPO to another. The notation\(^1\) $\text{Dom } f$ will be used to refer to the "meaningful" domain of $f$; that is, the set of all $d \in D$ where $f(d) \neq \bot$.

---

\(^1\) All of the notation presented in this section is adapted from that in the Standard ML definition [48, pp. 16–17]. Note, however, that here all mappings are total functions, while the Standard ML notation is used with partial functions having finite domains.
Similarly, Ran $f$ will be used to refer to the "meaningful" range of $f$; that is, the set of all $e \in E$ not equal to $\bot$ such that there is some $d \in D$ where $f(d) = e$. We will say a particular $d \in D$ is defined in $f$ when $f(d) \neq \bot$. Otherwise, when $f(d) = \bot$, we say that $d$ is undefined in $f$.

Because the mappings associated with typical software subsystems usually have small meaningful domains, they are sometimes written out explicitly. The notation $f = \{d_1 \mapsto e_1, \ldots, d_n \mapsto e_n\}$ will be used to explicitly define a mapping with $\text{Dom } f = \{d_i : 1 \leq i \leq n\}$ and $\text{Ran } f = \{e_i : 1 \leq i \leq n\}$. Note that $f$ is still a total function, and it is understood that for all $d \notin \text{Dom } f$, $f(d) = \bot$. Thus, the shorthand $\{\}$ refers to the mapping for which $f(d) = \bot$ for all $d \in D$.

Given two maps $f$ and $g$ both from $D$ to $E$, we can also talk about combining them. The map $f + g$, read $f$ modified by $g$ [48, p. 17], is the map with meaningful domain $\text{Dom } f \cup \text{Dom } g$ defined as follows:

\[
(f + g)(d) = \begin{cases} 
  g(d) & \text{if } g(d) \neq \bot \\
  f(d) & \text{otherwise}
\end{cases}
\]  

(4.2)

The term name-to-value mapping will be used for any mapping $f : \text{Names} \to E$ from the space of $\text{Names}$ to another CPO $E$. The space $\text{Names}$ is defined in Section 4.3.

4.1.3 Environments

Name-to-value mappings are intuitively easy to understand, especially when related to their roots in program states. In traditional operational or denotational semantics, a program state simply records the value of every named object at a particular point.
in the execution history of a program. One can also think of such a collection of name-to-value associations as an "execution environment," or simply an "environment" that gives particular bindings for some set of names. As a result, the term environment will be used for any ACTI mathematical space that is a name-to-value mapping.

4.1.4 Signatures

Because ACTI respects the central concerns of software engineering, its design was affected by the need for appropriately separating specifications from implementations. As a result, many of the mathematical spaces in the model are related in pairs—one space to represent actual software objects, and the other to represent specifications, or conceptual models, of those objects. A space of specification-only values is called a space of signatures. Such a space contains behavioral signatures, each of which describes the externally observable behavior of some class of software objects. This is a richer notion that contrasts with the usual use of the term "signature" in computer science, where it is often used to mean a purely syntactic description, such as just an operation's parameter profile.

For example, the space of Abstract Templates described in Section 4.8 captures the denotations of generic specifications. It is closely tied to the space of Abstract Template Signatures, which capture the designer's conceptual model of a generic specification—its parameter profile and an abstract description of its behavior expressed as pre- and postconditions. All ACTI spaces labeled as spaces of signatures serve this purpose. As one would expect from the informal overview presented in Chapter III, the signature for a concrete instance is an abstract instance.
4.1.5 Well-Formed Objects

Unfortunately, simply defining the structure of values in the mathematical spaces of the ACTI model is not enough. Many kinds of objects in ACTI contain redundant or mutually dependent information, and the consistency of this information cannot be assured simply by examining the structure of the object.

Thus, meaningful objects that can model actual program structures form a subset of the values that are contained in ACTI's spaces. We call these objects well-formed objects, indicating that they are internally consistent. As each new mathematical space is introduced in the ACTI definition, complete criteria for all well-formed objects within that space are also defined. For spaces where no explicit criteria are given, all objects within the space are considered well-formed.

For many spaces, we are concerned with more than just internal consistency—we are also concerned with mutual consistency in the context of some other object, say the containing subsystem. Thus, some objects can only be judged as well-formed with respect to another object. Since subsystems are the cornerstone of the ACTI world, the well-formedness of any particular subsystem is absolute, independent of any other objects. This is reflected in the criteria for well-formed abstract instances, concrete instances, abstract templates, and concrete templates. For most other spaces, however, the notion of a well-formed object is explicitly defined with respect to an object from another space, often an abstract or concrete instance.
Here, we are only concerned with collections of meaningful objects. Throughout the definition we assume that all specific objects under discussion are well-formed.

### 4.1.6 Assertive Programming

Some programming languages, such as RESOLVE and OBJ, already allow formal behavioral descriptions of program elements. In order to support the formal semantics of assertions used in these descriptions, ACTI incorporates the ideas of *assertive programming*. Assertive programming is centered around the idea of giving meaning to behavioral assertions, such as pre- and postconditions, the same way meaning is given to procedures or statements—in the formal semantics, assertions affect the run-time state of execution.

To record the effects the formal assertions have on a program's state within the formal semantics, we follow Ernst *et al.* [12, 10] and add the notion of an *Assert Status* to the ACTI definition. An assert status component is added to the denotation of a run-time state to reflect the cumulative effect of "executing" the assertions in a program.

An *Assert Status* value is one of \{CF, VT, NL, \}. Certain assertions in a program must be true in order for the program to be considered correct, and the violation of such an assertion results in an assert status of *categorically false* (CF). Similarly, some assertions are assumed to be true, and if such an assertion fails to hold, it results in an assert status of *vacuously true* (VT). The NL assert status value stands for *neutral*, and is used in program states where no assertions have been violated. The value "bottom" (\(\bot\)) is used when the assert status of a particular program state is
indeterminate. The curious reader is encouraged to explore the concepts of assertive programming further ([12], [10, pp. 267-268]).

4.1.7 Intuitive Explanations

Continuing the approach begun in Chapter III, the definition of the ACTI model is written to appeal as much as possible to one's intuition and common sense. This is as much to ease the understanding of the formal definition as it is to make the material accessible to a wider audience. To this end, the formal definitions of all but the simplest mathematical spaces are followed by an "intuitive explanation" of the ideas that are captured in the formal definition. Most frequently, this means relating the abstract formal definition to more concrete programming language terms, but the additional explanatory material also may contain rationale for some choices made in the formal definition. The reader who is more interested in getting a quick grasp of the general ideas behind ACTI's structure rather than a perfect mastery of the formal definitions themselves should focus attention toward these intuitive explanations.

4.2 Outline of the Mathematical Spaces in ACTI

Because of the four-way partitioning of the kinds of subsystems in the ACTI world, the formal definition of ACTI is centered around four distinct mathematical spaces: one for each kind of subsystem. For simplicity, these spaces are named after each of the subsystem classes: abstract instances, concrete instances, abstract templates, and concrete templates.
The remainder of this chapter presents the definitions of these mathematical spaces, along with the details of the smaller spaces from which they are constructed. These are presented in the following order:

• **Names (N)**

• **Modeling Domains (MD)**

• **Environments or Concrete Instances (E)**
  
  – **Type Environments (TE)**
    
    * **Type Models (TM)**
  
  – **Variable Environments (VE)**
    
    * **Variable Models (VM)**
    * **Variable Signatures (VS)**
  
  – **Operation Environments (OE)**
    
    * **Operation Meanings (OM)**
    * **Operation Models (OMOD)**
    * **Parameter Profiles (PP)**
  
  – **Environment Invariants (EI)**

  – **Abstract Instance Environments (AIE)**
    
    * **Context-Bound Abstract Instances (CBAI)**

  – **Concrete Instance Environments (CIE)**
* Context-Bound Concrete Instances (CBCI)

- Interpretation Mapping Environments (IME)
- Abstract Template Environments (ATE)
- Concrete Template Environments (CTE)
- Interpretation Mapping Template Environments (IMTE)

• Abstract Instances (AI)

- Variable Environment Signatures (VES)
- Operation Environment Signatures (OES)
- Abstract Template Environment Signatures (ATES)
- Concrete Template Environment Signatures (CTES)

• Concrete Templates (CT)

- Concrete Template Signatures (CTS)

• Abstract Templates (AT)

- Abstract Template Signatures (ATS)

• Specification Adornment Environments (SAE)

- Specification Adornment Variable Environments (SAVE)
- Specification Adornment Environment Invariants (SAEI)
- Specification Adornment Instance Environments (SAIE)
By convention, the plural form of a space name is used when referring to the entire space (e.g., Names), while the singular form is used when referring to a particular element within that space (e.g., a Name). Further, upper case script abbreviations are often used (indicated above in parentheses after the complete name).

After the definitions for ACTI's mathematical spaces are presented, this section continues with a discussion of operations on subsystems. Chapter V then uses a brief example to help illustrate the spaces.
4.3 Names ($\mathcal{N}$)

The mathematical space $Names, \mathcal{N}$, is the CPO of all identifiers available for labeling program entities, including types, variables, operations, adornment definitions, and all kinds of subsystems. A Name from $\mathcal{N}$ should not be confused with the notion of a textual identifier in a programming language, as discussed below. The only assumptions made about this space are that it is a CPO, and that it contains a countably infinite number of unique elements (i.e., has cardinality $\aleph_0$). In applying the ACTI model to a specific programming language, any convenient mathematical space satisfying these assumptions can be used for $Names$.

Throughout this chapter, we assume without loss of generality that all defined objects are given distinct names, and there is no notion of hiding by scope or overloading. Handling these cases is a natural extension, and is usually defined as part of the static (compilation phase) semantics of a particular programming language.

An Intuitive Explanation of Names

When writing a program, it is natural for one to give names to each declared entity. $\mathcal{N}$ helps to flesh out the formal analog of this idea. In ACTI, the world is made up of subsystems, and every declared entity within every subsystem must be associated with some (unique) identifier. These identifiers are drawn from $\mathcal{N}$, and are used when it is necessary to refer to specific objects within a particular subsystem. The requirement that $\mathcal{N}$ be countably infinite ensures that there will always be enough
unique identifiers to label every object in a subsystem (or program), no matter how large.

Since we are working in the realm of mathematics, our view of identifiers can be simplified in comparison with textual names in a programming language. In particular, since ACTI is only concerned with the dynamic (execution phase) semantics of programs, syntactic conveniences like allowing distinct objects in distinct scopes to carry the same name, or allowing the same name to denote distinct operations in different contexts, are not relevant. For the purposes of this definition, we can consider all of these issues to be handled effectively by an appropriate static semantics defining the relevant naming rules for a given programming language syntax. The rules for a particular programming language might even restrict the set of possible textual names to be finite (e.g., no names longer than 32 characters), but this usually only puts a limit on the number of textual names visible within a given scope, not on the number of distinct objects that may exist within a program.

Within the dynamic semantics, the concept of a Name becomes a unique, logical identifier for some program object, and this identifier need not be related to the actual textual name by which the programmer might refer to that object in source code. In the remainder of this chapter, whenever the term "identifier" is used, it will always refer to some element of $\mathcal{N}$. 
4.4 Modeling Domains ($\mathcal{MD}$)

The mathematical space *Modeling Domains, $\mathcal{MD}$*, is the CPO of all CPOs available for use in abstractly modeling data types contained in other ACTI structures. The only assumptions made about this space are that it is a CPO, and that it contains a countably infinite number of CPOs (i.e., has cardinality $\aleph_0$). Although $\mathcal{MD}$ itself must be denumerable, there is no restriction on the cardinality of the elements of $\mathcal{MD}$.

In applying the ACTI model to a specific programming language, any convenient mathematical space satisfying these assumptions can be used for $\mathcal{MD}$.

A programming language designer can construct a useful candidate for $\mathcal{MD}$ by performing the following steps:

1. Select a basis set $B$ of CPOs from the category CPO. Ensure that $B$ is at most countably infinite, and ensure that it encompasses reasonable elements for modeling the critical data types at hand.

2. Form the closure of $B$, say $B'$, under finite application of the cartesian product operator ($\times$) and function formation operator ($\rightarrow$) on CPOs.

3. Form a flat CPO from the set $B'$, and use it as $\mathcal{MD}$.

The space resulting from this process will be denumerable.

**An Intuitive Explanation of Modeling Domains**

$\mathcal{MD}$ is the realm of mathematical domains available for use as the abstract model of any data type. It is important that this space be a CPO itself in order to guarantee
the existence of least fixed points, a guarantee needed if ACTI will be used as part of a language's denotational semantics.

Fortunately, because \( \mathcal{N} \) is denumerable, no software construct (modeled in ACTI) can refer to more than a countably infinite number of distinct types. In practice, no program can refer to more than a finite number of distinct types. As a result, we can be sure that, given the correct \( \mathcal{MD} \), ACTI will be able to reasonably model all of the types relevant to any given program. The choice of the exact \( \mathcal{MD} \) to use, however, is left open in this definition. The question of how to best choose \( \mathcal{MD} \) for any specific programming language is beyond the scope of this work, but is not essential to understanding the ACTI model.

4.5 Environments or Concrete Instances (\( \mathcal{E} \))

The space of Environments is a cartesian product:

\[
\mathcal{E} = \mathcal{AI} \times \mathcal{SAE} \times \mathcal{T}E \times \mathcal{VE} \times \mathcal{OE} \times \mathcal{EI} \times \mathcal{AIE} \times \mathcal{CIE} \times \mathcal{IME} \times \mathcal{ATE} \times \mathcal{CTE} \times \mathcal{IMTE}
\]

Thus, an Environment (or, alternatively, a Concrete Instance) is a 12-tuple \( E = (\mathcal{CTXT}, \mathcal{SAE}, \mathcal{TE}, \mathcal{VE}, \mathcal{OE}, \mathcal{EI}, \mathcal{AIE}, \mathcal{CIE}, \mathcal{IME}, \mathcal{ATE}, \mathcal{CTE}, \mathcal{IMTE}) \), where:

\( \mathcal{CTXT} \) is an Abstract Instance (Section 4.6, p. 107) declaring all external references appearing in this concrete instance.

\( \mathcal{SAE} \) is a Specification Adornment Environment (Section 4.9, p. 118) containing all of the specification adornment definitions provided by \( E \).
$TE$ is a *Type Environment* (Section 4.5.1, p. 94) containing all of the types provided by $E$.

$VE$ is a *Variable Environment* (Section 4.5.2, p. 95) containing all of the variables provided by $E$.

$OE$ is an *Operation Environment* (Section 4.5.3, p. 97) containing all of the operations provided by $E$.

$EI$ is an *Environment Invariant* (Section 4.5.4, p. 101) that always holds for $E$.

$AIE$ is an *Abstract Instance Environment* (Section 4.5.5, p. 101) containing all of the *Abstract Instances* provided by $E$.

$CIE$ is an *Concrete Instance Environment* (Section 4.5.6, p. 104) containing all of the *Concrete Instances* provided by (i.e., sub-environments contained in) $E$.

$IME$ is an *Interpretation Mapping Environment* (Section 4.5.7, p. 106) containing all of the *Interpretation Mappings* (Section 4.10, p. 126) provided by $E$.

$ATE$ is an *Abstract Template Environment* (Section 4.5.8, p. 106) containing all of the *Abstract Templates* provided by $E$.

$CTE$ is an *Concrete Template Environment* (Section 4.5.9, p. 106) containing all of the *Concrete Templates* provided by $E$.

$IMTE$ is an *Interpretation Mapping Template Environment* (Section 4.5.10, p. 107) containing all of the *Interpretation Mapping Templates* (Section 4.11, p. 136) provided by $E$. 
In order to state the well-formedness criteria for an environment, it is necessary to define *cycle-freedom*. We can say an environment $E_1$ *directly contains* another environment $E_2$ if $E_2 \in \text{Ran } E_1 \cdot \text{CIE}$. We can further say that $E_1$ *indirectly contains* $E_2$ if any environment in $\text{Ran } E_1 \cdot \text{CIE}$ contains $E_2$ (directly or indirectly). An environment $E$ is *cycle-free* if it does not contain itself, either directly or indirectly.

An *Environment* $E$ is well-formed if it is cycle-free, each of its twelve components is well-formed with respect to $E$, and $E.EI(E)$ is true—i.e., the invariant within the concrete instance holds.

For convenience, there are several notational shorthand forms we will adopt when discussing concrete instances. To talk about the set of all identifiers defined in a concrete instance $E$, we will use $\text{Dom } E$:

$$
\text{Dom } E = \text{Dom } E \cdot \text{CTXT} \cup \text{Dom } E \cdot \text{SAE} \cup \text{Dom } E \cdot \text{TE} \cup \text{Dom } E \cdot \text{VE} \cup \text{Dom } E \cdot \text{OE} \cup \text{Dom } E \cdot \text{AIE} \cup \text{Dom } E \cdot \text{CIE} \cup \text{Dom } E \cdot \text{IME} \cup \text{Dom } E \cdot \text{CTE} \cup \text{Dom } E \cdot \text{ATE} \cup \text{Dom } E \cdot \text{IMTE}
$$

We also will use $\text{TypeNames } E$ to refer to the set of all identifiers from $\mathcal{N}$ that are mapped to a *Type Model* by $E$ or some concrete subsystem nested in $E$, excluding specification adornment definitions:

$$
\text{TypeNames } E = \text{Dom } E \cdot \text{TE} \cup \text{TypeNames } E \cdot \text{CIE} \cup \text{TypeNames } E \cdot \text{CTXT}
$$
The TypeNames operator only reflects the names of types that are provided by some concrete instance—i.e., that belong to an implementation. While $E.AIE$ also provides type definitions, no programmatic implementations of those types are provided.

TypeNames is really an overloaded set of operators that are mutually defined: one for abstract instances, one for concrete instances, one for abstract instance environments, and one for concrete instance environments. We will consider VariableNames and OperationNames to be defined similarly. Also, we will define LocalTypeNames $E$ to be shorthand for TypeNames $E - TypeNames E.CTXT$, with LocalVariableNames and LocalOperationNames defined similarly.

We know that Dom $E.TE$ has no elements in common with TypeNames $E.AIE$, TypeNames $E.CIE$, or TypeNames $E.CTXT$, since all defined objects are assumed to have unique identifiers. Thus, we can also define a shorthand expression for the type model associated with a type name $t$ defined somewhere in $E$:

$$
TypeModel (t, E) = \begin{cases} 
E.TE(t) & \text{if } t \in \text{Dom } E.TE \\
TypeModel (t, E.CIE) & \text{if } t \in \text{TypeNames } E.CIE \\
TypeModel (t, E.CTXT) & \text{if } t \in \text{TypeNames } E.CTXT 
\end{cases}
$$

As with TypeNames, TypeModel is really an overloaded set of operators that are mutually defined: one for abstract instances, one for concrete instances, one for abstract instance environments, and one for concrete instance environments. The operators VariableModel and OperationModel are both defined the same way.

The signature of a concrete instance is an abstract instance. It can be generated directly from the concrete instance by the abstract operator described in Section 4.13.1, page 140.
An Intuitive Explanation of Concrete Instances

An Environment (or a Concrete Instance) is the run-time denotation of an executable subsystem (or module). Often, it is useful to think of a concrete instance as the implementation of a subsystem, such as a package body, a class implementation, or a realization.

The $CTXT$ component of an environment $E$ is an explicit declaration of all entities outside $E$ that are referred to directly in the values of any objects within $E$. This is the "context" introduced in Section 3.4.3. Effectively, $CTXT$ provides names and abstract specifications for the external objects needed to fully define $E$. Items within $E$ can only "see" the outside world through this explicit interface. In the Partial_Map_Facility example (Figure 11, p. 47), external RESOLVE facilities such as Standard_Character_String_Facility, Standard_Integer_Facility, and Standard_Boolean_Facility were all imported as context.

The $SAE$ component of $E$ collects all of the specification adornment definitions present in the subsystem (see Section 3.4.5). $TE$, $VE$, $OE$, $AIE$, $CIE$, $ATE$, and $CTE$ represent collections of sub-objects within $E$ of various kinds, and are all the direct formal analogs of the module substructures shown in Figure 26. The predicate $EI$ is a formal representation of a subsystem-level invariant which must always hold for $E$.

Finally, the $IME$ and $IMTE$ components of a concrete instance $E$ represent all of the Interpretation Mappings (Section 4.10, p. 126) and Interpretation Mapping Templates (Section 4.11, p. 136) provided by that subsystem, respectively. Briefly, interpretation mappings capture relationships between subsystems—they allow one
to express how a subsystem (either abstract or concrete) can be "interpreted" as providing the behavior described by a particular abstract instance. By allowing interpretation mappings to be given names and exported by subsystems, they become first-class citizens of the programming world that can be identified and reused independently of any pair of subsystems to which they might be applicable.

4.5.1 Type Environments (\(\mathcal{TE}\))

A Type Environment is a name-to-value mapping from \(\mathcal{N}\) to Type Models. A Type Model is a two-tuple \((MD, CNSTR)\), where:

- \(MD\) is a Modeling Domain, which is simply a CPO from the space of Modeling Domains.

- \(CNSTR\) is a predicate over \(MD\), which acts as a "constraint" or further restriction on allowable values of the corresponding type.

A Type Model is well-formed if its \(CNSTR\) is applicable to all values in its \(MD\). A Type Environment \(te\) is well-formed if all of the type models in \(\text{Ran} \, te\) are well-formed.

As a notational convenience, we will use \(CNSTR(MD)\) to refer to the set of all values from \(MD\) for which the predicate \(CNSTR\) is true. This set represents the collection of all legitimate values for a program object interpreted as belonging to a type associated with a given Type Model.

Because a type environment contains only modeling information and not any runtime value information, a type environment is its own signature.
An Intuitive Explanation of Type Models

As indicate by the name, a *Type Model* is a direct mathematical expression of the designer's conceptual model for a type. Effectively, any well-defined set of values (any CPO) can serve as the basis for such a conceptual model. The designer then has the option of further constraining this set to indicate which values are allowable or meaningful for program objects of a given type. The ability to express such a constraint is not strictly needed for expressiveness, but does allow designers to provide simpler conceptual models.

In *Partial_Map_Facility* (Figure 7, p. 43), the exported type *Partial_Map* is the only item in its type environment. This type environment would map the name from $\mathcal{N}$ associated with the program identifier *Partial_Map* to a type model formalizing the contents of Figure 8 (p. 44).

### 4.5.2 Variable Environments ($\mathcal{VE}$)

A *Variable Environment* is a name-to-value mapping from $\mathcal{N}$ to *Variable Models*. A *Variable Model* is a two-tuple $(\mathcal{VSIG}, \mathcal{VAL})$, where:

- $\mathcal{VSIG}$ is a *Variable Signature*.

- $\mathcal{VAL}$ is some value from the domain of the type named by $TY$.

A *Variable Signature* is simply a type name from $\mathcal{N}$.

A *Variable Environment* $ve$ is well-formed with respect to a concrete instance $E$ when all of the variable models in $\text{Ran} ve$ are well-formed with respect to $E$. A *Variable Model* $vm$ is well-formed with respect to a concrete instance $E$ if:
1. $vm.VSIG$ is well-formed with respect to $E$.

2. $vm.VALE \in tm.CNSTR(tm.MD)$, where $tm \equiv \text{TypeModel}(vm.VSIG, E)$. This means that the variable's value comes from the mathematical domain associated with the model of its type and satisfies the constraint associated with that model.

A Variable Signature $VSIG$ is well-formed with respect to an (abstract or concrete) subsystem $S$ if $VSIG \in \text{TypeNames} S$, meaning the variable's type is defined in $S$ or in some subsystem nested in $S$.

A Variable Environment $ve$ is well-formed with respect to a specification adornment environment $SAE$ and the subsystem $S$, either an abstract or concrete instance it is contained in, when all of the variable models in Ran $ve$ are well-formed with respect to $SAE$ and $S$. A Variable Model $vm$ is well-formed with respect to a specification adornment environment $SAE$ and a subsystem $S$ if:

1. $vm.VSIG$ is well-formed with respect to $SAE$.

2. $vm.VALE \in tm.CNSTR(tm.MD)$, where $tm \equiv \text{TypeModel}(vm.VSIG, SAE)$.

A Variable Signature $VSIG$ is well-formed with respect to a specification adornment environment $SAE$ if $VSIG \in \text{SATypeNames} SAE$.

An Intuitive Explanation of Variable Models

A Variable Model, just like a Type Model, is a mathematical expression of the designer's conceptual model for a variable. It encompasses both the type and the value
of the variable. The variable environment includes all of the variables provided by a
subsystem (not including variables provided by nested subsystems).

4.5.3 Operation Environments (OE)

An Operation Environment is a name-to-value mapping from N to Operation Mean­
ings. An operation environment oe is well-formed with respect to a concrete instance
E when all of the operation meanings in Ran oe are well-formed with respect to E.

Operation Meanings (OM)

An Operation Meaning is a three-tuple om = (OMOD, PR, SF), where:

OMOD is an Operation Model (Section 4.5.3, p. 99) describing the parameter profile
and abstract behavior of the operation.

PR is a "procedure relation," a relation over the input arguments, the containing con­
crete instance before execution, the output values, and the containing concrete
instance after execution (Args (om.OMOD.PP) × E × Args (om.OMOD.PP) ×
E → B), which represents the actual relation computed by the operation.

SF is a "status function," an Assert Status-valued function over the input arguments
and the containing concrete instance (Args (om.OMOD.PP) × E → AS), rep­
resenting the assert status computed by the operation.

An Operation Meaning om is well-formed with respect to an environment E when
the following conditions are met:

1. om.OMOD is well-formed with respect to E.
2. For all input arguments and concrete instance values satisfying \( om.OMOD.DP \), all allowable output arguments related to these inputs by \( om.PR \) must also satisfy \( om.OMOD.EP \). In other words, the actual relation computed by the operation must be consistent with the postcondition in the operation signature.

3. For all input arguments and concrete instance values satisfying \( om.OMOD.DP \), \( om.SF \) must not produce the value \( CF \).

**An Intuitive Explanation of Operation Meanings**

An *Operation Meaning* records the designer’s conceptual model of a program operation as an operation model, described below. In addition, an operation meaning describes in \( PR \) the actual relation computed by a program operation for use in the run-time semantics. In simple terms, one can imagine \( PR \) as just a relation from input values to output values. However, it is defined to be slightly richer, so that the entire environment \( E \) containing a particular operation is also part of the space on which we define the relation \( PR \). This allows ACTI to capture program operations that affect the visible state of a subsystem, by making the subsystem itself an explicit parameter to every program operation.

The \( SF \) component of an operation meaning is similar to the \( PR \) component, but instead of computing the output values of the corresponding program operation given the input values, it computes the “assert status” value that results from executing the operation. This is important for supporting assertive programming (Section 4.1.6, p. 81).
Operation Models (OMOD)

An Operation Model is a three-tuple \( (PP, DP, EP) \), where:

\( PP \) is a Parameter Profile describing the number and types of parameters that must be passed into an operation conforming to this signature.

\( DP \) is a "domain predicate," a boolean-valued function over the arguments to the operation and the containing concrete instance \( (\text{Args}(PP) \times E \rightarrow B) \) that represents the precondition for the operation.

\( EP \) is an "effect predicate," a boolean-valued function over the input arguments, the containing concrete instance before execution, the output values, and the containing concrete instance after execution \( (\text{Args}(PP) \times E \times \text{Args}(PP) \times E \rightarrow B) \), that represents the postcondition for the operation.

An Operation Model \( omod \) is well-formed with respect to a subsystem \( S \) (either an abstract or concrete instance) when \( omod_PP \) is well-formed with respect to \( S \).

An Intuitive Explanation of Operation Models

An Operation Model records the designer’s conceptual model of a program operation. This includes both the parameter profile of the operation, represented in \( PP \), and an abstract description of the operation’s behavior. This behavioral description is recorded as separate precondition and postcondition assertions. The precondition is represented by \( DP \), the “domain predicate” that is true for input values that meet the precondition it defines. The postcondition is represented by \( EP \), the “effect predicate”
that, given a particular set of input values, is true for output values that meet the postcondition it defines.

Note that both DP and EP are defined so that the (abstract or concrete) instance containing this particular operation signature is an explicit parameter accounted for in both the pre- and postconditions. This allows operation models to completely encompass the behavior of operations that depend on or affect visible state variables, either in the subsystem where the operation is defined, or in that subsystem's explicitly declared context. Often, programmers loosely think of such state variables as "global variables."

**Parameter Profiles (PP)**

A *Parameter Profile* is a string of identifiers from $\mathcal{N}$. A string is simply a finite sequence of elements, which will be written as $(t_1, t_2, \ldots, t_n)$. The empty string will be written as $\epsilon$. For a thorough treatment of basic string theory, refer to an introductory book on discrete mathematics [22, pp. 251–264] or the theory of computation [39, pp. 29–33].

Since the space of Parameter Profiles is a CPO, it must have an ordering relation. For strings, we will use the relation $\sqsubseteq$ defined as follows:

$$\forall \text{ strings } a, b : a \sqsubseteq b \iff a \text{ is a prefix of } b \quad (4.7)$$

Thus, $\epsilon$ is the least element ($\bot$) for a string CPO using this ordering relation.
A Parameter Profile $pp$ is well-formed with respect to a subsystem $S$ (either an abstract or concrete instance) when all of the identifiers appearing in $pp$ are elements of $\text{TypeNames } S$.

For a parameter profile $pp = (t_1, t_2, \ldots, t_n)$ that is well-formed with respect to a subsystem $S$, let $\langle MD_i, C_i \rangle = \text{TypeModel} (t_i, S)$, the Type Model associated with $t_i$. An argument tuple conforming to $pp$ is an element of the cartesian product:

$$C_1(MD_1) \times C_2(MD_2) \times \ldots \times C_n(MD_n)$$

We will use $\text{Args}(pp)$ as shorthand for this cartesian product, the set of all argument tuples conforming to the parameter profile $pp$.

An Intuitive Explanation of Parameter Profiles

A Parameter Profile describes the parameter signature of an operation. For a parameter profile $\langle t_1, t_2, \ldots, t_n \rangle$, $t_i$ is the name of the type of the operations $i$th argument.

This part of the ACTI model is common among virtually all modern imperative languages.

4.5.4 Environment Invariants ($\mathcal{EI}$)

An Environment Invariant is a boolean-valued function over an Environment (Section 4.5, p. 89). All environment invariants are well-formed.

4.5.5 Abstract Instance Environments ($\mathcal{AIE}$)

An Abstract Instance Environment is a name-to-value mapping from $\mathcal{N}$ to Context-Bound Abstract Instances ($\mathcal{CBAI}$). An abstract instance environment $\text{aie}$ is well-
formed with respect to a subsystem $S$, either an environment or an abstract instance, if all of the context-bound abstract instances in $\text{Ran } aie$ are well-formed with respect to $S$.

A Context-Bound Abstract Instance is a two-tuple $(AI, IM)$, where:

- $AI$ is an Abstract Instance (Section 4.6, p. 107).
- $IM$ is an Interpretation Mapping (Section 4.10, p. 126).

A context-bound abstract instance $cbai$ is well-formed with respect to a subsystem $S$ if the following conditions hold:

1. Both $cbai.AI$ and $cbai.IM$ are well-formed.

2. $IM$ can be used to interpret $S$ as fulfilling the context required by $cbai.AI$. Formally, if $S$ is an abstract instance, then $cbai.IM.DP(S, cbai.AI.CTXT)$ must be true; if $S$ is a concrete instance, then $cbai.IM.DP(\text{abstract}(S), cbai.AI.CTXT)$ must be true.

The operators TypeNames, VariableNames, OperationNames, TypeModel, VariableModel, and OperationModel introduced on page 91 are all defined on abstract instance environments. The definition for TypeNames is:

$$\text{TypeNames } aie = \bigcup_{cbai \in \text{Dom } aie} \text{TypeNames } cbai.AI \quad (4.9)$$

The definitions of VariableNames and OperationNames are similar. The definition for TypeModel $(t, aie)$ is:

$$\text{TypeModel } (t, aie) = \begin{cases} 
\text{TypeModel } (t, AI) & \text{if } \exists AI \in \text{Ran } aie \text{ where } \ t \in \text{TypeNames } AI \\
\bot & \text{otherwise}
\end{cases} \quad (4.10)$$
The definitions of VariableModel and OperationModel are similar.

### An Intuitive Explanation of Abstract Instance Environments

An *Abstract Instance Environment*, say *aie*, is just a collection of named (abstract) subsystems that are contained within some other (abstract or concrete) subsystem, say *S*. Each *Context-Bound Abstract Instance cbai* within the abstract instance environment represents a single nested subsystem within *S*. See Section 4.13.6, page 146, for more information about binding the context of a subsystem.

Since *S* "contains" every context-bound abstract instance in *aie*, it really forms the environment, or "context," in which those instances live. Thus, each *cbai* contains not only a smaller abstract instance (*cbai.AI*), but also an explanation for how the outer containing environment *S* meets the context requirements of that inner instance. This binding between the containing environment and the explicit context interface of the abstract instance *cbai.AI* is expressed as an *Interpretation Mapping*, which explains how to "interpret" *S* as fulfilling the context interface of *cbai.AI*. As a result, *cbai.AI* is no longer independent of the environment in which it lives—its context is *bound* to the containing subsystem *S*. This is the meaning behind the name "context-bound."

In order for a context-bound abstract instance *cbai* to be well-formed, both of its subcomponents must be well-formed. Further, the interpretation mapping contained in *cbai* must be applicable to both the containing subsystem *S* and the nested abstract instance *cbai.AI*. This is why one cannot tell whether a context-bound abstract instance is well-formed in isolation; this property is always relative to the contain-
ing (abstract or concrete) subsystem \( S \). This is what is formally expressed in the requirement (2) for well-formed context-bound abstract instances.

### 4.5.6 Concrete Instance Environments (CIE)

A *Concrete Instance Environment* is a name-to-value mapping from \( N \) to *Context-Bound Concrete Instances* (CBCI). A concrete instance environment \( \text{cie} \) is well-formed with respect to a concrete instance \( E \) if all of the context-bound concrete instances in \( \text{Ran cie} \) are well-formed with respect to \( E \).

A *Context-Bound Concrete Instance* is a two-tuple \( (CI, IM) \), where:

- \( CI \) is an *Concrete Instance* (Section 4.5, p. 89).
- \( IM \) is an *Interpretation Mapping* (Section 4.10, p. 126).

A context-bound concrete instance \( cbci \) is well-formed with respect to a concrete instance \( E \) if the following conditions hold:

1. Both \( cbci.CI \) and \( cbci.IM \) are well-formed.
2. \( cbci.IM.DP(\text{abstract}(E), cbci.CI.CTXT) \) is true; that is, \( IM \) can be used to interpret \( E \) as fulfilling the context required by \( cbci.CI \).

The operators *TypeNames*, *VariableNames*, and *OperationNames* introduced on page 91 are all defined on concrete instance environments. The definition for *TypeNames* is:

\[
\text{TypeNames cie} = \bigcup_{cbci \in \text{Dom cie}} \text{TypeNames cbci.CI} \tag{4.11}
\]
The definitions of VariableNames and OperationNames are similar. The definition for \( \text{TypeModel} (t, \text{cie}) \) is:

\[
\text{TypeModel} (t, \text{cie}) = \begin{cases} 
\text{TypeModel} (t, CI) & \text{if } \exists CI \in \text{Ran cie where } t \in \text{TypeNames } CI \\
\bot & \text{otherwise}
\end{cases} 
\]  

The definitions of VariableModel and OperationModel are similar.

An Intuitive Explanation of Concrete Instance Environments

The structure and meaning of a Concrete Instance Environment exactly parallel that of an Abstract Instance Environment. There are only two critical differences: a Concrete Instance Environment contains concrete, rather than abstract, instances; and a Concrete Instance Environment can only appear in a concrete instance. Since abstract instances contain no implementation details, their CIE component is really just another Abstract Instance Environment.

A Concrete Instance Environment, say \( \text{cie} \), is just a collection of named concrete instances that are contained within some other concrete instance, say \( E \). Each Context-Bound Concrete Instance \( \text{cbci} \) within the concrete instance environment represents a single nested subsystem within \( E \).

Just as with an Abstract Instance Environment, since \( E \) “contains” every context-bound concrete instance in \( \text{cie} \), it forms the environment, or “context,” in which those instances live. Thus, each \( \text{cbci} \) contains not only a smaller concrete instance \( (\text{cbci}.CI) \), but also an explanation for how the outer containing environment \( E \) meets the context requirements of that inner instance. This binding between the containing environment and the explicit context interface of the concrete instance \( \text{cbci}.CI \) is expressed as an
Interpretation Mapping, which explains how to "interpret" $E$ as fulfilling the context interface of $cbci.CI$. Now, $cbci.CI$ is no longer independent of the environment in which it lives; its context is bound to the containing concrete instance $E$.

The requirements for well-formed context-bound concrete instances are exactly analogous to those for context-bound abstract instances.

4.5.7 Interpretation Mapping Environments ($\text{IME}$)

An Interpretation Mapping Environment is a name-to-value mapping from $\mathcal{N}$ to Interpretation Mappings (Section 4.10, p. 126). An Interpretation Mapping Environment $\text{ime}$ is well-formed when all of the interpretation mappings in $\text{Ran}\ ime$ are well-formed.

4.5.8 Abstract Template Environments ($\text{ATE}$)

An Abstract Template Environment is a name-to-value mapping from $\mathcal{N}$ to Abstract Templates (Section 4.8, p. 116). An Abstract Template Environment $\text{ate}$ is well-formed when all of the abstract templates in $\text{Ran}\ ate$ are well-formed.

4.5.9 Concrete Template Environments ($\text{CTE}$)

A Concrete Template Environment is a name-to-value mapping from $\mathcal{N}$ to Concrete Templates (Section 4.7, p. 113). A Concrete Template Environment $\text{cte}$ is well-formed when all of the concrete templates in $\text{Ran}\ cte$ are well-formed.
4.5.10 Interpretation Mapping Template Environments (IMTE)

An Interpretation Mapping Template Environment is a name-to-value mapping from \(\mathcal{N}\) to Interpretation Mapping Templates (Section 4.11, p. 136). An Interpretation Mapping Template Environment \(\text{imte}\) is well-formed when all of the interpretation mapping templates in \(\text{Ran} \text{imte}\) are well-formed.

4.6 Abstract Instances (\(\mathcal{AI}\))

The space of Abstract Instances is a cartesian product:

\[
\mathcal{AI} = \mathcal{AI} \times \mathcal{SAE} \times \mathcal{TE} \times \mathcal{VES} \times \mathcal{OES} \times \mathcal{EI} \times \mathcal{AIE} \times \mathcal{AIE} \times \mathcal{IME} \times \\
\mathcal{ATES} \times \mathcal{CTES} \times \mathcal{IMTE} \\
\text{(4.13)}
\]

Thus, an Abstract Instance is a 12-tuple \(\mathcal{AI} = (\mathcal{CTXT}, \mathcal{SAE}, \mathcal{TE}, \mathcal{VES}, \mathcal{OES}, \mathcal{EI}, \mathcal{AIE}, \mathcal{CIE}, \mathcal{IME}, \mathcal{ATES}, \mathcal{CTES}, \mathcal{IMTE})\), where:

\(\mathcal{CTXT}\) is an Abstract Instance declaring all external references appearing in this abstract instance. Here, it is worth pointing out that since the space of Abstract Instances is a CPO, this component may be \(\bot\), indicating there are no external dependencies.

\(\mathcal{SAE}\) is a Specification Adornment Environment (Section 4.9, p. 118) containing all of the math definitions provided by \(\mathcal{AI}\).

\(\mathcal{TE}\) is a Type Environment (Section 4.5.1, p. 94) containing all of the types provided by \(\mathcal{AI}\).
VES is a Variable Environment Signature (Section 4.6.1, p. 111) containing all of the variables provided by AI.

OES is an Operation Environment Signature (Section 4.6.2, p. 112) containing all of the operations provided by AI.

EI is an Environment Invariant (Section 4.5.4, p. 101) that always holds for any Environment that can be interpreted as satisfying AI.

AIE is an Abstract Instance Environment (Section 4.5.5, p. 101) containing all of the Abstract Instances provided by AI.

CIE is another Abstract Instance Environment containing Abstract Instances describing the sub-environments contained in AI.

IME is an Interpretation Mapping Environment (Section 4.5.7, p. 106) containing all of the Interpretation Mappings provided by AI.

ATES is an Abstract Template Environment Signature (Section 4.6.3, p. 112) containing all of the Abstract Template Signatures provided by AI.

CTES is a Concrete Template Environment Signature containing all of the Concrete Template Signatures provided by AI.

IMTE is an Interpretation Mapping Template Environment (Section 4.5.10, p. 107) containing all of the Interpretation Mapping Templates (Section 4.11, p. 136) provided by AI.
We can say an abstract instance $AI_1$ directly contains another abstract instance $AI_2$ if $AI_1.CTXT = AI_2$, if $AI_2 \in \text{Ran} AI_1.AIE$, or if $AI_2 \in \text{Ran} AI_1.CIE$. We can further say that $AI_1$ indirectly contains $AI_2$ if any environment in the set $\{AI_1.CTXT\} \cup \text{Ran} AI_1.AIE \cup \text{Ran} AI_1.CIE$ contains $AI_2$ (directly or indirectly). An abstract instance $AI$ is cycle-free if it does not contain itself, either directly or indirectly.

An Abstract Instance $AI$ is well-formed if it is cycle-free, and each of its twelve components is well-formed with respect to $AI$.

Just as with concrete instances, for notational convenience we will use $\text{Dom} AI$ as a shorthand for the union of the meaningful domains of the ten name-to-value mappings contained in $AI$. TypeNames, introduced on page 91, will be defined in the obvious way:

$$\text{TypeNames } AI = \text{Dom } AI.TE \cup \text{TypeNames } AI.CIE \cup \text{TypeNames } AI.CTXT$$

(4.14)

VariableNames and OperationNames also have parallel definitions. LocalTypeNames, LocalVariableNames, and LocalOperationNames are also applicable to abstract instances, with the same definitions as for concrete instances. The definition of TypeModel $(t, AI)$ is:

$$\text{TypeModel } (t, AI) = \begin{cases} 
AI.TE(t) & \text{if } t \in \text{Dom } AI.TE \\
\text{TypeModel } (t, AI.CIE) & \text{if } t \in \text{TypeNames } AI.CIE \\
\text{TypeModel } (t, AI.CTXT) & \text{if } t \in \text{TypeNames } AI.CTXT 
\end{cases}$$

(4.15)

VariableModel and OperationModel also have parallel definitions.
An Intuitive Explanation of Abstract Instances

An Abstract Instance is the run-time denotation of a subsystem (or module) specification. Its structure exactly mirrors that of a Concrete Instance, but while a concrete instance contains the designer's conceptual models for objects and corresponding runtime values, an abstract instance contains only the designer's conceptual models for the objects it contains. Usually, such a conceptual model is called a "signature," since it defines the conceptual shape of a program object without defining its actual value.

In Section 3.4, both Partial_Map_Facility and Communal_PMap_Facility were specifications. In ACTI, the natural denotation for such a programming construct is an abstract instance.

Just as in a concrete instance, the CTXT component of an abstract instance AI is an explicit declaration of all entities outside AI that are referred to directly in the signatures of any objects within AI. Effectively, CTXT provides names and abstract specifications for the external objects needed to fully define AI. Items within AI can only "see" the outside world through this explicit interface.

The SAE component of AI collects all of the specification adornment definitions present in the abstract instance (see Section 3.4.5). TE, VES, OES, AIE, CIE, IME, ATES, CTES, and IMTE represent collections of sub-objects within AI of various kinds, and are all the direct counterparts of the similarly named subcomponents of a concrete instance. Now, however, since AI does not contain any run-time values for variables, operations, or nested subsystems, several of these components hold only
signatures. Note that despite its name, CIE holds an Abstract Instance Environment. This is because abstract instances play the role of "signatures" for concrete instances.

Finally, the predicate $EI$ is a formal representation of a subsystem-level invariant which must always hold for any concrete instance that satisfies the abstract description $AI$. In no sense can $EI$ "hold" for $AI$, since $EI$ is a predicate over the space of Concrete Instances. Further, since $AI$ contains only abstract models of program entities, not their actual run-time values, it would not make sense to ask whether or not it satisfies such an invariant.

4.6.1 Variable Environment Signatures ($\mathcal{VES}$)

A Variable Environment Signature is a function from $\mathcal{N}$ to $\mathcal{N}$. A Variable Environment Signature $ves$ is well-formed with respect to an abstract instance $AI$ if every identifier in $\text{Ran} \ ves$ is in $\text{TypeNames} \ AI$.

An Intuitive Explanation of Variable Environment Signatures

A Variable Environment Signature is the "signature" of a Variable Environment. It gives the type associated with each variable name defined in the environment, but leaves out the value information. One can think of it as performing the same function as a plain Variable Environment, but instead of mapping each variable identifier to a complete Variable Model (Section 4.5.2, p. 95), it maps each variable identifier to just the projection of the $TY$ component of its Variable Model.
4.6.2 Operation Environment Signatures (OES)

An Operation Environment Signature is a function from $\mathcal{N}$ to Operation Models. An Operation Environment Signature $oes$ is well-formed with respect to an abstract instance $AI$ if every operation signature in Ran $oes$ is well-formed with respect to $AI$.

An Intuitive Explanation of Operation Environment Signatures

An Operation Environment Signature is the "signature" of an Operation Environment. It gives the pre- and postcondition information associated with each operation name defined in the abstract instance, but leaves out information about the actual relation and status function computed by the operation.

In Partial.Map.Facility (Figure 7, p. 43), the exported operations Define, Undefine, Undefine.Any.One, Is_DEFINED, Size, Initialize, Finalize, and Swap make up the meaningful domain of its operation environment signature. For example, this operation environment signature would map the name from $\mathcal{N}$ associated with the program identifier Define to an operation model formalizing the contents of Figure 9 (p. 45).

4.6.3 Abstract Template Environment Signatures (ATES)

An Abstract Template Environment Signature is a function from $\mathcal{N}$ to Abstract Template Signatures (Section 4.8.1, p. 117). An Abstract Template Environment Signature $ates$ is well-formed with respect to an abstract instance $AI$ if every abstract template signature in Ran $ates$ is well-formed with respect to $AI$. 
An Intuitive Explanation of Abstract Template Environment Signatures

An *Abstract Template Environment Signature* is the "signature" of an *Abstract Template Environment*. It gives the pre- and postcondition information associated with each abstract template name defined in the abstract instance, but leaves out information about the actual instance-to-instance function and status function computed by the template when it is applied.

4.6.4 Concrete Template Environment Signatures (*CTES*)

A *Concrete Template Environment Signature* is a function from $N$ to *Concrete Template Signatures* (Section 4.7.1, p. 115). A *Concrete Template Environment Signature* $ctes$ is well-formed with respect to an abstract instance $AI$ if every concrete template signature in $\text{Ran} ctes$ is well-formed with respect to $AI$.

An Intuitive Explanation of Concrete Template Environment Signatures

A *Concrete Template Environment Signature* is the "signature" of an *Concrete Template Environment*. It gives the pre- and postcondition information associated with each concrete template name defined in the abstract instance, but leaves out information about the actual instance-to-instance function and status function computed by the template when it is applied.

4.7 Concrete Templates (*CT*)

A *Concrete Template* is a three-tuple $\langle CTS, CICIF, SF \rangle$, where:

$CT$ is a *Concrete Template Signature*, defined below.
114

*CICIF* is the “concrete instance-to-concrete instance function” \((E \rightarrow E)\) actually computed by instantiating the template.

*SF* is a “status function,” an Assert Status-valued function over the input concrete instance \((E \rightarrow AS)\) representing the assert status actually computed by instantiating the template.

A *Concrete Template* *ct* is well-formed when the following two conditions are met:

1. For all input concrete instance values *E* satisfying \(ct.CTS.CIDP\), the effect predicate \(ct.CTS.CIEP(E, ct.CICIF(E))\) must be true. In other words, the actual instance-to-instance function computed by the template must be consistent with the postcondition described by \(ct.CTS.CIEP\).

2. For all input concrete instance values *E* satisfying \(ct.CTS.CIDP\), \(ct.SF(E)\) must not produce the value \(CF\).

**An Intuitive Explanation of Concrete Templates**

A *Concrete Template* is a generic subsystem realization, which should be thought of as a function from concrete instances to concrete instances. The characteristics of this function are captured in the three-tuple described above. The structure of a template mirrors that of an *Operation Meaning* (Section 4.5.3, p. 97): *CTS* is the signature, which describes both the precondition for applying the template, and the (relational) postcondition describing the effects of applying the template; and *CICIF* and *SF* describe the actual computation carried out when the template is applied. For
more information on applying template to generate new instances, see Section 4.13.7, page 148.

Note that a conceptual template only has a single concrete instance argument. This is all that is needed, since a given concrete instance can contain many nested concrete instances as subcomponents.

Further, notice that the acceptable concrete instances to which this template may be applied are defined by the precondition $CIDP$, not simply by conformance with some particular abstract instance. As a result, it is possible to construct templates in ACTI that can be applied to several structurally different concrete instances, generating an appropriate new instance for each one. The most interesting application of this feature is in generalizing the notion of generics to encompass a variable number of parameters.

With the appropriate precondition, an ACTI concrete template could be constructed that could be applied to a concrete instance containing one, two, or $n$ nested subsystems. For each of these possible inputs, this template could generate an appropriate output instance. Similarly, generics that build on a variable number of type parameters or operation parameters can also be defined.

### 4.7.1 Concrete Template Signatures (CTS)

A *Concrete Template Signature* is a two-tuple $(CIDP, CIEP)$, where:

$CIDP$ is a "concrete instance domain predicate," a boolean-valued function over the input concrete instance ($E \rightarrow B$) that represents the precondition for the template.
$CIEP$ is a “concrete instance effect predicate,” a boolean-valued function over the input and output concrete instances $(\mathcal{E} \times \mathcal{E} \rightarrow \mathcal{B})$ that represents the postcondition of instantiating the template.

All concrete template signatures are well-formed.

### 4.8 Abstract Templates ($\mathcal{AT}$)

An *Abstract Template* is a three-tuple $(\mathcal{ATS}, \mathcal{AIAIF}, \mathcal{SF})$, where:

- $\mathcal{ATS}$ is an *Abstract Template Signature*, defined below.
- $\mathcal{AIAIF}$ is the “abstract instance-to-abstract instance function” $(\mathcal{AI} \rightarrow \mathcal{AI})$ actually computed by instantiating the template.
- $\mathcal{SF}$ is a “status function,” an *Assert Status*-valued function over the input abstract instance $(\mathcal{AI} \rightarrow \mathcal{AS})$ representing the assert status actually computed by instantiating the template.

An *Abstract Template* at is well-formed when the following two conditions are met:

1. For all input abstract instance values $\mathcal{AI}$ satisfying $at.\mathcal{ATS}.AIDP$, the effect predicate $at.\mathcal{ATS}.AIEP(\mathcal{AI}, at.\mathcal{AIAIF}(\mathcal{AI}))$ must be true. In other words, the actual instance-to-instance function computed by the template must be consistent with the postcondition described by $at.\mathcal{ATS}.AIEP$.

2. For all input abstract instance values $\mathcal{AI}$ satisfying $at.\mathcal{ATS}.AIDP$, $at.\mathcal{SF}(\mathcal{AI})$ must not produce the value $CF$. 
An Intuitive Explanation of Abstract Templates

An Abstract Template is a generic module specification, which should be thought of as a function from abstract instances to abstract instances. Its structure exactly parallels that of a Concrete Template. Just like a concrete template, the nested structure of the single abstract instance parameter allows an abstract template effectively to be parameterized by many instances. Because the requirements on the single parameter are expressed through a predicate, that parameter may contain variable numbers of nested instances, allowing variable-length generic parameter lists.

In Section 3.4.6, Communal_PMap_Template was a parameterized component specification. In ACTI, the natural denotation for such a programming construct is an abstract template.

4.8.1 Abstract Template Signatures (ATS)

An Abstract Template Signature is a two-tuple \( (AIDP, AIEP) \), where:

\( AIDP \) is an "abstract instance domain predicate," a boolean-valued function over the input abstract instance \( (AI \rightarrow B) \) that represents the precondition for the template.

\( AIEP \) is an "abstract instance effect predicate," a boolean-valued function over the input and output abstract instances \( (AI \times AI \rightarrow B) \) that represents the post-condition of instantiating the template.

All abstract template signatures are well-formed.
4.9 Specification Adornment Environments (\(SAE\))

A Specification Adornment Environment (or Specification Adornment Instance) is a five-tuple \(SAE = (SATY, SAVE, SAIE, SATE)\), where:

\(SATY\) is a Type Environment (Section 4.5.1, p. 94) containing all of the types provided by \(SAE\).

\(SAVE\) is a Specification Adornment Variable Environment (Section 4.9.1, p. 121) containing all of the variables provided by \(SAE\).

\(SAEI\) is a Specification Adornment Environment Invariant (Section 4.9.2, p. 122) that always holds for \(SAE\).

\(SAIE\) is a Specification Adornment Instance Environment (Section 4.9.3, p. 123) containing all of the nested Specification Adornment Instances provided by \(SAE\).

\(SATE\) is a Specification Adornment Template Environment (Section 4.9.4, p. 123) containing all of the Specification Adornment Templates (or theories) provided by \(SAE\).

A Specification Adornment Environment \(SAE\) is well-formed with respect to a subsystem \(S\), either an abstract or concrete instance, if each of its six components is well-formed with respect to \(S\) and \(SAE.SAEI(SAE)\) is true (i.e., the invariant within \(SAE\) holds).

Just as we defined the operator TypeNames for abstract and concrete instances, we will use SATypeNames \(SAE\) to refer to the set of all identifiers from \(N\) that are
mapped to a *Type Model* by *SAE* or some specification adornment environment nested within *SAE*:

\[ \text{SATypeNames } S\text{AE } = \text{ Dom } S\text{AE.SATY } \cup \text{ SATypeNames } S\text{AE-SAIE} \quad (4.16) \]

SATypeNames is an overloaded set of operators that are mutually defined: one for specification adornment instances, one for specification adornment instance environments, one for abstract instances, and one for concrete instances. The definition of SATypeNames when applied to specification adornment instance environments is given in Section 4.9.3, page 123. For a subsystem \( S \) that is either an abstract instance or a concrete instance:

\[ \text{SATypeNames } S \equiv \text{ SATypeNames } S.S\text{AE } \cup \text{ SATypeNames } S.C\text{TXT} \quad (4.17) \]

We will consider SAVariableNames to be defined similarly.

In addition, we will use SATypeModel \((t, S\text{AE})\) as shorthand for the type model associated with a type name \( t \) defined somewhere in \( S\text{AE} \):

\[ \text{SATypeModel } (t, S\text{AE}) = \begin{cases} S\text{AE.SATY}(t) & \text{if } t \in \text{ Dom } S\text{AE.SATY} \\ \text{SATypeModel } (t, S\text{AE-SAIE}) & \text{if } t \in \text{ SATypeNames } S\text{AE-SAIE} \end{cases} \quad (4.18) \]

The notation SATypeModel \((t, S)\), where \( S \) is an abstract or concrete instance, will be used with the following meaning:

\[ \text{SATypeModel } (t, S) = \begin{cases} \text{SATypeModel } (t, S.S\text{AE}) & \text{if } t \in \text{ SATypeNames } S.S\text{AE} \\ \text{SATypeModel } (t, S.C\text{TXT}) & \text{if } t \in \text{ SATypeNames } S.C\text{TXT} \end{cases} \quad (4.19) \]

The operation SAVariableModel on specification adornment environments, abstract instances, and concrete instances is defined in the same way.
An Intuitive Explanation of Specification Adornment Environments

"Specification adornments" are auxiliary definitions that a subsystem designer creates to make it easier to describe the conceptual models of program objects. Basically, these definitions are purely mathematical constructs that can be used as building blocks in describing any Type Model, Operation Model, Environment Invariant, or other subsystem structure. These definitions are called specification adornments because they do not describe any of the actual behavior of a subsystem, but are merely "adornments" that make it easier to write down a specification of that behavior.

A Specification Adornment Environment is a module-like structure for organizing a collection of specification adornment definitions. These definitions may include type models, variable models whose values are constrained by an invariant, or nested specification adornment environments or templates. Note that mathematical functions or other structures useful in defining predicates, invariants, preconditions, and so on, can easily be defined as particular variables within the specification adornment environment simply by ensuring that the variable has a type with the appropriate mathematical domain in its type model.

The second version of Partial.Map.Facility presented in Section 3.4.5 defines the math type PARTIAL_FUNCTION and the math operation DEFINED_IN in its local context (Figure 14, p. 53). In ACTI, these definitions would make up the specification adornment environment in the abstract instance denoted by Partial.Map.Facility (Figure 17, p. 56).
4.9.1 Specification Adornment Variable Environments (SAVE)

A Specification Adornment Variable Environment is simply a Variable Environment. The criteria for well-formedness are different for specification adornment variable environments, however.

A Specification Adornment Variable Environment save is well-formed with respect to a specification adornment environment SAE and the subsystem S, either an abstract or concrete instance, when all of the variable models in Ran save are well-formed with respect to SAE and S. A Variable Model vm is well-formed with respect to a specification adornment environment SAE and a subsystem S if:

1. vm.VSIG is well-formed with respect to SAE and S.

2. vm.VAL ∈ tm.CNSTR(tm.MD), where tm ≡ SATypeModel (vm.VSIG, S).

A Variable Signature VSIG is well-formed with respect to SAE and S if VSIG ∈ SATypeNames S.

An Intuitive Explanation of Specification Adornment Variable Environments

A specification adornment variable environment save associates names with variable models, just as a variable environment does. The only difference is that the variables represented by associations in save are present for specification purposes only. As a result, the types of these variables are limited to specification adornment types by the rules for well-formedness given above.
Also, it is interesting to note that specification adornment variable environments can also be used to model specification-level operations as well as specification-level variables. A mathematical operation being used for specification purposes can be represented as a specification adornment variable whose type is the appropriate space of functions. As a result, there is no need for a "specification adornment operation environment."

In the Communal_PMap_Facility introduced in Section 3.4.5, total.size is an example of a specification adornment variable. The specification adornment variable environment in the abstract instance denoted by Communal_PMap_Facility would map the name from $\mathcal{N}$ associated with the program identifier total.size to an appropriate variable model (see Figure 20, p. 20). In addition, both of the math operations DEFINED_IN and MAX_TOTAL_LENGTH declared in Communal_PMap_Facility's local context would also be present in its specification adornment variable environment.

4.9.2 Specification Adornment Environment Invariants ($SAEI$)

A Specification Adornment Environment Invariant is a boolean-valued function over the space of Specification Adornment Environments (Section 4.9, p. 118). All specification adornment environment invariants are well-formed.
4.9.3 Specification Adornment Instance Environments
\((SAIE)\)

A Specification Adornment Instance Environment is a name-to-value mapping from \(N\) to Specification Adornment Environments. A Specification Adornment Instance Environment \(saie\) is well-formed when all of the specification adornment instances in \(\text{Ran } saie\) are well-formed.

The operators \(\text{SATypeNames}, \text{SAVariableNames}, \text{SATypeModel},\) and \(\text{SAVariableModel},\) introduced on page 118 are all defined on specification adornment instance environments. The definition for \(\text{SATypeNames}\) is:

\[
\text{SATypeNames } saie = \bigcup_{SAE \in \text{Dom } saie} \text{SATypeNames } SAE
\]

The definition of \(\text{SAVariableNames}\) is similar. The definition for \(\text{SATypeModel} (t, saie)\) is:

\[
\text{SATypeModel} (t, saie) = \begin{cases} 
\text{SATypeModel} (t, SAE) & \text{if } \exists SAE \in \text{Ran } saie \text{ where } t \in \text{SATypeNames } SAE \\
\bot & \text{otherwise}
\end{cases}
\]

The definition of \(\text{SAVariableModel}\) is similar.

4.9.4 Specification Adornment Template Environments
\((SAT\epsilon)\)

A Specification Adornment Template Environment is a name-to-value mapping from \(N\) to Specification Adornment Templates. A Specification Adornment Template Environment \(sate\) is well-formed when all of the specification adornment templates in \(\text{Ran } sate\) are well-formed.
4.9.5 Specification Adornment Templates \((SAT)\)

A Specification Adornment Template is a three-tuple \((ATS, AIAIF, SF)\), where:

\(SATS\) is a Specification Adornment Template Signature (Section 4.9.6, p. 125).

\(SAISAIF\) is the "specification adornment instance-to-specification adornment instance function" \((SAE \rightarrow SAE)\) actually computed by instantiating the template.

\(SF\) is a "status function," an Assert Status-valued function over the input specification adornment instance \((SAE \rightarrow AS)\) representing the assert status actually computed by instantiating the template.

A Specification Adornment Template \(sat\) is well-formed when the following two conditions are met:

1. For all input specification adornment environment values \(SAE\) satisfying the domain predicate \(sat.SATS.SAIDP, sat.SATS.SAIEP(SAE, sat.SAISAIF(SAE))\) must be true. In other words, the actual instance-to-instance function computed by the template must be consistent with the postcondition described by \(sat.SATS.SAIEP\).

2. For all input specification adornment instance values \(SAE\) satisfying the domain predicate \(sat.SATS.SAIDP, sat.SF(SAE)\) must not produce the value \(CF\).

An Intuitive Explanation of Specification Adornment Templates

A Specification Adornment Template is the natural analog of a concrete (or abstract) template in the world of specification adornment definitions. A Specification Adorn-
**ment Template** is a generic specification adornment module, which should be thought of as a function from specification adornment instances to specification adornment instances. Its structure exactly parallels that of a *Concrete Template* or an *Abstract Template*. Just like those two kinds of templates, the nested structure of the single specification adornment instance parameter allows a specification adornment template effectively to be parameterized by many instances. Because the requirements on the single parameter are expressed through a predicate, that parameter may contain a variable number of nested instances, allowing variable-length generic parameter lists.

### 4.9.6 Specification Adornment Template Signatures (**SATS**)  

A *Specification Adornment Template Signature* is a two-tuple \((S\text{AIDP}, S\text{AIEP})\), where:

\(S\text{AIDP}\) is a “specification adornment instance domain predicate,” a boolean-valued function over the input specification adornment instance \((S\text{AE} \rightarrow B)\) that represents the precondition for the template.

\(S\text{AIEP}\) is a “specification adornment instance effect predicate,” a boolean-valued function over the input and output specification adornment instances \((S\text{AE} \times S\text{AE} \rightarrow B)\) that represents the postcondition of instantiating the template.

All specification adornment template signatures are well-formed.
4.10 Interpretation Mappings ($\mathcal{IM}$)

An *Interpretation Mapping* between abstract instances describes how one can be interpreted as providing the behavior described by the other. For ease of communication, we will speak of an interpretation mapping as being *from* a more specific abstract instance, say $AI_S$, to a more general abstract instance, say $AI_G$, describing how $AI_S$ can be "interpreted as" providing behavior wholly consistent with $AI_G$. For an interpretation mapping $IM$, we can write this symbolically as:

$$IM \models AI_S \text{ is-interpretable-as } AI_G$$  \hspace{1cm} (4.22)

Such an interpretation mapping is a nine-tuple $IM = \langle DP, CTXTmap, TYmap, VARmap, OPmap, MCORR, MCORV, ATEmap, CTEmap \rangle$, where:

$DP$ is a "domain predicate," a boolean-valued function over both of the abstract instances $(AI \times AI \to B)$ which indicates the applicability of $IM$. For the remainder of this section, we assume without loss of generality that $AI_S$ and $AI_G$ are the two abstract instances under consideration and that $DP(AI_S, AI_G) = \text{true}$.

$CTXTmap$ is an *Interpretation Mapping*.

$TYmap$ is a *Type Interpretation Mapping* (Section 4.10.1, p. 129).

$VARmap$ is a *Variable Interpretation Mapping* (Section 4.10.2, p. 131).

$OPmap$ is an *Operation Interpretation Mapping* (Section 4.10.3, p. 131).
MCORR represents a "module-level correspondence," a relation over two abstract instances ($\mathcal{A}_I \times \mathcal{A}_I \rightarrow B$).

MCONV represents a "module-level convention," a predicate over the more specific abstract instance ($\mathcal{A}_I \rightarrow B$).

ATmap is an Abstract Template Interpretation Mapping (Section 4.10.4, p. 133).

CTmap is a Concrete Template Interpretation Mapping (Section 4.10.5, p. 135).

An Interpretation Mapping IM is well-formed if the following conditions are met for every pair of abstract instances $AI_S$, $AI_G$ that satisfy IM.DP:

1. All of the components of IM are well-formed with respect to $AI_S$, $AI_G$, and IM.

2. $IM.CTXTmap \models AI_S.CTXT$ is-interpretable-as $AI_G.CTXT$ (see the intuitive explanation for an explanation of this notation).

3. MCONV holds for $AI_S$. Furthermore, every operation in $AI_S$ that is "interpreted as" some operation in $AI_G$ must respect MCONV. More formally, the following assertion must be true:

$$\forall O \in \text{Ran } OPmap \cap \text{LocalOperationNames } AI_S : ( \exists OMOD \in OMOD : ( OMOD = \text{OperationModel}(O, AI_S) \text{ and } \forall \text{args}, \text{args}' \in \text{Args}(OMOD.PP) : ( \forall AI_{S'} \in AT : ( (OMOD.DP(\text{args}, AI_S) \text{ and } OMOD.EP(\text{args}, AI_S, \text{args}', AI_{S'})) \implies MCONV(AI_{S'}))))))$$ (4.23)
Figure 27: $IM$ Shows How to "Interpret" Specification $A$ as Satisfying Specification $B$

An Intuitive Explanation of Interpretation Mappings

An Interpretation Mapping is a correspondence between two abstract instances that shows how one of them can be "interpreted as" satisfying the abstract behavior described by the other. This relationship is illustrated in Figure 27, and is written symbolically as:

$$IM \models A \text{ is-interpretable-as } B$$

Thus, an interpretation mapping shows how one abstract instance ($A$) can be used in place of another ($B$), because the externally visible behavior the first describes is perfectly consistent with the behavior described by the second.

Since an interpretation mapping $IM$ shows how to interpret an $A$ as a $B$, it is natural to visualize the interpretation mapping as being "directed" from $A$ to $B$. This interpretation can be considered "lossy," since there may be many features provided by $A$ that are not needed in order to provide the (usually simpler, more abstract) behavior described by $B$. 
Note, however, that the name-to-value mappings defined within IM go in the opposite direction. These maps associate each name in B to the corresponding entity in A that is "interpreted as" it. This is a mathematical convenience that makes it easier to ensure that every entity is B has some analog in A. It also helps provide a place to hang additional information about properties of the interpretation (i.e., there is more to an interpretation map than just a one-to-one association of names between A and B).

For practical purposes, an interpretation mapping can be used to define a "subtype"-like relationship between subsystem specifications. Knowing that IM correctly explains how an A can be interpreted as a B means knowing that any place a subsystem satisfying specification B is required, one satisfying A will do. Thus, B is "more general" than A, or "more weakly specified" than A. Similarly, the class of concrete instances that provide the behavior described in A also necessarily provide the behavior described in B, so one can consider A "a (subsystem) subtype of" B.

4.10.1 Type Interpretation Mappings (TIM)

A Type Interpretation Mapping tim is a name-to-value mapping from N (type names in LocalTypeNames AIq) to three-tuples (N, R, C), where:

N is a name, presumably from LocalTypeNames AIq (e.g., a type defined in the more specific abstract instance).

R represents the "correspondence" between the two types. It is a relation over the math domains associated with the two type names, along with both AIq and
\( AI_S \) (for an identifier \( t \in \text{Ran \( tim \), } R \) is an element of TypeModel \((t, AIG).MD \times AI \times \text{TypeModel}(N, AIG).MDA \rightarrow B\)).

\( C \) represents the "convention" over the implementation type. It is a predicate over the math domain associated with \( N \) (e.g., TypeModel \((N, AIG).MD\)).

A Type Interpretation Mapping \( tim \) is well-formed with respect to an abstract instance \( AI_S \), and abstract instance \( AIG \), and an interpretation mapping \( IM \) if the following conditions hold:

1. The type interpretation mapping \( tim \) must completely cover the set of LocalTypeNames \( AIG \), and must use a non-\( \bot \) element from LocalTypeNames \( AIS \) as the interpretation for each entity in LocalTypeNames \( AIG \). More formally, for every such \( AIS \) and \( AIG \), \( tim \) must satisfy two conditions:

\[
\text{LocalTypeNames } AIG \subseteq \text{Dom } tim \\
\text{LocalTypeNames } AIS \supseteq \bigcup_{(N, R, C) \in \text{Ran } tim} \{N\}
\]

(4.25) (4.26)

Condition (4.25) requires that \( tim \) provide an interpretation for every type exported by \( AIG \). Condition (4.25) requires that each such interpretation be meaningful (i.e., non-\( \bot \)).

2. For every \((N, R, C) \in \text{Ran } tim\), every operation used in the interpretation respects the "type convention" \( C \) for all output values it produces of type \( N \). In other words, for every operation name \( O \in \text{Ran } IM.\text{OPmap} \) (i.e., operation in \( AIS \) being interpreted as some operation in \( AIG \)), the postcondition in the op-
eration model associated with $O$ by $AIs$ (OperationModel ($O, AIs$).EP) implies that $C$ holds for all output values from $O$ of type $N$.

4.10.2 Variable Interpretation Mappings ($\mathcal{VIM}$)

A Variable Interpretation Mapping is a name-to-value mapping from $\mathcal{N}$ (variable names in LocalVariableNames $AI_G$) to $\mathcal{N}$ (variable names in LocalVariableNames $AI_S$). A Variable Interpretation Mapping $vim$ is well-formed with respect to an abstract instance $AI_S$, and abstract instance $AI_G$, and an interpretation mapping $IM$ if for every variable name $N$ in LocalVariableNames $AI_G$, $T_N = \text{VariableModel}(N, AI_G)$ (a variable signature, which is a type name) is mapped to $T_{vim(N)}\text{VariableModel}(vim(N), AI_S)$ by $IM.TYmap$. In other words, $vim$ must respect the Type Interpretation Mapping in $IM$.

4.10.3 Operation Interpretation Mappings ($\mathcal{OIM}$)

An Operation Interpretation Mapping $oim$ is a name-to-value mapping from $\mathcal{N}$ to $\mathcal{N}$. $oim$ is intended to map operation names in LocalOperationNames $AI_G$ to (a subset of) the operation names in LocalOperationNames $AI_S$.

An Operation Interpretation Mapping is well-formed with respect to an abstract instance $AI_S$, and abstract instance $AI_G$, and an interpretation mapping $IM$ if the following conditions hold:

1. The operation interpretation mapping $oim$ must completely cover the set of names in LocalOperationNames $AI_G$, and must use a non-$\bot$ element from LocalOperationNames $AI_S$ as the interpretation for each operation in LocalOperation-
Names $A_{IG}$. More formally, for every such $A_{IS}$ and $A_{IG}$, $oim$ must satisfy two conditions:

\[
\text{LocalOperationNames } A_{IG} \subseteq \text{Dom } oim \quad (4.27)
\]
\[
\text{LocalOperationNames } A_{IS} \supseteq \text{Ran } oim \quad (4.28)
\]

Condition (4.27) requires that $oim$ provide an interpretation for every operation exported by $A_{IG}$. Condition (4.28) requires that each such interpretation be meaningful (i.e., nn-$\bot$).

2. For each operation $O \in \text{LocalOperationNames } A_{IG}$, let $O_S$ and $O_G$ be defined as follows:

\[
O_S = \text{OperationModel (} oim(O), A_{IS} \text{)} \quad (4.29)
\]
\[
O_G = \text{OperationModel (} O, A_{IG} \text{)} \quad (4.30)
\]

The lower level (more specific) operation model $O_S$ must be consistent with the behavior described by the higher level (more general) operation model $O_G$. Essentially, we want to ensure that the precondition in $O_S$ is the same or weaker as that of $O_G$, while the postcondition is the same or stronger. Unfortunately, these pre- and postconditions may be based on different type models for the parameters.

The type correspondences defined in $IM.TYmap$ for each type visible in $A_{IG}$ provide the basis for “interpreting” the lower level pre- and postconditions of $O_S$ to the higher level conceptual models so that we can properly compare the
behavior described by the two operation models. First, we require that both $O_S$ and $O_G$ have the same number of parameters in the same order, and that for each parameter type $T$ in $O_G.PP$, the corresponding parameter type in $O_S.PP$ is $IM.TYmap(T).N$.

Next, we can "translate" $O_S.DP$ and $O_S.EP$ by composing them with the type correspondence relations for each parameter type $T$ (i.e., $IM.TYmap(T).R$). Once this is done, we require that $O_G.DP$ imply the translated version of $O_G.DP$. Similarly, the translated version of $O_S.EP$ must imply $O_G.EP$.

### 4.10.4 Abstract Template Interpretation Mappings ($ATIM$)

An *Abstract Template Interpretation Mapping* is a name-to-value mapping from $N$ (abstract template names in $AIG$) to $N$ (abstract template names in $AIS$). To describe the conditions necessary for an abstract template interpretation mapping to be well-formed, we need to be able to talk about the identifiers of abstract template signatures defined in an abstract instance. For this purpose, we now define an overloaded operator AbstractTemplateNames.

The operator AbstractTemplateNames can be applied to abstract instances or abstract instance environments. For an abstract instance $AI$, AbstractTemplateNames $AI$ is defined as follows:

\[
\text{AbstractTemplateNames } AI = \text{Dom } AI.ATES \cup \text{AbstractTemplateNames } AI.CIE \cup \text{AbstractTemplateNames } AI.CTXT \quad (4.31)
\]
For an abstract instance environment $AIE$, the set of AbstractTemplateNames $AIE$ is defined as follows:

$$\text{AbstractTemplateNames } AIE = \bigcup_{AI \in \text{Dom } AIE} \text{AbstractTemplateNames } AI \quad (4.32)$$

As expected, LocalAbstractTemplateNames $AI$ is defined as AbstractTemplateNames $AI$—AbstractTemplateNames $AI.CTXT$. We can also give AbstractTemplateModel $(x, AI)$ the obvious analogous definition for $x \in \text{AbstractTemplateNames } AI$.

An Abstract Template Interpretation Mapping $atim$ is well-formed with respect to an abstract instance $AIS$, and abstract instance $AI_G$, and an interpretation mapping $IM$ if the following three conditions hold:

$$\forall x \in \text{LocalAbstractTemplateNames } AI_G : (\forall AT \in AT : (\text{AbstractTemplateModel } (x, AI_G).AIDP(AT) \text{ implies } \text{AbstractTemplateModel } (IM.ATmap(x), AIS).AIDP(AT)) \text{ and } (\text{AbstractTemplateModel } (IM.ATmap(x), AIS).AIEP(AT) \text{ implies } \text{AbstractTemplateModel } (x, AI_G).AIEP(AT))))$$

$$\forall x \in \text{LocalAbstractTemplateNames } AIS : \subseteq \text{Dom } atim \quad (4.33)$$

$$\forall x \in \text{LocalAbstractTemplateNames } AIS : \supseteq \text{Ran } atim \quad (4.34)$$

Condition (4.33) requires $atim$ provide an interpretation for every abstract template signature exported by $AI_G$. Condition (4.34) requires that each such interpretation be meaningful (i.e., $nn_{-\bot}$). Condition (4.35) requires that each such interpretation respect the pre- and postconditions of the abstract template signatures under consideration (i.e., the domain predicates of abstract template signatures in $AIS$ must
be the same or weaker than those in $AIG$, and the effect predicates in $AIS$ must be
the same or stronger).

### 4.10.5 Concrete Template Interpretation Mappings ($CTIM$)

A **Concrete Template Interpretation Mapping** is a name-to-value mapping from $N$
(concrete template names in $AIG$) to $N$ (concrete template names in $AIS$). To
describe the conditions necessary for a concrete template interpretation mapping to be
well-formed, we need to be able to talk about the identifiers of concrete template sig-
natures defined in an abstract instance. For this purpose, we now define an overloaded
operator $ConcreteTemplateNames$.

The operator $ConcreteTemplateNames$ can be applied to abstract instances or
abstract instance environments. For an abstract instance $AI$, the set of $Concrete-
TemplateNames AI$ is defined as follows:

$$ConcreteTemplateNames AI = \text{Dom } AI.CTES \cup \text{ConcreteTemplateNames } AI.CIE \cup \text{ConcreteTemplateNames } AI.CTXT \quad (4.36)$$

For an abstract instance environment $AIE$, $ConcreteTemplateNames AIE$ is defined
as follows:

$$ConcreteTemplateNames AIE = \bigcup_{AI \in \text{Dom } AIE} \text{ConcreteTemplateNames } AI \quad (4.37)$$

As expected, $LocalConcreteTemplateNames AI$ is defined as $ConcreteTemplateNames AI$–
$ConcreteTemplateNames AIE$. We can also give $ConcreteTemplateModel (x, AI)$
the obvious analogous definition for $x \in \text{ConcreteTemplateNames } AI$. 


A Concrete Template Interpretation Mapping \(ctim\) is well-formed with respect to an abstract instance \(AI_s\), and abstract instance \(AI_g\), and an interpretation mapping \(IM\) if the following three conditions hold:

LocalConcreteTemplateNames \(AI_g\) \(\subseteq\) Dom \(ctim\)  \hspace{1cm} (4.38)

LocalConcreteTemplateNames \(AI_s\) \(\supseteq\) Ran \(ctim\)  \hspace{1cm} (4.39)

\[ \forall x \in \text{LocalConcreteTemplateNames} \ AI_g : ( \forall CT \in CT : ( \text{ConcreteTemplateModel} (x, AI_g).CIDP(CT) \implies ( \text{ConcreteTemplateModel} (IM.CTmap(x), AI_s).CIDP(CT) \ \text{and} \ \text{ConcreteTemplateModel} (IM.CTmap(x), AI_s).CIEP(CT) \implies \text{ConcreteTemplateModel} (x, AI_g).CIEP(CT)))) \]  \hspace{1cm} (4.40)

Condition (4.38) requires that \(ctim\) provide an interpretation for every concrete template signature exported by \(AI_g\). Condition (4.39) requires that each such interpretation be meaningful (i.e., \text{nn-\bot}). Condition (4.40) requires that each such interpretation respect the pre- and postconditions of the concrete template signatures under consideration (i.e., the domain predicates of concrete template signatures in \(AI_s\) must be the same or weaker than those in \(AI_g\), and the effect predicates in \(AI_s\) must be the same or stronger).

4.11 Interpretation Mapping Templates (\(IMT\))

An Interpretation Mapping Template is a three-tuple \((IMTS, AIIMF, SF)\), where:

\(IMTS\) is an Interpretation Mapping Template Signature, defined below.
AIIMF is the "abstract instance-to-interpretation mapping function" \((AI \rightarrow IM)\) actually computed by instantiating the template.

SF is a "status function," an Assert Status-valued function over the input abstract instance \((AI \rightarrow AS)\) representing the assert status actually computed by instantiating the template.

An Interpretation Mapping Template \(imt\) is well-formed when the following two conditions are met:

1. For all input abstract instance values \(AI\) satisfying \(imt.IMTS.AIDP\), the effect predicate \(imt.IMTS.IMEP(AI, imt.AIIMF(AI))\) must be true. In other words, the actual abstract instance-to-interpretation mapping function computed by the template must be consistent with the postcondition described by \(imt.IMTS.IMEP\).

2. For all input abstract instance values \(AI\) satisfying \(imt.IMTS.AIDP\), \(imt.SF(AI)\) must not produce the value \(CF\).

An Intuitive Explanation of Interpretation Mapping Templates

An Interpretation Mapping Template is a generic or parameterized interpretation mapping, which should be thought of as a function from abstract instances to interpretation mappings. The characteristics of this function are captured in the three-tuple described above. Its structure exactly parallels that of a Concrete Template or an Abstract Template. Just like a concrete or abstract template, the nested structure of the single abstract instance parameter allows an interpretation mapping template
effectively to be parameterized by many instances. Because the requirements on
the single parameter are expressed through a predicate, that parameter may contain
variable numbers of nested instances, allowing variable-length generic parameter lists.

4.11.1 Interpretation Mapping Template Signatures (IMTS)

An Interpretation Mapping Template Signature is a two-tuple \(<AIDP, IMEP>\), where:

\(AIDP\) is an "abstract instance domain predicate," a boolean-valued function over
the input abstract instance \((\mathcal{A}I \rightarrow \mathcal{B})\) that represents the precondition for the
template.

\(IMEP\) is an "interpretation mapping effect predicate," a boolean-valued function over
the input abstract instance and the output interpretation mapping \((\mathcal{A}I \times \mathcal{I}M \rightarrow \mathcal{B})\) that represents the postcondition of instantiating the template.

All interpretation mapping template signatures are well-formed.

4.12 Basis Environments (BE)

A Basis Environment is a top-level run-time environment for a program. A Basis
Environment is a two-tuple \(<AS, E>\), where:

\(AS\) is an Assert Status, one of \(\{NL, VT, CF, \bot\}\).

\(E\) is an Environment (Section 4.5, p. 89).
An Intuitive Explanation of Basis Environments

A basis environment is simply a single concrete instance bundled together with an assert status (see Section 4.1.6). The concrete instance component of the basis environment is intended to model the traditional notion of a run-time program state—a name-to-value mapping that assigns appropriate values to all of the software objects in a program. Using a concrete instance for this purpose gives structure to the program state, allowing the value assignment the state represents to be compartmentalized along the subsystem boundaries within the program. Adding an assert status value supports formal verification in the “assertive programming” style. This combination is directly analogous to the program state model presented by Ernst et al. [10, p. 268] [12, pp. 2–3, 8–9], and is necessary for supporting modular verification of software [12, pp. 2–3].

4.13 Operations on Subsystems

All of the mathematical spaces in ACTI support the basic mathematical operations one would expect from their structure: projection and injection for cartesian products, application and composition for functions, and so on. Given these operations, however, there are several more sophisticated operations on the spaces which can help describe what happens to the current execution environment when subsystem-changing statements are executed (e.g., when instantiations occur, when inheritance is applied, or when subsystems are combined). For example, when instantiating a
component in RESOLVE the following tasks are all performed (not necessarily in this order):

- An abstract template (concept) is selected, say \( C \).

- Values for the RESOLVE **conceptual context** parameters from the current execution environment (e.g., a parameter value for the abstract template) are selected.

- A concrete template (realization), say \( R \), is selected that implements the chosen abstract template (i.e., there exists some interpretation mapping \( IM \) such that \( IM \models R \) is-interpretable-as \( C \)).

- Values for the RESOLVE **realization context** parameters from the current execution environment (e.g., a parameter value for the concrete template) are selected.

- All definitions provided by the resulting module instantiation are added to the execution environment.

The operations described in this section make it easier to talk about the primitives involved in higher-level operations like instantiation of a RESOLVE-style component.

### 4.13.1 Abstracting an Environment

Because *Concrete Instances* and *Abstract Instances* are so similarly structured, it is useful to define an operation that, given a concrete instance, will produce a corresponding abstract instance which contains "as much information as possible." Intu-
itively, one might think of this generated abstract instance as a "maximal specification" for the corresponding concrete instance, in that it contains exactly the same conceptual models, with all of the run-time value information removed.

This operation, called \texttt{abstract}, has the following signature:

\[
\text{abstract} : \mathcal{E} \rightarrow \mathcal{AI}
\]  

(4.41)

It works simply by eliminating some information from the environment as follows:

- Removing all value information from the \texttt{Variable Environment}, turning it into a \texttt{Variable Environment Signature}.

- Converting each \texttt{Operation Meaning} in the \texttt{Operation Environment} to an \texttt{Operation Model} by removing the procedure relation and status function components. This turns the \texttt{Operation Environment} into an \texttt{Operation Environment Signature}.

- Converting each \texttt{Concrete Instance} in the \texttt{Concrete Instance Environment} into an \texttt{Abstract Instance} by recursively applying the \texttt{abstract} operator.

- Converting each \texttt{Abstract Template} in the \texttt{Abstract Template Environment} to an \texttt{Abstract Template Signature} by removing its AIAIF and SF components.

- Converting each \texttt{Concrete Template} in the \texttt{Concrete Template Environment} to a \texttt{Concrete Template Signature} by removing its CICIF and SF components.

- Leaving the \texttt{CTXT, SAE, TE, EI, AIE, IME, and IMTE} components unchanged.
4.13.2 Reinterpreting an Environment

Given the abstract operator, it is now possible to talk about interpretation mappings that show how a particular interface (abstract instance) is a interpretation of an environment. For a given interpretation mapping $IM$, abstract instance $AI$, and environment $E$, if:

\[ IM \models \text{abstract}(E) \text{ is-interpretable-as } AI \]  \hspace{1cm} (4.42)

then we can say that the concrete instance $E$ can be interpreted as meeting the behavioral description in $AI$ under $IM$ (or, alternatively, that $E$ is an implementation of $AI$). Thus, the following is shorthand for the above expression, with application of the abstract operator omitted but understood:

\[ IM \models E \text{ is-interpretable-as } AI \]  \hspace{1cm} (4.43)

4.13.3 Cutting Down an Environment by an Interface

Given that we can talk about an interface generalizing an environment, say $IM \models E \text{ is-interpretable-as } AI$, we can then talk about a “cut” operator that can use such an interface along with the corresponding interpretation mapping to “reduce” the environment so that only what is “visible through the interface” remains. This is useful because $E$ may contain many objects that are not used in the interpretation. A cut operator allows one to formally model the effect of “encapsulating” the implementation $E$, so that only the features explicitly exported by the interface $AI$ are visible.
The cut operator has the following signature:

$$\downarrow: \mathcal{E} \times \mathcal{IM} \times \mathcal{AI} \rightarrow \mathcal{E}$$ \hfill (4.44)

Intuitively, this operator simply removes all information from the input environment that is not mapped through the interface via $\mathcal{IM}$. For a given $\mathcal{E}$, $\mathcal{IM}$, and $\mathcal{AI}$, applying this operator produces a new environment $\mathcal{E}_2 = (\mathcal{E} \downarrow_{\mathcal{IM}} \mathcal{AI})$. Essentially, this new environment is $\mathcal{AI}$, extended with information about the variable values, procedure relations, status functions, and concrete instances from $\mathcal{E}$, as viewed through the module and type correspondences defined in $\mathcal{IM}$. As a result, $\text{abstract}(\mathcal{E}_2) \approx \mathcal{AI}$.

The only place where $\text{abstract} \ E_2$ and $\mathcal{AI}$ differ is in their $\text{CTX}T$ components.

Since $\mathcal{E}$ may have more external dependencies in its context than $\mathcal{AI}$ does, these extra dependencies must be preserved in order for the implementation of $\mathcal{E}$ to remain valid. As a result, $\mathcal{E}_2.\text{CTX}T$ is defined as:

$$\mathcal{E}_2.\text{CTX}T = \begin{cases} 
\text{CTX}T &= \mathcal{E}.\text{CTX}T.\text{CTX}T \\
\text{SAE} &= \mathcal{E}.\text{CTX}T.\text{SAE} \\
\text{TE} &= \mathcal{E}.\text{CTX}T.\text{TE} \\
\text{VES} &= \mathcal{E}.\text{CTX}T.\text{VES} \\
\text{OES} &= \mathcal{E}.\text{CTX}T.\text{OES} \\
\text{EI} &= \mathcal{E}.\text{CTX}T.\text{EI} \\
\text{AIE} &= \mathcal{E}.\text{CTX}T.\text{AIE} \\
\text{CIE} &= \mathcal{E}.\text{CTX}T.\text{CIE} + \\
\text{IME} &= \mathcal{E}.\text{CTX}T.\text{IME} \\
\text{ATES} &= \mathcal{E}.\text{CTX}T.\text{ATES} \\
\text{CTES} &= \mathcal{E}.\text{CTX}T.\text{CTES} \\
\text{IMTE} &= \mathcal{E}.\text{CTX}T.\text{IMTE} \\
\end{cases}$$ \hfill (4.45)

\[\text{unused} \mapsto \langle (\text{AI.CTX}T, \bot, \bot, \bot, \bot, \text{true}, \downarrow, \bot, \bot, \bot, \bot, \bot), \text{IM.CTX}T\text{map} \rangle\]
Here, *unused* represents some name from \( \mathcal{N} \) that is not in \( \text{Dom } E.CTXT.CIE \). This modification creates an empty subsystem with the same context as \( AI \), and then embeds (see Section 4.13.6 below) it within the context of \( E \), and uses the result as the context for \( E_2 \). This ensures that all external dependencies from \( E \) are preserved, while their interpretation under \( IM \) as the more abstract context of \( AI \) is also recorded.

### 4.13.4 Derivation of Instances by Difference

Given two abstract instances \( AI_1 \) and \( AI_2 \), it is natural to begin thinking about module composition operators. For example, one would expect:

\[
AI_1 + AI_2 = AI_3
\]  

(4.46)

to produce a third abstract instance which defines the "sum" of the types, operations, sub-instances, etc., of \( AI_1 \) and \( AI_2 \) (assuming that the sets of names defined in \( AI_1 \) and \( AI_2 \) are disjoint). In fact, this is a basic generalization of the "modifies" operator for name-to-value mappings introduced in Section 4.1.2. An abstract (or concrete) instance contains only three components that are not name-to-value mappings: \( CTXT \), \( SAE \), and \( EI \). Specification adornment environments contain only name-to-value mappings and an invariant \( SAEI \). Thus, we can recursively define the + operator on specification adornment environments as:

\[
SAE_1 + SAE_2 = \begin{cases} 
SATY &= SAE_1.SATY + SAE_2.SATY \\
SAVE &= SAE_1.SAVE + SAE_2.SAVE \\
SAEI &= SAE_1.SAEI \land SAE_2.SAEI \\
SAIE &= SAE_1.SAIE + SAE_2.SAIE \\
SATE &= SAE_1.SATE + SAE_2.SATE 
\end{cases}
\]  

(4.47)
From this, we can now define the + operator on abstract instances:

\[
AI_1 + AI_2 = \begin{cases}
    CTXT &= AI_1.CTXT + AI_2.CTXT \\
    SAE &= AI_1.SAE + AI_2.SAE \\
    TE &= AI_1.TE + AI_2.TE \\
    VES &= AI_1.VES + AI_2.VES \\
    OES &= AI_1.OES + AI_2.OES \\
    EI &= AI_1.EI \land AI_2.EI \\
    AIE &= AI_1.AIE + AI_2.AIE \\
    CIE &= AI_1.CIE + AI_2.CIE \\
    IME &= AI_1.IME + AI_2.IME \\
    ATES &= AI_1.ATES + AI_2.ATES \\
    CTES &= AI_1.CTES + AI_2.CTES \\
    IMTE &= AI_1.IMTE + AI_2.IMTE
\end{cases}
\] (4.48)

Modification of concrete instances is defined similarly.

Given this basic modification operation, all of Goguen’s module expression calculus [21, pp. 194–198] can be easily adapted to abstract and concrete instances. It can even be applied to templates and interpretation mappings. As a result, Goguen’s entire “horizontal composition” strategy, which amounts to inheritance used for programming by difference—purely as a definitional mechanism without any effect on the typing system—can be easily interpreted in the framework of the ACTI model.

4.13.5 Composing Interpretation Mappings

Given two interpretation mappings, \( IM_1 \) and \( IM_2 \), and an abstract instance \( AI \), it is possible to define the composite mapping:

\[
IM_1 \circ AI \circ IM_2 = IM_3
\] (4.49)

This is a natural result of the fact that interpretation mappings form a preorder over abstract instances.
4.13.6 Binding Context and Composing Instances

Because every abstract and concrete instance in ACTI has an explicit context interface (recorded in its \textit{CTXT} component), the question arises of how such an interface is tied to specific external definitions in the subsystem's surrounding environment. As indicated in Section 4.5.5 (pages 101-104), "context-bound" abstract and concrete instances describe subsystems with context explicitly tied to their surrounding environment. To "fix" the context of a subsystem \( S \), one must literally embed \( S \) into

---

\textbf{Figure 28: A Subsystem} \( S \) \textit{and the Outer Environment That Will Provide Its Context}

---
some larger subsystem that represents the environment in which $S$ lives, say $OUTER$. This outer environment may be either an abstract or a concrete instance.

Before binding such a subsystem's context, one must have a subsystem $S$ in isolation, i.e., with free context, as well as some outer environment $OUTER$ that will provide the surrounding medium providing actual objects matching the context requirements of $S.CTXT$. This situation is depicted in Figure 28.

The only missing component is an explanation of how $OUTER$ actually fulfills the context dependencies of $S$. All that is required is an interpretation mapping $IM$
such that $IM \models OUTER$ is-interprettable-as $S.CTXT$. Thus, $IM$ formally defines how $OUTER$ (or abstract($OUTER$) if $OUTER$ is concrete) can be interpreted as the context of $S$. This is shown in Figure 29.

Given that $S$, $OUTER$, and $IM$ are all well-formed, and $IM.DP(OUTER, S.CTXT)$, the precondition for $IM$, is true, all that is left is to embed the pair $\langle S, IM \rangle$ in the appropriate (abstract or concrete) instance environment in $OUTER$ with a new name, say $S_{name}$. Suppose both $S$ and $OUTER$ are abstract instances. Embedding $\langle S, IM \rangle$ in the concrete instance environment of $OUTER$ is accomplished by modifying $OUTER.CIE$ to be:

$$OUTER.CIE + \{S_{name} \mapsto \langle S, IM \rangle\}$$ (4.50)

### 4.13.7 Instantiating a Template

Intuitively, instantiating a template is just like applying a function. A template’s parameter is a single subsystem, and applying the template to that parameter produces another subsystem. Templates are only meaningfully applied to subsystems that meet their preconditions—i.e., satisfy their domain predicates.

We can use normal functional notation to represent the application of a template. Suppose $AT$ is an abstract template, and $S$ is an abstract instance such that $AT.AIDP(S)$ is true. Then $AT.AIAIF(S)$, or simply $AT(S)$, is also an abstract template, which may later be embedded in another subsystem given an appropriate interpretation mapping to bind its context.
4.14 Connection with Traditional Denotational Semantics

Most conventional work on denotational semantics for particular programming languages has focused on the meaning of lower-level programming constructs: types, variables, control flow constructs, procedures, and so on. Work has also been done on intra-module assertions like correspondence relations and conventions [33]. From this perspective, a view of a program state as a simple mapping from variable names to variable values has been adequate. As suggested by Ernst et al., however, the notion of program state must expand to include information about procedure parameter profiles and behavioral specifications (i.e., Operation Models) and even information about module specifications, if modular verification is to be possible [12, p. 2]. This is exactly what has been done in ACTI.

Fortunately, adding this additional information and a corresponding structure to the notion of a program state does not invalidate previous work on the meaning of lower-level programming constructs. ACTI is geared toward integration with a procedure-level denotational semantic model similar to that described by Ernst et al. [12, 10, 13, 11]. In fact, if operation meanings included a procedure function (instead of a procedure relation), no changes would be necessary to this prior work in order for it to be used in combination with ACTI.

However, because of the relational nature of Operation Models, and the relational nature of type correspondence relations and subsystem state correspondence relations in interpretation mappings, A strictly functional notion of procedural semantics is not viable. This is due to the fact that in ACTI, a procedure's meaning can only be defined
in terms of the context of the subsystem it is contained in. This context contains only operation models, not operation meanings, so that the meaning of the procedure in question can be defined independently of any other external operations upon which it relies. Since the behavioral description in an operation model can be relational, regardless of the actual computation carried out by the corresponding code, in general a procedure depending on such context cannot be given a strictly functional meaning.

Interestingly, John Sanderson points to a solution to this problem [56]. He describes a relational calculus of computation suitable for modeling the denotational semantics of lower-level program constructs. His calculus has the compositional and least fixed point properties necessary for use in modeling the semantics of conventional languages. This calculus could be adopted for integration with ACTI to provide a complete denotational semantic model for a programming language, such as RESOLVE.

4.15 Strengths and Weaknesses

In addition to the contributions listed in Section 1.3, the ACTI model as defined here has several important strengths:

- It provides detailed support for modeling the behavior of all software objects. Models of software objects, if written down formally or informally are, critical for supporting the formation of effective mental models. As a result, they are critical for supporting understandability.
• It provides strong support for separation of specification (or abstract modeling) from implementation in all of the details of programming. The prevalence of spaces of “signatures” in the ACTI definition shows that this separation is reflected in many more ways than simply the separation of module specifications from module implementations.

• It meets all criteria for a “comprehensive” model of software structure and meaning presented in Appendix B.

• It combines the advantages of the previous models of software described in Appendix A, while avoiding their disadvantages.

Notably, however, the generality achieved in this model brings with it some traits that might be considered weaknesses from the perspective of more traditional work in modeling software meaning:

• The ACTI model is not “fully abstract” in the sense described by Allen Stoughton [60]. Full abstractness is based on the idea of behavioral equivalence. Two programs are behaviorally equivalent if they compute the same input-output function (or relation). They are indistinguishable in terms of their externally visible behavior. For a model of software to be fully abstract, it must model behaviorally equivalent software constructs using the same denotation.

  In ACTI two subsystems that are behaviorally equivalent need not be identical. Even though their behavior is the same, the models used to describe their behavior, their specification adornment definitions, and so on, may all be different.
This can pose problems for formal reasoning about the equivalence of software parts.

It may be possible to address this limitation by formalizing the notion of a "reversible" or "two-way" interpretation mapping. Effectively, such a mapping would capture the idea of a behavioral isomorphism between the two subsystems.

- Similarly, the fact that two software objects are identical in ACTI does not mean that are always legitimately interchangeable. For example, the Finalize operation for Partial.Map introduced in Section 3.4.2 has a behavioral specification that says it has no externally visible effect. This does not mean, however, that its execution can be eliminated by an optimizer—substituted by a null operation that has the same external behavior. This is typically only a problem with infrastructure-related functions, like Initialize and Finalize, that are built-in to a specific language or execution model.

- Finally, interpretation mappings do not define a partial ordering over abstract instances; they define a preordering. That is, for two abstract instances $A$ and $B$, $A$ is-interpretable-as $B$ and $B$ is-interpretable-as $A$ does not necessarily imply $A = B$. Much of the past work on relationships between specifications (subtyping, refinement, and so on) relies on a partial ordering of specifications. This problem is partly due to the fact that interpretation mappings are more general than other proposed specification relations, and also partly due to the fact that ACTI is not a fully abstract model of software.
4.16 Chapter Summary

This chapter presents the formal definition of the ACTI software model. This model is centered around the mathematical spaces modeling the four types of subsystems: Abstract Instances, Concrete Instances, Abstract Templates, and Concrete Templates. The model also includes Interpretation Mappings and Interpretation Mapping Templates as central ideas. All of the spaces in ACTI are complete partial orders (CPOs), so that the model can be used as the starting point for the denotational semantics of a programming language.

Many of the mathematical spaces in ACTI contain name-to-value mappings. These mappings associate identifiers from the space of Names with objects from other ACTI spaces. This is analogous to the traditional idea of a program state. In ACTI, however, a program's state can be modeled with a Basis Environment, which is a highly structured collection of name-to-value mappings, rather than a single flat state function.

In addition to the notion of name-to-value mappings, the ACTI definition also uses the concept of signatures. Many of the mathematical spaces in the ACTI model are related in pairs—one space to represent actual software objects, and the other to represent specifications, or conceptual models, of those objects. Such a space of specification-only values is called a space of signatures.

The ACTI definition presented in this chapter provides the foundation necessary to understand the intuitive concepts introduced in Chapter III at a formal level.
Chapter V continues the example introduced in Chapter III in the context of this formal definition.
CHAPTER V

An Extended ACTI Example

This chapter provides an extension of the running Partial_Map example introduced in Chapter III, to illustrate some of the more advanced aspects of ACTI. It demonstrates the use of interpretation mappings between abstract and concrete instances, and explains the semantics of subsystem-level program statements such as generic instantiations in terms of the ACTI model. This is done by examining the effect that the execution of a single RESOLVE facility instantiation statement has on a basis environment modeling the state of a program. After describing the instantiation example and the initial basis environment, this chapter shows how to instantiate an abstract template, instantiate a concrete template, instantiate an interpretation mapping template, combine the three instances, and embed the result back in the original basis environment. A good understanding of RESOLVE will be helpful in fully appreciating the example presented in chapter.

5.1 Facility Instantiation in RESOLVE

Chapter III introduced a Partial_Map ADT, which is modeled as a set of ordered pairs of D_Items and R_Items. Each ordered pair in a Partial_Map object represents
one key-to-value association stored in that map, and the map as a whole is simply a collection of such associations for some group of unique keys \( \text{D.Items} \). This abstract notion of a set of key-value associations can be used for many purposes where database-like behavior is required, such as in symbol tables, dictionaries, or associative memories.

Chapter III also introduced the idea of a "communally bounded" version of an ADT. The \texttt{Communal_PMap.Template} concept presented in Figures 22 and 23 describes a type \texttt{Partial.Map} that is communally bounded—it has an upper limit on the aggregate number of associations stored at any one time in all \texttt{Partial.Map} objects. Here, we further examine the \texttt{Communal_PMap.Template} concept, a RESOLVE realization for this concept, and how the two are instantiated to form a facility, or instance module, that can then be used in a program.

Figure 30 shows the RESOLVE syntax for instantiating \texttt{Communal_PMap.Template} to create a facility \texttt{Int.Multiset}. Suppose this facility is being created by a programmer implementing another abstraction, say a multiset of integers. Such a multiset could be conveniently represented by just storing the distinct elements of the multiset, and then storing an occurrence count for each of those distinct elements. A partial map could be used for this purpose.

Box \( (a) \) of Figure 30 shows \texttt{Communal_PMap.Template} being instantiated with the program type \texttt{Integer} as its \texttt{D.Item} (the kind of elements in the multiset), \texttt{Integer} as its \texttt{R.Item} (the occurrence count for the corresponding element), and 1024 as the \texttt{max.total.size} limit. Further, box \( (a) \) of Figure 30 shows that the \texttt{Unordered.Stack}
facility Int.Int.Pair.Facility is
  Record2.Template (Integer, Integer)
  realized by
  Standard

facility Int.Int.Stack.Facility is
  Stack.Template (Int.Int.Pair.Facility.Record2)
  realized by
  List

facility Int.Multiset is
  Communal_PMap.Template (Integer, Integer, 1024)
  realized by

Figure 30: Instantiating Communal_PMap.Template in RESOLVE

realization of Communal_PMap.Template is being selected to provide the implementation for Int.Multiset facility. This realization, adapted directly from an unbounded implementation presented by Bucci et al. [2, pp. 42-44], was chosen for simplicity and understandability. It represents each Partial.Map object as a stack of ordered pairs, each containing one D.Item and one R.Item. The stack is not maintained in any particular order, hence the name Unordered.Stack.

There are two additional instantiation statements present in Figure 30, one for Int.Int.Pair.Facility and one for Int.Int.Stack.Facility. These statements are there to declare instances of other components that will be used by the implementation
of Int_Multiset, and they are thus provided as parameters to Unordered_Stack as part of the instantiation in box (a). Int_Int_Pair_Facility defines a type Record2 that is an ordered pair of two integers. Int_Int_Stack_Facility defines a type Stack that is a stack of Record2 ordered pairs. It is clear how these two pieces can be used to construct the Unordered_Stack realization of the Communal_PMap_Template concept, and the source code for this implementation will be presented in Section 5.5.

In ACTI terms, Communal_PMap_Template is an abstract template, Unordered_Stack is a concrete template, and the three facilities Int_Int_Pair_Facility, Int_Int_Stack_Facility, and Int_Multiset, are all concrete instances. Further, implicitly there is an interpretation mapping that describes why the behavior of Unordered_Stack, written in terms of Record2 and Stack objects, conforms with the more abstract behavior specified in Communal_PMap_Template, written in terms of sets of ordered pairs. In this chapter, we look at how the single RESOLVE statement in box (a) of Figure 30 can be interpreted in ACTI terms.

To set the stage for this, it is important to note that in RESOLVE, all of the parametric context to Communal_PMap_Template is available inside Unordered_Stack. Figure 31 makes explicit the fact that the realization is parameterized by the concept's parameters. This is done by repeating the concept's parameters in the arguments to the realization. Figure 31 also makes explicit the interpretation mapping that describes the relationship between the results of the abstract template and the concrete template. Here, this interpretation mapping is called UStack_To_CPMap, and it will be discussed in detail in Section 5.6.
facility Int_Multiset is
  Communal_PMap_Template (
    Standard_Integer_Facility.Integer,
    Standard_Integer_Facility.Integer,
    1024)
realized by
  Unordered.Stack (
    Standard_Integer_Facility.Integer,
    Standard_Integer_Facility.Integer,
    1024,
    Int_Int_Pair_Facility,
    Int_Int_Stack_Facility,
    Standard_Integer_Facility.Are.Equal)
interpreted by
  UStack_To_CPMap (
    Standard_Integer_Facility.Integer,
    Standard_Integer_Facility.Integer)

Figure 31: A More Detailed View of the Instantiation

Making these details visible (where they are implicit in RESOLVE as currently defined) allows us to view the semantics of the single RESOLVE-like statement in Figure 31 in ACTI terms. The instantiation involves five distinct steps:

1. Instantiating Communal_PMap_Template (an abstract template) to produce an abstract instance.

2. Instantiating Unordered_STACK (a concrete template) to produce a concrete instance.

3. Instantiating UStack_To_CPMap (an interpretation mapping template) to produce an interpretation mapping.
4. Combining the results of steps (1), (2), and (3) to form a new facility, which is a concrete instance.

5. Associating this newly created concrete instance with the name \texttt{Int\_Multiset} in our model of the program's state.

After laying the groundwork for discussing this example in ACTI terms, we explore each of these steps in greater detail.

5.2 Notation

Throughout Chapter III an informal graphical representation of subsystems was used, as typified by Figure 26 (p. 71). Given the formal definition of the ACTI mathematical spaces in Chapter IV, it is now possible to give such figures a more rigorous meaning. Figure 32 summarizes the formal definition of an ACTI abstract instance.

The outermost box in Figure 32 represents an entire abstract instance, and all other module-like structures—abstract instances, concrete instances, and specification adornment environments—within that abstract instance are also depicted using boxes. The outermost abstract instance is actually a 12-tuple; the individual components of the tuple are presented vertically and identified by their names, as defined in Section 4.6. All other tuple objects will be represented similarly. The name-to-value mappings within the abstract instance in Figure 32 are represented using the notation introduced in Section 4.1.2, laid out vertically. Interpretation mappings are shown simply as upward-pointing double-arrows, and will be fully described in separate
<table>
<thead>
<tr>
<th>$CTXT$ =</th>
<th>![Blank Box]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SATY$ =</td>
<td>{...}</td>
</tr>
<tr>
<td>$SAVE$ =</td>
<td>{...}</td>
</tr>
<tr>
<td>$SAEI$ =</td>
<td>{...}</td>
</tr>
<tr>
<td>$SAIE$ =</td>
<td>{ Names $\mapsto$ ![Blank Box] }</td>
</tr>
<tr>
<td>$SATE$ =</td>
<td>{ Names $\mapsto$ ( ![Blank Box] $\rightarrow$ ![Blank Box] ) }</td>
</tr>
<tr>
<td>$TE$ =</td>
<td>{...}</td>
</tr>
<tr>
<td>$VES$ =</td>
<td>{...}</td>
</tr>
<tr>
<td>$OES$ =</td>
<td>{...}</td>
</tr>
<tr>
<td>$EI$ =</td>
<td>{...}</td>
</tr>
<tr>
<td>$AIE$ =</td>
<td>{ Names $\mapsto$ ![Blank Box] }</td>
</tr>
<tr>
<td>$CIE$ =</td>
<td>{ Names $\mapsto$ ![Blank Box] }</td>
</tr>
<tr>
<td>$IME$ =</td>
<td>{ Names $\mapsto$ $\uparrow$ }</td>
</tr>
<tr>
<td>$ATE$ =</td>
<td>{ Names $\mapsto$ ( ![Blank Box] $\rightarrow$ ![Blank Box] ) }</td>
</tr>
<tr>
<td>$CTE$ =</td>
<td>{ Names $\mapsto$ ( ![Blank Box] $\rightarrow$ ![Blank Box] ) }</td>
</tr>
<tr>
<td>$IMTE$ =</td>
<td>{ Names $\mapsto$ ( ![Blank Box] $\rightarrow$ $\uparrow$ ) }</td>
</tr>
</tbody>
</table>

Figure 32: Formal Notation for Subsystems

figures when required. Similarly, templates are graphically represented as functions (arrows) from a parameter to a result.

To simplify subsystem diagrams of this form, we will adopt the convention that only non-$\bot$ components of tuple objects need be shown. Thus, any tuple fields that are omitted are understood to have the $\bot$ value from their corresponding domain.
The only exceptions to this rule are invariants, boolean predicates that when omitted are understood to be universally true.

5.3 A Complete Computational Environment

Intuitively, the instantiation of_Communal_PMap_Template presented in Figure 30 declares a new program unit, the facility Int_Multiset, that can be used in later program statements. In order to meaningfully discuss the formalization of this intuition, one needs a formal analog of the "state" of a program. This sets the groundwork for presenting the effect of the Int_Multiset instantiation statement, in terms of how that statement alters the state that the program was in before the instantiation was executed.

In ACTI, the natural denotation for a program state is a Basis Environment. Figure 33 shows a basis environment \( BE \) that represents the state of a program before the execution of the Int_Multiset instantiation. Figure 33 shows all of the components of \( BE \), but using the conventions presented in Section 5.2, we can abbreviate \( BE \) by eliding \( \bot \)-valued components. Figure 34 shows \( BE \), omitting all of the components with "uninteresting" values.

The basis environment \( BE \) records the assert status (section 4.1.6) of the program state, and an environment (concrete instance) \( E \) that associates identifiers with values. \( BE \) includes four nested concrete instances, each representing a particular RESOLVE facility. Two of the four are standard facilities defining the programming types Boolean and Integer. The remaining two are the result of the first two instan-
\[
A S = NL
\]

\[
CTXT = \bot
\]

\[
SAE = \\
\]

\[
SAVE = \{size\_limit \mapsto [V SIG = \text{math-integer}]
\]

\[
TE = \bot
\]
\[
VE = \bot
\]
\[
OE = \bot
\]
\[
EI = \text{true}
\]
\[
AIE = \bot
\]

\[
BE = E = \\
\]

\[
CIE = \\
\]

\[
IME = \bot
\]

\[
ATE = \{ \text{Communal.PMap.Template} \mapsto ([\square] \rightarrow [\square])
\]

\[
CTE = \{ \text{Unordered.Stack} \mapsto ([\square] \rightarrow [\square])
\]

\[
IMTE = \{ \text{UStack.To.CPMap} \mapsto ([\square] \rightarrow \uparrow)
\]

Figure 33: An Initial Basis Environment \(BE\)
Figure 34: BE: A Simplified View

tiations shown in Figure 30. BE also associates values with Communal_PMap_Template, Unordered_Stack, and UStack_To_CMap, each of which will be discussed in a separate section. Finally, BE has a size.limit specification adornment variable with the integer value 1024.
To fully understand the signature (i.e., type) of \texttt{size.limit}, a brief explanation of the mathematical models for the traditional scalar types in RESOLVE is useful$^1$. In RESOLVE, the reserved words \texttt{boolean} and \texttt{integer} are used to refer to the usual \textit{mathematical} (not programming) notions, which are built-in to the RESOLVE specification notation for convenience. The corresponding programming types are not built-in to the language, but are instead defined in facilities, just like user-defined programming types. As a result, the program identifiers "boolean" and "integer" are overloaded, each representing both a mathematical type used for model-based specifications, and a programming type used to construct implementations. Because of the ubiquitous nature of these scalar programming types, a RESOLVE programming environment will contain a "standard" implementation for each of them.

By convention, the RESOLVE literature uses \texttt{boolean} or \texttt{integer} (in bold face) as a reserved word to refer to the corresponding mathematical type, and \texttt{Boolean} or \texttt{Integer} (with initial capitalization) to refer to the programming type. Because all ACTI identifiers are assumed to be unique, throughout this example "math-\texttt{boolean}" (or "math-\texttt{integer}") will be used to refer to the mathematical type denoted by the corresponding RESOLVE reserved word, and "prog-Boolean" (or "prog-Integer") will be used to refer to the programming type. Further, for the purposes of the figures in this chapter, we will present both the mathematical and programming types as being defined within the corresponding standard facility. Of course, defining the denotational semantics of RESOLVE within the ACTI model would naturally involve

$^1$ See [66] for a more detailed description.
Figure 35: \( BE \) After Instantiating Communal.PMap.Template

completely defining an initial basis environment capturing all of the types, operations, facilities, and templates that are built-in to the language or provided as part of the
standard programming environment. That is beyond the scope of this dissertation, and not essential to understanding the example.

The Int_Multiset instantiation statement presented in Figure 30 has the simple effect of adding one more name-to-value association to the $BE.E.CIE$—one that associates Int_Multiset with a context-bound concrete instance representing the facility produced by the instantiation statement. After executing the Int_Multiset instantiation, the resulting program state $BE'$ is the one depicted in Figure 35. This is the high-level view of the semantics of this single program statement. We will now proceed to describe the five smaller steps that are combined to achieve this effect.

**5.4 Instantiating Communal_PMap_Template**

As described in Section 5.1, the first step in the Int_Multiset instantiation is applying the abstract template Communal_PMap_Template. Communal_PMap_Template was first introduced in Chapter III, Figures 22 and 23.

As indicated in Chapter III, it is possible to generalize the PARTIAL_FUNCTION adornment definitions and move them into a separate program unit so that they can be reused in many specifications. Figures 36 through 38 present a slight variation of Communal_PMap_Template where this has been done. Here, the specification adornment definition of the type PARTIAL_FUNCTION, as well as several operations on partial functions, have been moved into a parameterized RESOLVE mathematics module [28]. This mathematics module is called PARTIAL_FUNCTION_THEORY_TEMPLATE, and is presented in Section 5.4.1. As shown in Figure 36, this mathematics module is
concept Communal_PMap_Template

context
  global context
    facility Standard_Boolean_Facility
    facility Standard_Integer_Facility

  mathematics PARTIAL_FUNCTION THEORY TEMPLATE

parametric context
  type D.Item
  type R.Item

  constant max_total_size : integer
    restriction
      max_total_size > 0

local context

  math facility PARTIAL_FUNCTION THEORY is
    PARTIAL_FUNCTION THEORY TEMPLATE (
      math[D.Item], math[R.Item])

  state variables
    total_size : integer
    constraint
      0 ≤ total_size ≤ max_total_size
    initialization
      ensures total_size = 0

interface

  type Partial_Map is modeled by PARTIAL_FUNCTION
  exemplar m
  initialization
    ensures m = empty_set

Figure 36: The Communal_PMap_Template Concept
operation Define (  
    alters m : Partial.Map  
    consumes d : D.Item  
    consumes r : R.Item  
  )  
  referenced state variables  
  alters total.size  
  requires not DEFINED_IN (m, d) and  
  total.size < max.total.size  
  ensures m = #m union {(#d, #r)} and  
  total.size = #total.size + 1

operation Undefine (  
    alters m : Partial.Map  
    preserves d : D.Item  
    produces d.copy : D.Item  
    produces r : R.Item  
  )  
  referenced state variables  
  alters total.size  
  requires DEFINED_IN (m, d)  
  ensures (d, r) is in #m and  
  m = #m - {(d, r)} and  
  d.copy = d and  
  total.size = #total.size - 1

operation Undefine.Any.One (  
    alters m : Partial.Map  
    produces d : D.Item  
    produces r : R.Item  
  )  
  referenced state variables  
  alters total.size  
  requires m ^ empty.set  
  ensures (d, r) is in #m and  
  m = #m - {(d, r)} and  
  total.size = #total.size - 1

Figure 37: The Communal_PMap_Template Concept (continued)
operation Is_Defined (  
    preserves m : Partial_Map  
    preserves d : D_Item  
  ) : Boolean  
  ensures Is_Defined iff DEFINED_IN (m, d)

operation Size (  
    preserves m : Partial_Map  
  ) : Integer  
  ensures Size = |m|

end Communal.PMap.Template

Figure 38: The Communal.PMap.Template Concept (continued)

imported and instantiated by Communal.PMap.Template. Examining the meaning of  
PARTIAL_FUNCTION_THEORY_TEMPLATE, and how it might be instantiated, will help in  
the understanding of Communal.PMap.Template.

5.4.1 Instantiating PARTIAL_FUNCTION_THEORY_TEMPLATE

PARTIAL_FUNCTION_THEORY_TEMPLATE is a RESOLVE parameterized mathematics mod­
ule. Mathematics modules only contain mathematical definitions used in the spec­
ification of program types and operations. Because of this, it is natural to use a  
Specification Adornment Environment as the denotation for a mathematics module.  
Similarly, a Specification Adornment Template is the natural denotation for a param­
eterized mathematics module. Figures 39 and 40 present the RESOLVE definition of  
PARTIAL_FUNCTION_THEORY_TEMPLATE.
mathematics PARTIAL_FUNCTION_THEORY_TEMPLATE

context
  parametric context

  math type DOMAIN_ITEM

  math type RANGE_ITEM

interface

  math subtype PARTIAL_FUNCTION is set of ( 
    d : DOMAIN_ITEM
    r : RANGE_ITEM
  )

exemplar  m

constraint
  for all d : DOMAIN_ITEM, r1, r2 : RANGE_ITEM
  where ((d, r1) is in m and (d, r2) is in m) \((a)\)
  \((r_1 = r_2)\)

math operation EMPTY_PARTIAL_FUNCTION : PARTIAL_FUNCTION

  explicit definition
  empty_set

math operation DEFINED_IN ( 
  m : PARTIAL_FUNCTION
  d : DOMAIN_ITEM
) : boolean

  explicit definition
  \[\text{there exists } r : \text{RANGE_ITEM } ((d, r) \text{ is in } m)\] \((b)\)

Figure 39: The PARTIAL_FUNCTION_THEORY_TEMPLATE Mathematics
math operation DIFFERONLYAT (m1 : PARTIAL_FUNCTION, m2 : PARTIAL_FUNCTION, d_items : set of DOMAIN_ITEM) : boolean

explicit definition

\[
\text{for all } \text{dr_pair} : (\begin{array}{ll}
    d & : \text{DOMAIN_ITEM} \\
    r & : \text{RANGE_ITEM}
\end{array} \) \text{ where (dr_pair.d is not in d_items)}
\]
\[
    (\text{dr_pair is in m1 iff dr_pair is in m2})
\]

end PARTIAL_FUNCTION_THEORY_TEMPLATE

Figure 40: The PARTIAL_FUNCTION_THEORY_TEMPLATE Mathematics (continued)

A specification adornment template is a function from specification adornment environments to specification adornment environments, as shown in Figure 41. Thus, PARTIAL_FUNCTION_THEORY_TEMPLATE takes a single specification adornment environment as its parameter, and produces a single specification adornment environment as its result. As indicated by the parametric context in Figure 39, PARTIAL_FUNCTION_THEORY_TEMPLATE expects two types (or, in ACTI terms, a specification adornment environment providing two types) as its parameter, and there are no restrictions on what abstract models can be used for these types. In ACTI terms, we can say that the specification adornment template denoted by PARTIAL_FUNCTION_THEORY_TEMPLATE will accept as its parameter any specification adornment environment matching the one in Figure 42. Effectively, the domain predicate (e.g., SATS.SAIDP) for this
Figure 41: **PARTIAL_FUNCTION_THEORY_TEMPLATE** is a Specification Adornment Template.

This template is true for any specification adornment environment matching Figure 42, and false for others.

Similarly, we can characterize both the effect predicate (e.g., **SAIS**.SAIEP) and the specification adornment instance function (**SAISAIF**) associated with the specification adornment template **PARTIAL_FUNCTION_THEORY_TEMPLATE** using another figure to show the structure of the result produced by the template. Figure 43 shows the specification adornment instance that results from applying this template to a parameter matching Figure 42. This result depends on the specific *Type Model* values associated with the identifiers **DOMAIN.ITEM** and **RANGE.ITEM** in the parameter. If the parameter maps **DOMAIN.ITEM** to the pair \(\langle MD_1, CNSTR_1 \rangle\), then in Figure 43 we use **DI-Model** to refer to the space \(CNSTR_1(MD_1)\). **RI-Model** is defined similarly.

\[
SATY = \begin{cases} 
\text{DOMAIN.ITEM} & \mapsto <\text{Any model, say DI-Model}> \\
\text{RANGE.ITEM} & \mapsto <\text{Any model, say RI-Model}>
\end{cases}
\]

Figure 42: **PARTIAL_FUNCTION_THEORY_TEMPLATE**'s Parameter
\[
\begin{align*}
SATY &= \{ \\
\text{PARTIAL\_FUNCTION} &\mapsto \begin{cases} \\
MD &= \mathcal{P}(\text{DI-Model} \times \text{RI-Model}) \\
CNSTR &= \text{<Constraint defined in Fig. 39.(a)>} \\
\end{cases} \\
\text{Anonymous1} &\mapsto \begin{cases} \\
MD &= \text{PARTIAL\_FUNCTION} \times \text{DI-Model} \to B \\
CNSTR &= \text{true} \\
\end{cases} \\
\text{Anonymous2} &\mapsto \begin{cases} \\
MD &= \text{PARTIAL\_FUNCTION} \times \text{PARTIAL\_FUNCTION} \times \mathcal{P}(\text{DI-Model}) \to B \\
CNSTR &= \text{true} \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
SAVE &= \{ \\
\text{EMPTY\_PARTIAL\_FUNCTION} &\mapsto \begin{cases} \\
VSIG &= \text{PARTIAL\_FUNCTION} \\
VAL &= \{\} \\
\end{cases} \\
\text{DEFINED\_IN} &\mapsto \begin{cases} \\
VSIG &= \text{Anonymous1} \\
VAL &= \text{<Function defined in Fig. 39.(b)>} \\
\end{cases} \\
\text{DIFFER\_ONLY\_AT} &\mapsto \begin{cases} \\
VSIG &= \text{Anonymous2} \\
VAL &= \text{<Function defined in Fig. 40.(c)>} \\
\end{cases}
\end{align*}
\]

Figure 43: PARTIAL\_FUNCTION\_THEORY\_TEMPLATE's Resulting Specification Adornment Instance
The type `PARTIAL_FUNCTION` in the resulting specification adornment instance has a type model that is the power set of pairs of domain and range items (e.g., $\mathcal{P}(\text{DI-Model} \times \text{RI-Model})$). The constraint over this mathematical domain is defined in box (a) of Figure 39. The resulting specification adornment instance also defines two other types, here named `Anonymous1` and `Anonymous2`. These types are used as the types of mathematical operations defined in the mathematics module, but are not specifically given program identifiers in the RESOLVE description of the template.

It is interesting to note that the math operations defined in the RESOLVE templates are interpreted as variables. Specification adornment environments do not contain "operations" per se, since a mathematical operation can be represented as a variable whose type is a space of functions.

### 5.4.2 Using the Communal_PMap_Template Abstract Template

The `Communal_PMap_Template` defined in Figures 36 through 38 uses the `PARTIAL_FUNCTION_THEORY TEMPLATE` specification adornment template. `Communal_PMap_Template` is a RESOLVE parameterized concept, most naturally denoted in ACTI by an abstract template. Figure 44 shows `Communal_PMap_Template` as an abstract tem-

![Diagram](image)

**Figure 44:** `Communal_PMap_Template` is an Abstract Template
plate, which takes a single abstract instance as a parameter and produces a single abstract instance as its result.

The **parametric context** section of *Communal_PMap_Template*, shown in Figure 36, explicitly describes the requirements placed on legitimate parameters to the template. Figure 45 graphically depicts the structure of any abstract instance that can be used as a parameter to *Communal_PMap_Template*. Each of the three distinct parameters from the template's parametric context is explicitly represented. This figure characterizes the domain predicate (*ATS.AIDP*) of the abstract template denoted by *Communal_PMap_Template*.

When *Communal_PMap_Template* is applied to a parameter of the form given in Figure 45, it produces a resulting abstract instance with the structure shown in Figures 46 and 47. The **CTXT** component of the resulting abstract instance defines all of the external dependencies that the resulting instance will have. Before the re-

<table>
<thead>
<tr>
<th><strong>CTXT</strong></th>
<th><strong>SAE</strong></th>
<th><strong>SATY</strong></th>
<th><strong>MD</strong></th>
<th><strong>CNSTR</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SAE</td>
<td>SATY</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>= math-integer→</td>
<td>MD</td>
<td>= Z</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SATY</td>
<td>CNSTR</td>
<td>= true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>SAE</strong></th>
<th><strong>SAVE</strong></th>
<th><strong>VSIG</strong></th>
<th><strong>VAL</strong></th>
<th><strong>SAEI</strong></th>
<th><strong>VSIG</strong></th>
<th><strong>VAL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>SAE</td>
<td>SAVE</td>
<td>VSIG</td>
<td>VAL</td>
<td>SAEI</td>
<td>VSIG</td>
<td>VAL</td>
</tr>
<tr>
<td></td>
<td>= max.total.size→</td>
<td>=</td>
<td>=</td>
<td>= max.total.size &gt; 0</td>
<td>=</td>
<td>&lt;Any value &gt; 0&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>TE</strong></th>
<th><strong>D.Item</strong></th>
<th><strong>R.Item</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>TE</td>
<td>D.Item</td>
<td>R.Item</td>
</tr>
<tr>
<td>{</td>
<td>&lt;Any model, say DI-Model&gt;</td>
<td>&lt;Any model, say RI-Model&gt;</td>
</tr>
</tbody>
</table>

Figure 45: *Communal_PMap_Template*’s Parameter
Figure 46: Communal_PMap_Template’s Resulting Abstract Instance
\[
\begin{align*}
OES & = \{ \\
\text{Define} & \rightarrow \begin{cases} \\
PP = (\text{Partial\_Map}, \text{D\_Item}, \text{R\_Item}) & \text{DP} = \langle \text{See precondition in Fig. 37} \rangle \\
EP = \langle \text{See postcondition in Fig. 37} \rangle \\
\end{cases} \\
\text{Undefine} & \rightarrow \begin{cases} \\
PP = (\text{Partial\_Map}, \text{D\_Item}, \text{D\_Item}, \text{R\_Item}) & \text{DP} = \langle \text{See precondition in Fig. 37} \rangle \\
EP = \langle \text{See postcondition in Fig. 37} \rangle \\
\end{cases} \\
\text{Undefine\_Any\_One} & \rightarrow \begin{cases} \\
PP = (\text{Partial\_Map}, \text{D\_Item}, \text{R\_Item}) & \text{DP} = \langle \text{See precondition in Fig. 37} \rangle \\
EP = \langle \text{See postcondition in Fig. 37} \rangle \\
\end{cases} \\
\text{Is\_Defined} & \rightarrow \begin{cases} \\
PP = (\text{Partial\_Map}, \text{D\_Item}, \text{prog\_Boolean}) & \text{DP} = \text{true} \\
EP = \langle \text{See postcondition in Fig. 37} \rangle \\
\end{cases} \\
\text{Size} & \rightarrow \begin{cases} \\
PP = (\text{Partial\_Map}, \text{prog\_Integer}) & \text{DP} = \text{true} \\
EP = \langle \text{See postcondition in Fig. 37} \rangle \\
\end{cases} \\
\text{Initialize} & \rightarrow \begin{cases} \\
PP = (\text{Partial\_Map}) & \text{DP} = \text{true} \\
EP = \langle \text{See postcondition in Fig. 37} \rangle \\
\end{cases} \\
\text{Finalize} & \rightarrow \ldots \\
\text{Swap} & \rightarrow \ldots
\end{align*}
\]

Figure 47: Communal\_PMap\_Template's Resulting Abstract Instance (continued)
resulting instance can be embedded in some other environment, such as BE, external entities will have to be mapped onto this context interface using an interpretation mapping.

The SAE component of the resulting abstract instance shows the specification adornment definitions it provides. These definitions include two integer-valued variables, \texttt{max\_total\_size} and \texttt{total\_size}. \texttt{Max\_total\_size} records the value provided in the parameter to the template, while \texttt{total\_size} is a specification variable (as opposed to a program variable) described in the state variables section of the Communal_PMap_Template concept in Figure 36. The SAE component of the resulting instance also associates the name \texttt{PARTIAL\_FUNCTION\_THEORY} with the result of applying \texttt{PARTIAL\_FUNCTION\_THEORY\_TEMPLATE} on the given parameters.

The TE component of the resulting abstract instance contains the single program type that it provides, \texttt{Partial\_Map}. As shown in Figure 46, the type model associated with this identifier is the power set of pairs of domain and range items. Figure 47 shows the OES component of the resulting abstract instance, which contains the operation models for all of the operations the instance provides. Each operation is associated with a parameter profile, domain predicate, and effect predicate derived directly from the corresponding behavioral specification in the Communal_PMap_Template concept. Note that \texttt{Initialize}, \texttt{Finalize}, and \texttt{Swap} operations are also provided here, even though they are not explicitly described in the RESOLVE concept specification. In RESOLVE, these three operations must be provided for every type [9]. The behavior of the \texttt{Initialize} and \texttt{Finalize} operations are described as part of the
corresponding type’s declaration. Swap always has the same behavior: it exchanges the abstract values of its two operands.

To instantiate Communal.PMap.Template, we need only provide it with a parameter matching the structure of Figure 45. Unfortunately, in the basis environment BE, there is no such parameter. But the instantiation statement in Figure 31 explains exactly which entity in BE is to be substituted for each parameter to Communal.PMap.Template. What that parameter substitution information describes is a way to interpret (a subset of) BE as a valid parameter to Communal.PMap.Template. This information can be rephrased as an interpretation mapping.

\[
CPMT-Parm-IM \equiv
\]
\[
\begin{align*}
DP &= \{(AI_S, AI_G) : AI_S = BE \text{ and } AI_G = CPMap-Params \}
\end{align*}
\]
\[
TYmap = \begin{cases} 
D._{Item} \mapsto & \begin{cases} 
N = \text{prog-Integer} \\
R = \{(d_1, d_2) : d_1 = d_2\} \\
C = \text{true} 
\end{cases} \\
R._{Item} \mapsto & \begin{cases} 
N = \text{prog-Integer} \\
R = \{(r_1, r_2) : r_1 = r_2\} \\
C = \text{true} 
\end{cases}
\end{cases}
\]
\[
MCORR = \text{SAVariableModel(size.limit, } AI_S).VAL = \text{SAVariableModel(max.total.size, } AI_G).VAL
\]
\[
MCONV = 0 \leq \text{SAVariableModel(size.limit, } AI_S).VAL
\]

Figure 48: An Interpretation Mapping for Communal.PMap.Template’s Parameter

Figure 48 shows the interpretation mapping implicitly described by the arguments to Communal.PMap.Template in Figure 31. This interpretation mapping ex-
explains specifically how to interpret $BE.E$ as having the form shown in Figure 45. Let $CPMT-Params$ represent an abstract instance exactly like Figure 45, but with its $DI-Model$ and $RI-Model$ both $(Z, true)$.

The interpretation mapping $CPMT-Parm-IM$ has a domain predicate ($DP$) that is only true for $BE$ and $CPMap-Params$. In Figure 48, conventional set notation is used to describe the domain predicate, with the understanding that the predicate is true for all argument tuples in the set, and false for others. The $TE$ component of the mapping describes the type-to-type interpretations, including a correspondence relation ($R$) and convention ($C$) for each one. In this case, prog-Integer is mapped to both $D.Item$ and $R.Item$, using an identity correspondence. The $MCORR$ predicate describes the relationship between the variable values in the two abstract instances, and the $MCONV$ represents an invariant that must be true of the subset of $BE$ involved in the mapping.

Using the $\downarrow$ operator described in Section 4.13.3, it is now possible to reshape $BE$ so that it is in an appropriate form, and then apply the template:

$$CPMap-Instance \equiv \text{Communal.PMap.Template(}
\text{abstract}(BE \downarrow CPMT-Parm-IM \ CPMT-Params)) \quad (5.1)$$

The result is $CPMap-Instance$, an abstract instance of the form shown in Figures 46 and 47.
5.5 Instantiating Unordered_Stack

The second step in the Int_Multiset instantiation is applying the concrete template denoted by Unordered_Stack, a RESOLVE realization of the Communal_PMap_Template. This concrete template takes a single concrete instance as a parameter and produces a single concrete instance as its result.

realization header Unordered_Stack for Communal_PMap_Template

context

global context
conce p Record2_Template
concept Stack_Template

parametric context

facility D_R_Pair_Facility is
   Record2_Template (D.Item, R.Item)

facility Stack_Facility is
   Stack_Template (Record2)

operation Are_Equal (  
   preserves d1 : D.Item
   preserves d2 : D.Item
) : Boolean
ensures Are_Equal iff d1 = d2

end Unordered_Stack

Figure 49: The Unordered_Stack Realization Header
The Unordered.Stack realization presented here is derived directly from the unbounded version described by Bucci et al. [2, pp. 42-44]. Figure 49 contains the realization header for Unordered.Stack. Every RESOLVE realization has a realization header, which contains all of the information a client programmer needs in order to use the realization. This includes describing the **parametric context** of the realization, which captures the essence of the single parameter to the concrete instance.

Figure 50 graphically depicts the structure of any concrete instance that can be used as a parameter to Unordered.Stack. Note that in RESOLVE, realizations automatically include the parameters to their corresponding concepts, so all of the

| CTXT = SAE = SATE = {math-integer| MD = Z CNSTR = true} |
| SAE = \[
\begin{align*}
SAE &= \{ \max_{\text{total size}} \rightarrow [\text{VSIG} = \text{math-integer}] \\
&\quad \text{VAL} = \langle \text{Any value} > 0 > \\
&\quad \text{EI} = \max_{\text{total size}} > 0
\end{align*}
\]

| TE = \{
| D.Item \mapsto <\text{Any model, say DI-Model}> \\
| R.Item \mapsto <\text{Any model, say RI-Model}> \\
| PP = \langle\text{D.Item, D.Item, prog-Boolean}\rangle |
| OMOD = \langle DP = \text{true}\rangle |
| EP = <\text{See postcondition in Fig. 49}> |

| OE = \{
| Are.Equal \mapsto PP = \langle\text{D.Item, D.Item, prog-Boolean}\rangle |
| OMOD = \langle DP = \text{true}\rangle |
| PR = \ldots |
| SF = \ldots |

| CIE = \{
| D.R.Pair.Facility \mapsto <\text{See Fig. 51}> |
| Stack.Facility \mapsto <\text{See Fig. 52}> |

Figure 50: Unordered.Stack's Parameter
Communal_PMap_Template parameters have been explicitly duplicated in Figure 50. In addition to the three parameters from Communal_PMap_Template's parametric context, Figure 50 includes the three additional parameters from the realization header: an Are_Equal operation, a D.R_Pair_Facility, and a Stack_Facility.

The Are_Equal operation is shown in the operation environment of the concrete instance in Figure 50. The operation meaning associated with Are_Equal includes the operation model defined by the specification in the realization header, along with the procedure relation and status function computed by the actual operation substituted for this parameter.

D.R_Pair_Facility is shown as a context-bound concrete instance in the CIE component of the parameter concrete instance. This context-bound concrete instance is shown in Figure 51. The concrete instance component (CI) of D.R_Pair_Facility provides a type Record2, and the appropriate operations for that type, as defined by the RESOLVE concept Record2_Template used to specify this parameter in the realization header. Further, this facility depends on two types defined outside of it: Item1 and Item2, the types of the two fields in each Record2 object. The interpretation mapping component (IM) of D.R_Pair_Facility describes how the context of its CI component is fulfilled by the remainder of the parameter concrete instance shown in Figure 50. As shown in Figure 51, the type D.Item from the parameter concrete instance is bound to the Item1 type in D.R_Pair_Facility's context, and R.Item is bound to Item2. Thus, an object of the Record2 type provided by D.R_Pair_Facility contains a D.Item and R.Item pair.
Figure 51: The D.R._Pair._Facility Context-Bound Concrete Instance

Figure 52: The Stack._Facility Context-Bound Concrete Instance
Figure 52 shows the Stack_Facility context-bound concrete instance. The concrete instance component ($CI$) of Stack_Facility provides the types and operations described by the corresponding RESOLVE concept used to specify this parameter in the realization header. The interpretation mapping component binds the context of the $CI$ component, just as before.

Figures 53 through 57 give the complete RESOLVE code for the Unordered_S tack realization body. The main difference between this realization and the one presented by Bucci et al. is that here, an internal global state variable communal_length (Figure 55) is used to track the cumulative size of all Partial_Map objects combined. The code for the various operations provided by this realization show how this state variable is updated.

In addition to having an internal state variable, the Unordered_S tack realization also includes two internal operations, Pop_Until_Passed and Combine. These operations are used locally within the realization, but in RESOLVE they are not exported outside the realization. Also, note that here no implementations for Initialize, Finalize, or Swap on Partial_Map objects are provided. This is because a RESOLVE compiler could automatically use the corresponding operations on the Stack type being used as the Partial_Map representation, and in this case the appropriate behavior would be exhibited. If this did not provide the appropriate behavior, the programmer writing this realization would be obliged to provide an implementation herself.
realization body Unordered Stack for Communal PMap Template

context

global context
mathematics STRING_OCCURS_COUNT_MACHINERY
mathematics STRING_REVERSE_MACHINERY
mathematics STRING_ELEMENTS_MACHINERY

local context
renaming type Record2 as D_R_Pair
with subcomponent field1 as domain.value
with subcomponent field2 as range.value

math facility STRING_OCCURS_COUNT_FACILITY is
STRING_OCCURS_COUNT_MACHINERY (math[D_R_Pair])

math facility STRING_REVERSE_FACILITY is
STRING_REVERSE_MACHINERY (math[D_R_Pair])

math facility STRING_ELEMENTS_FACILITY is
STRING_ELEMENTS_MACHINERY (math[D_R_Pair])

math operation CONTAINS (s : string of math[D_R_Pair],
d : math[D_Item]) : boolean
explicit definition
there exists r : math[R.Item]
  (OCCURS_COUNT (s, (d,r)) > 0)

math operation CONTAINS_AS_FIRST_ENTRY (s : string of math[D_R_Pair],
d : math[D_Item]) : boolean
explicit definition
there exists r: math[R.Item],
  t : string of math[D_R_Pair]
  (s = <(d,r)> * t)

Figure 53: The Unordered Stack Realization Body
operation Pop_Until_Passed (  
    alters s1 : Stack  
    alters s2 : Stack  
    preserves d : D.Item  
)  
requires s2 = empty_string  
ensures ELEMENTS (s1 * s2) = ELEMENTS (#s1) and  
  (if s2 = empty_string  
    then not CONTAINS (s1, d)  
    else CONTAINS_AS_FIRST_ENTRY (s2, d))  
context  
variables  
  found : Boolean  
  p : D.R_Pair  
begin  
  loop maintaining  
    not CONTAINS (s2, d) and  
    REVERSE (s2) * s1 = REVERSE (#s2) * #s1  
    if Length (s1) = 0 then  
      s1 :=: s2  
      exit  
    end if  
  Pop (s1, p)  
  found := Are_Equal (p.domain_value, d)  
  Push (s2, p)  
  if found then  
    exit  
  end if  
end loop  
end Pop_Until_Passed  

operation Combine (  
    alters s1 : Stack  
    consumes s2 : Stack  
)  
ensures s1 = REVERSE (#s2) * #s1  

Figure 54: The Unordered_Stack Realization Body (continued)
context
  variables
    p : D.R_Pair
begin
  loop maintaining
    \text{REVERSE} (s2) * s1 = \text{REVERSE} (#s2) * #s1
  while Length (s2) > 0 do
    Pop (s2, p)
    Push (si, p)
  end loop
end Combine
end

state variables
  communal.length : Integer
convention
  0 \leq \text{communal.length} \leq \text{max.total.size}
correspondence
  total.size = \text{communal.length}

interface

  type Partial\_Map is represented by Stack
convention
  for all d : math[D.Item], r : math[R.Item]
    (\text{OCCURS.COUNT} (m.rep, (d,r)) \leq 1)
correspondence
  m = \text{ELEMENTS} (m.rep)

operation Define (
  alters m : Partial\_Map
  consumes d : D.Item
  consumes r : R.Item
)
begin
  Push (m.rep, (d, r))
  \text{communal.length} := \text{communal.length} + 1
end Define

Figure 55: The Unordered.Stack Realization Body (continued)
operation Undefine (  
    alterns  m : Partial.Map  
    preserves  d : D.Item  
    produces  d.copy : D.Item  
    produces  r : R.Item  
  )
context
variables
  catalyst : Stack
begin
  Pop_Until_Passed (m.rep, catalyst, d)
  Pop (catalyst, (d.copy, r))
  Combine (m.rep, catalyst)
  communal.length := communal.length - 1
end Undefine

operation Undefine.Any.One (  
    alters  m : Partial.Map  
    produces  d : D.Item  
    produces  r : R.Item  
  )
begin
  Pop (m.rep, (d, r))
  communal.length := communal.length - 1
end Undefine.Any.One

operation Is_Defined (  
    preserves  m : Partial.Map  
    preserves  d : D.Item  
  ) : Boolean
context
variables
  catalyst : Stack
  defined : Boolean
begin
  Pop_Until_Passed (m.rep, catalyst, d)

Figure 56: The Unordered.Stack Realization Body (continued)
if Length (catalyst) > 0 then
  Combine (m.rep, catalyst)
defined := true
end if
return defined
end Is_Defined

operation Size ( preserves m : Partial_Map ) : Integer
begin
  return Length (m.rep)
end Size

end Unordered_Stack

Figure 57: The Unordered_Stack Realization Body (continued)

Just as the realization header characterizes the acceptable parameters to the
Unordered_Stack concrete template, the realization body characterizes the result­
ing concrete instances. Figure 58 presents an outline of the structure of a resulting
concrete instance. Many of the details are elided here for simplicity.

As with the result of Communal_PMap_Template, the CTXT component describes
all of the external dependencies of the resulting concrete instance. These dependencies
can be taken almost directly from the global context sections of the realization body
and the concept, and from the parameters to Unordered_Stack. The concept’s global
context must be included here because in RESOLVE, the realization implicitly imports
everything the corresponding concept does. Thus, the CTXT for Unordered_Stack’s
resulting instance will depend on D_Item, R_Item, max_total_size, Standard_Bool-
Figure 58: Unordered.Stack's Resulting Concrete Instance

ean_Facility, Standard_Integer_Facility, D.R_Pair_Facility, Stack_Facility, and assorted parameterized mathematics modules. Similarly, the SAE component of the resulting concrete instance will contain all of the mathematics definitions introduced in the local context section of the Unordered.Stack realization body.
The type, variable, and operation environments in the resulting concrete instance are where the differences from `Communal_PMap_Template` are most noticeable. In Figure 58 we see that the type `Partial_Map` in the realization is modeled as a string of `Items` (e.g., the space "Item"), which is the mathematical model for a stack exported by the `Stack_Facility` parameter. Further, all operations in the realization have behavioral specifications and implementations in terms of this type model. We also see that `communal.length` is an integer-valued program variable, and it is exposed along with all of the "internal" operations.

In RESOLVE, no implementation-level behavioral specifications (in terms of `Stack` objects) are given for the operations on `Partial_Map`. Instead, these operations are obligated to conform to the more abstract behavioral descriptions in the corresponding concept. In ACTI, however, concrete instances and abstract instances are fully independent, meaning that the `Unordered.Stack` concrete instance might be used to supply behavior conforming with any number of different abstract instances. As a result, we cannot depend on any one abstract instance as the target when defining the type and operation models used in the concrete instance. We know that, in theory, there is a minimal precondition and maximal postcondition for the code of each operation in the `Unordered.Stack` realization, expressible in terms of the `Stack` objects being manipulated. The RESOLVE verification rules do not require that these be made explicit, but nevertheless they are available for use in the denotation of `Unordered.Stack`. 
Also, because abstract and concrete instances are independent, there is no notion of "local" or "internal" resources in an ACTI concrete instance. An operation like \texttt{Combine}, or a variable like \texttt{communal.length}, is only "internal" because it is not exported. The question of what is exported can only be answered relative to an interface. Thus, the resulting concrete instance produced by an \texttt{Unordered.Stack} instantiation in ACTI makes all of its variables and operations available. It is only

\begin{align*}
US\text{-}Parm\text{-}IM & \equiv \\
DP & = \{(AIS, AI_G) : AIS = BE \text{ and } AI_G = US\text{-}Params\} \\
\quad \downarrow \quad \downarrow \\
TYmap & = \begin{cases}
N = \text{prog-Integer} \\
D\text{.Item} \mapsto R = \{(d_1, d_2) : d_1 = d_2\} \\
C = \text{true} \\
N = \text{prog-Integer} \\
R\text{.Item} \mapsto R = \{(r_1, r_2) : r_1 = r_2\} \\
C = \text{true} \\
N = \text{Int\_Int\_Record} \\
Record2 \mapsto R = \{(r_1, r_2) : r_1 = r_2\} \\
C = \text{true} \\
N = \text{Int\_Int\_Stack} \\
Stack \mapsto R = \{(s_1, s_2) : s_1 = s_2\} \\
C = \text{true} \\
\end{cases} \\
OPmap & = \begin{cases}
\text{Swap\_Field1} \mapsto \ldots \\
\text{Swap\_Field2} \mapsto \ldots \\
\text{Push} \mapsto \ldots \\
\vdots
\end{cases} \\
MCORR & = \text{SAVariableModel(size\_limit, AIS)}\text{.VAL} = \text{SAVariableModel(max\_total\_size, AI_G)}\text{.VAL} \\
MCONV & = 0 \leq \text{SAVariableModel(size\_limit, AIS)}\text{.VAL}
\end{align*}

Figure 59: An Interpretation Mapping for \texttt{Unordered.Stack}'s Parameter
when that concrete instance is combined with a more restrictive abstract instance that the notion of “internal” or “internally hidden” comes into play.

Figure 59 shows the interpretation mapping implicitly described by the arguments to Unordered_Stack in Figure 31. This interpretation mapping explains specifically how to interpret BE.E as having the form shown in Figure 50. Let US-Params represent a concrete instance exactly like Figure 50, but with its DI-Model and RI-Model both \((Z, \text{true})\).

The interpretation mapping US-Parm-IM has a domain predicate \((DP)\) that is only true for BE and US-Params. The remainder of the interpretation mapping is exactly analogous to the one used in instantiating Communal_PMap_Template, shown in Figure 48. Using the \(\downarrow\) operator, one can now reshape BE so that it is in an appropriate form, and then apply the template:

\[
U\text{Stack-Instance} \equiv \\
\text{Unordered_Stack(} \\
\text{BE} \downarrow \text{US-Parm-IM} \text{ US-Params)}
\]

The result is UStack-Instance, a concrete instance of the form outlined in Figure 58.

5.6 Instantiating UStack_To_CMap

The third step in the Int_Multiset instantiation is applying the implementation mapping template UStack_To_CMap. In the current version of RESOLVE, no separate construct exists to represent such a parameterized mapping, or give it a name\(^2\). However, RESOLVE does require programmers to provide all of the information nec-

\(^2\)One has been proposed as a result of this research; Figures 60 and 61 show an example.
necessary to construct such an interpretation, and it is all recorded in the realization body.

RESOLVE has explicit syntactic slots for recording type-specific correspondence and convention assertions, as well as subsystem-level correspondence and convention assertions. RESOLVE also implicitly restricts type and operation interpretation mappings by requiring the programmer to use the same program identifier for a type or operation in both a concept and a realization when the two are connected. For the\texttt{Unordered.Stack} realization body in Figures 53 through 57, those bits and pieces can be pulled out and repackaged into a RESOLVE-like description of an interpretation mapping template. The result is shown in Figures 60 and 61.

\texttt{UStack.To.CPMap} is a parameterized interpretation mapping. It has the same parametric context as \texttt{Communal.PMap.Template}, and for any RESOLVE instantiation of \texttt{Communal.PMap.Template} realized by \texttt{Unordered.Stack}, \texttt{UStack.To.CPMap} will generate the corresponding interpretation mapping. In the \texttt{mapping} section of \texttt{UStack.To.CPMap}, the \texttt{from} and \texttt{to} clauses give local names to the corresponding instances of \texttt{Communal.PMap.Template} and \texttt{Unordered.Stack} for use in defining the mapping. Here they are called \texttt{R} (for realization) and \texttt{C} (for concept), respectively. Immediately below the \texttt{from} and \texttt{to} clauses, the correspondence and convention for the subsystem-level state can be specified. These syntactic slots are the syntactic analog of the \texttt{MCORR} and \texttt{MCONV} components of an ACTI interpretation mapping. \texttt{UStack.To.CPMap} then describes the type interpretation mapping and the operation interpretation mapping, both of which naturally follow from the \texttt{Unordered.Stack}
interpretation UStack_To_CMap

context
  local context
    math facility STRING_OCCURS_COUNT_FACILITY is
      STRING_OCCURS_COUNT_MACHINERY (math[D.Item], math[R.Item])
    math facility STRING_ELEMENTS_FACILITY is
      STRING_ELEMENTS_MACHINERY (math[D.Item], math[R.Item])

mapping
  from R is Unordered_Stack (D.Item, R.Item, max.total.size, *, *, *)
  to C is Communal_PMap_Template (D.Item, R.Item, max.total.size)

concept convention
  0 ≤ communal.length ≤ max.total.size

concept correspondence
  C.total.size = R.communal.length

Figure 60: The UStack_To_CMap Interpretation Mapping
types
R.Partial.Map to C.Partial.Map
exemplar from m_rep to m
convention
for all $d : \text{math}[D.Item], r : \text{math}[R.Item]$
\quad (\text{OCCURS}_\text{COUNT}(\text{m_rep}, (d,r)) \leq 1)$
correspondence
\quad m = \text{ELEMENTS}(m\_rep)$

operations
R.Define to C.Define
R.Undefine to C.Undefine
R.Undefine.Any.One to C.Undefine.Any.One
R.Is_Defined to C.Is_Defined
R.Size to C.Size

R.Initialize to C.Initialize
R.Finalize to C.Finalize
R.Swap to C.Swap

end UStack.To.CPMap

Figure 61: The UStack.To.CPMap Interpretation Mapping (continued)

realization body.

Once this information has been presented separately from the realization body itself, it should be clear how UStack.To.CPMap can be interpreted in ACTI. It is an interpretation mapping template, which takes a single abstract instance as a parameter and produces a single interpretation mapping as a result. The parametric context of UStack.To.CPMap is the same as that for Communal.PMap.Template, so the structure of its parameter is characterized by Figure 45. The resulting interpretation mapping is shown in Figure 62.
\[ DP = \{(A_{IS}, A_{IG}) : A_{IS} = R \text{ and } A_{IG} = C\} \]

\[ TYmap = \begin{cases} 
N = R.\text{Partial\_Map} \\
R = \{(m, m.\text{rep}) : m = \text{ELEMENTS}(m.\text{rep})\} \\
C = \{m.\text{rep} : \forall d \in \text{DI\_Model}, r \in \text{RI\_Model} \text{ OCCURS\_COUNT}(m.\text{rep}, (d, r)) \leq 1\} 
\end{cases} \]

\[ OPmap = \{ \\
\text{C.\_Define} \mapsto R.\text{Define} \\
\text{C.\_Undefine} \mapsto R.\text{Undefine} \\
\text{C.\_Undefine\_Any\_One} \mapsto R.\text{Undefine\_Any\_One} \\
\text{C.\_Is\_Defined} \mapsto R.\text{Is\_Defined} \\
\text{C.\_Size} \mapsto R.\text{Size} \\
\text{C.\_Initialize} \mapsto R.\text{Initialize} \\
\text{C.\_Finalize} \mapsto R.\text{Finalize} \\
\text{C.\_Swap} \mapsto R.\text{Swap} 
\} \]

\[ MCORR = C.\text{total\_size} = R.\text{communal\_length} \]
\[ MCONV = 0 \leq R.\text{communal\_length} \leq R.\text{max\_total\_size} \]

Figure 62: The Meaning of the Result of \text{UStack\_To\_CPMap} in ACTI

Because the parameters to \text{UStack\_To\_CPMap} have the same form as those to \text{Communal\_PMap\_Template}, we can simply reuse the earlier \text{CPMT\_Parm\_IM} interpretation mapping to cast \textit{BE} in the appropriate form, and then apply the interpretation mapping template:

\[ US2CPM-\text{Instance} \equiv \text{UStack\_To\_CPMap} (\text{abstract}(\text{BE} \downarrow \text{CPMT\_Parm\_IM} \text{ CPMT\_Params})) \quad (5.3) \]

The result is \textit{US2CPM-\text{Instance}}, an interpretation mapping of the form shown in Figure 62.
5.7 Combining the Instances and Binding Context

Now that Communal_PMap_Template, Unordered_Stack, and UStack_To_CMap have all been instantiated, they can be combined to form a new concrete instance:

\[
\text{Int. Multiset-Instance} \equiv \text{abstract}(\text{UStack-Instance} \downarrow \text{US2CPM-Instance} \quad \text{CMap-Instance})
\]

This new instance has the implementation-level procedure relations and status functions defined in the Unordered_Stack realization, but the abstract behavioral descriptions it contains are those from Communal_PMap_Template. Further, the \( \downarrow \) operator has removed all of the features of the concrete instance produced by the realization that were not described in the abstract instance produced by the concept. Thus, Combine, Pop_Until_Passed, and communal_length are not present in Int_Multiset_Instance. This is the way RESOLVE's hidden, internal resources can be modeled in ACTI.

Now all that remains is to convert Int_Multiset_Instance into a context-bound concrete instance so that it can be embedded in BE.CIE, as described in Section 4.13.6. This is done by providing an interpretation mapping that explains how BE.E can be interpreted as Int_Multiset_Instance.CTXT. Together, the interpretation mapping and Int_Multiset_Instance form a context-bound concrete instance that can be embedded as a new entry in BE.CIE. This produces the basis environment BE', which is the final program state achieved after the execution of the entire Int_Multiset instantiation.
5.8 Chapter Summary

In this chapter, an ACTI interpretation of the semantics of a RESOLVE instantiation was provided. This example helps to show some of the more advanced features of ACTI, including the use of interpretation mappings, the ↓ operator, and the abstract operator. Further, this example highlights the fact that in ACTI, concrete instances (implementations) are completely independent from abstract instances (specifications), and the relationships between instances are independent from the instances involved. This means that in addition to allowing multiple realizations for one concept, ACTI allows multiple concepts for the same realization, and even multiple interpretation mappings for a single concept-realization pair.

Also, it is clear that the key ideas in ACTI are only the core primitives needed to understand the semantics of subsystems. In a real programming language, the statements provided for writing software may have semantic denotations that are complex compositions of a series of more primitive steps describable in the terms presented in Chapter IV.
CHAPTER VI

Conclusions

This chapter summarizes the research conducted for this dissertation and presents the conclusions drawn from it. It continues by presenting the contributions of this research to the field of computer science, and concludes with a discussion of future research directions.

6.1 Summary

This dissertation defends the thesis that conventional programming languages are inadequate for constructing large, complex software systems that are "understandable." Chapter II explains the basis for this claim, namely that modern programming languages:

- Treat module-like structures as syntactic mechanisms whose primary purpose is grouping declarations and controlling visibility; and
- Give other program elements, such as procedures, meanings that are hierarchically constructed on the basis of their implementations.

While these are reasonable choices when designing a language to instruct computers, they do not address the cognitive limitations humans face when trying to understand
complex artifacts. In order to address these concerns, it is necessary to assign independent meanings to individual software building-blocks, to separate the abstract description of a software part’s behavior from its implementation, and to provide a mechanism for explaining how the implementation of the part provides behavior consistent with that abstract description.

Chapter III introduced a new language-independent model of software called ACTI. ACTI stands for “Abstract and Concrete Templates and Instances,” the four principle classes of software subsystems it defines. ACTI addresses the limitations of conventional programming languages described above because:

- In ACTI, each software subsystem (building-block) is given an intrinsic meaning; it is not just a syntactic construct. This meaning encompasses an abstract behavioral description of all the visible entities within a subsystem.

- The meaning of a software subsystem is not synthesized from the meanings of the lower-level subsystems it is built upon, and is completely independent of how the subsystem is implemented.

The introduction to the model presented in Chapter III was centered around a running code example, and was presented in informal, intuitive terms.

Chapter IV then provided a formal definition of ACTI. This model’s development followed the approach of denotational semantics, defining mathematical spaces of objects intended to represent the meanings of program structures such as specifications, implementations, and their relationships. ACTI was developed with the phase dis-
tinction between compile-time and run-time semantics in mind, and it only addresses the dynamic, execution-time aspects of module meanings.

Finally, Chapter V continued the running example that was begun in Chapter III to demonstrate some of the more advanced aspects of ACTI. It showed the implementation of a subsystem (a concrete instance), and the interpretation mapping explaining why that implementation was consistent with the abstract "cover story" described in the corresponding specification (an abstract instance).

The remainder of the research presented in this dissertation consists of three principle components: a discussion of five previous software models; a check list that represents an operational definition of the necessities for effectively structuring software; and an analysis of the ACTI model. Together, these parts form a defense of the claim that ACTI is a mathematically formal, language-independent model of software components that captures the underlying conceptual view of software architecture embedded in modern module-structured languages.

Appendix A gives overviews of the five prior models of software chosen as representative of previous work. The five are: the 3C model of reusable software, OBJ, RESOLVE, Eiffel, and Standard ML. Of these five "models," four are actually particular programming languages, of which two are functional programming languages, one is object-oriented, one is object-based, two support formal specification, one more supports informal specification, one supports formal verification, and three support parameterized programming. These five models were chosen because they represent the breadth of what researchers in computer science have said about software con-
struction, and how to best address the architectural problems of large-scale software systems.

Appendix B then takes these five models and uses them to devise an operational means of measuring the extent to which any model addresses the intuitive notions about software construction and reuse that have been voiced. These intuitive notions are captured qualitatively, but in the more rigorous form of a structured check list of 22 properties that any universally applicable model of software construction should have. Each property in the list is justified, and cross-checked against each of the previous models, and also against the viewpoints of various software engineering professionals who build, use, and manage software components. Finally, a summary of how each of the prior five models measures up against the check list is presented in Table 3.

Appendix C then puts ACTI to the same test, determining how well it measures up against the structured check list developed in Appendix B. Unlike prior models, ACTI possesses all of the 32 properties in the check list. It provides support for abstractions by modeling types, instance subsystems, and template subsystems. It faithfully supports separation of concerns by making all subsystems independent, requiring explicit context interfaces, and elevating the “is-implemented-by” relationship to a first-class object. Through interpretation mappings, it provides an intensional mechanism for binding subsystems together and defining behavioral subtyping relationships between types or subsystems. It also provides convenient operators for programming by difference, or deriving new software objects from existing ones, while completely separating this mechanism from interpretation mappings. Appendix C then continues, explain-
ing how each of the previous models could be mapped into the framework of the ACTI model.

6.2 Conclusions

The focus of this dissertation has been the development of a programming language-independent, but formal, model of software construction that gives meaning to the notion of a software "building-block." Throughout this research effort, it has become apparent how profoundly the notations programmers use can affect the way they view the world. ACTI was developed to encompass the best of previous attacks on the software structuring problem while adding the capabilities necessary to capture abstract models of software parts, and it strives to strictly separate the distinct concepts involved. It does this to better support language designers by giving a more flexible, capable shape to their world view, so they can in turn help programmers the same way.

ACTI is general enough to encompass a wide variety of program semantics, especially the meanings of architectural structuring features. This means that both good and bad strategies of software structuring can be modeled. It follows that ACTI might provide a useful framework for talking about what strategies and mechanisms are most beneficial from the perspectives of software engineering in general and reuse in particular. Three of the most obvious architectural issues affecting reuse and software engineering are:

1. Appropriate separation of specification and implementation.
2. Support for parameterized programming.

3. Separating subtyping from programming by difference.

Each of these issues are discussed in the sections below.

6.2.1 Separation of Specification and Implementation

Perhaps the most critical insight about software structure embodied in ACTI is that a module is not simply a specification plus an implementation. At its most general, the specification-to-implementation relationship is many-to-many, and each specification and each implementation can have an independent meaning on its own.

The relationship between specifications and implementations is also a unique feature of ACTI. It encompasses both type-level and module-level representation invariants and abstraction functions (or, more properly, abstraction relations). Over and above this relationship, both specifications and implementations may contain their own constraints, at both the type and the module levels.

Finally, ACTI provides true abstraction for both types and modules, in the sense described by Harms [24]. In other words, ACTI provides more than the ability to hide representations and implementations; it allows a module to provide an abstract "cover story" describing how a type should be logically viewed, or how an operation behaves. The manner in which an implementation fulfills such a cover story is captured through interpretation mappings. All of these characteristics are in strong support of good software engineering.
6.2.2 Parameterized Programming

ACTI follows the work of Goguen [21] in providing complete support for parameterized programming. Abstract and concrete templates, or generic specifications and implementations, provide the cornerstone of this support. Both specifications and implementations can be parameterized independently, as in RESOLVE [51, pp. 23–24]. Further, ACTI's capabilities for modeling nested modules and templates means that ACTI has true high-order generics. Finally, the way template parameter requirements are represented in ACTI allows for generics with variable-length parameter lists and correspondingly variable output instances.

6.2.3 Subtyping and Derivations (Inheritance)

While ACTI was developed to incorporate all of the beneficial aspects of OOP strategies, it takes a distinctly nontraditional approach to inheritance. It clearly distinguishes between its specification derivation mechanisms, implementation derivation mechanisms, and is-a relationships, unlike most traditional OO languages which use a single inheritance mechanism for all three [8]. By also providing support for modeling polymorphic type representations and dynamic binding behavior, ACTI manages to provide all of the OO features while avoiding the problems than can arise if they are all co-mingled in a single mechanism.

6.3 Contributions

The primary contribution of this research to the field of computer science is ACTI itself. Secondarily, the structured check list developed and described in Appendix B is
also a contribution, and can be extended on its own to form a more refined operational
definition of the requirements for a general purpose model of software.

As a model of software construction, ACTI is unique because:

1. It promotes the creation of complex software systems that are understandable.

   This is achieved by both supporting the formation of mental models of software
   parts, and addressing the inherent human (cognitive) limitations in dealing with
   those mental models.

2. It gives a real semantic denotation to “modules,” which are the building-blocks
   from which software is composed. This denotation identifies just what “mod­
   ules” should be so that they are meaningful, allow sophisticated composition,
   and present simple (not synthesized) conceptual models.

3. It identifies “interpretation mappings” as the mechanism that allows one to
   clearly represent why an abstraction (a specification with a simple model) cor­
   rectly describes the composite behavior of a complex combination of lower-level
   software parts (an implementation).

4. It is language-independent, and unifies the concepts behind object-oriented pro­
   gramming and more traditional module-based programming. Thus, it solves the
   problem of understandable software composition across module- and class-based
   languages.
5. It solves the problems normally associated with "inheritance" [8] by separating "programming by difference" from type-to-type and module-to-module subtyping relationships.

6. The research approach started with an explicit analysis of needs, and development of the model was then driven by these stated requirements, rather than by the features of a given language.

7. It strives for complete separation of concerns, which was not achieved by any of the prior models (see Table 3).

Further, ACTI builds on previous work:

1. It gives software modules a real semantic denotation.

2. It treats software module specifications as complete behavioral descriptions, not just "types for modules."

3. It treats relationships between modules as more than simple extension relationships. Interpretation mappings effectively tell one how to "interpret" lower-level module structures at the more abstract specification level.

4. It provides complete support for parameterized programming.

5. It supports modularly verifiable software.

6. It focuses on modularity of software construction.
These contributions are important for three reasons. First, ACTI can form the basis of a denotational semantic model for any module-based language, including an OOPL. Second, if ACTI is used as the starting point for such a semantic model, it helps to ensure that the lessons from past ventures in this area will be carried into new development. Third, and most importantly, if ACTI is used as such a starting point, it will help to ensure the resulting language has a vision for the meaning of software parts and how they contribute to the understandability of a complex software system.

6.4 Future Research

There are many promising possibilities for future work based on ACTI. A few of the most interesting areas are discussed here.

RESOLVE’s Denotational Semantics. Since ACTI was designed using the denotational approach, the next logical step is to use it as (part of) the semantic definition for a language designed to support effective software construction. RESOLVE is just such a language, and a worthwhile extension of this research would use ACTI as the dynamic semantics for RESOLVE. All that is necessary is to define a corresponding static semantics to form a complete language definition. Some facets of ACTI might also instigate refinements of some of RESOLVE’s language features, or even spur the addition of new features.

Language Features. Some of the mathematical spaces in ACTI are more expressive than any of the corresponding features the author has observed in programming
languages. For example, consider an abstract template, which takes an abstract instance as a parameter. The allowable abstract instances to which this template may be applied are described by a domain predicate, the AIDP component of the template. This is a boolean-valued function over the space of all abstract instances. Thus, in ACTI it is possible to model generics that are applied to a variety of parameters, not all of which have to have the same structure. This means that the same generic specification (or implementation) could operate on modules that contain different numbers of types, operations, etc., with different names, and so on.

The simplest ramification of this expressiveness is that ACTI templates can operate on "variable length" parameter lists—a single template could operate on a module with one type or with ten types and generate an appropriate output in each case, and similarly for variable numbers of operations, nested modules, or even nested templates. Further, in ACTI it is possible to define an abstract template that, given any abstract instance that exports at least one type, generates another abstract instance that is identical, but also exports Copy and Is_Equal operations for that type (or even for every type in the parameter). No programming language at the time of this writing allows one to define generics that are parameterized this flexibly.

The generality of such mathematical spaces in ACTI may point at new language features, or variations on old ones, for exploration.

Software Architecture Issues. Over time, it seems that some software practices have turned out to be more beneficial than other at managing complexity and dealing with problems of scale. Since ACTI is merely a model of how software can be put
together, it provides the necessary descriptive ability to discuss how software should be put together. A potentially fruitful avenue of research is to look at some of the issues discussed in Section 6.2, and determining which architectural practices are "good" and which are "bad," from a software engineering perspective.

**Exceptions.** For simplicity, ACTI does not attempt to represent exceptions or exception propagation behavior in any way. Many languages, including PL/I, Ada, CLU, Eiffel, and Standard ML, include some form of exception mechanism that programmers may use for handling erroneous conditions that arise at run-time. Exceptions are simply a control flow mechanism, however, not a software structuring mechanism. As a result, it seemed reasonable to leave them outside the scope of this research project. Incorporating exceptions into ACTI would be provide an interesting future research topic, however.

**Concurrency.** Again for simplicity, ACTI was developed assuming a single thread of control. Of course, if one cannot do the single-threaded case correctly, it is pointless to consider the more complex concurrent case, so this was a reasonable first step. Note, however, that since ACTI only maps out the mathematical spaces of meaning for program objects, it does not embody any explicit model of computation. Thus, the problem of how to effectively model the environment of a concurrent program within ACTI, or an extension of it, is a particularly exciting research direction. The question of whether this would entail any additional software modularization mechanisms is also open.
Polymorphism in OOPLs. Because ACTI judiciously separates its derivation mechanisms from its representation of type-to-type and module-to-module relationships, it has a noticeably different flavor than the programming model of a traditional OOPL. Looking at the effects of using mappings, rather than inheritance, to determine polymorphic behavior in OOPLs is another possible research topic.

Performance Specifications. While specifying the functional behavior of software operations is fairly well understood, specifying their time and space performance in a modular fashion is an area of open research. Promising work on incorporating support for modular verification of performance specifications within the RESOLVE framework has already begun to appear in the literature [34, 35, 58]. It would be interesting to examine whether ACTI can effectively capture the specification structures needed for this task, or how it would need to be extended to do so.

Conjoining Specifications. The need for performance specifications opens up a more general problem in software specification. That is, it is very desirable to permit semi-independent “facets” of a specification to be described independently. The theoretical foundations necessary to do this are uncertain, however. For example, usually one considers specification of the time performance of operations to be “independent” of specification of the functional behavior of those same operations. Unfortunately, both specifications are dependent on the abstract model of the data being manipulated by the operations. Typically, performance specifications require that more detail about the way data is actually represented should be visible, so that details
of performance variations can be adequately described. It is possible that through the use of interpretation mappings, distinct "facets" of a specification could be written using differing abstract models of data, and later conjoined to form a composite specification for a subsystem. Exploration in this direction could make the job of specification much easier.
APPENDIX A

Previous Models

Because the idea of formalizing the concept of software or of software development has been around for so long, there are many research efforts relevant to a new formal modeling attempt. The research approach followed in this dissertation is to start with five modern "models" chosen to be representative of previous research: the 3C model of reusable software, OBJ (as realized in OBJ3), RESOLVE, Eiffel, and Standard ML. These five models will form the foundation for developing a new model.

This appendix will describe these five representatives of prior work, highlighting the issues each brings to the problem of modeling software structure. This information will be used in Appendix B to form a description of the requirements for a "universal" model of software and how it is constructed. Appendix C will show how each of the five prior models presented here can be reinterpreted within the mathematical spaces defined in ACTI.

Unfortunately, most past work on "models" of software has centered on a particular language. As a result, only one of the five models considered in this appendix is presented as a language-independent conceptual model—the 3C model. The remaining four are actually specific languages. The abstract "model" of software components
that is under consideration in each of these four cases is the one implicit in each corresponding language. These four languages in particular were chosen for breadth of coverage: two of the languages are functional programming languages; one is object-oriented; one is object-based; two support formal specification; one more supports informal specification; one supports verification; three support parameterized programming; and they all aim to strongly support data abstraction and encapsulation.

They also all aim to support software that is modularly constructed, and claim to be applicable to the architectural problems of large-scale software systems.

The subsections in this appendix describe each of these five models in turn. While a complete presentation of each model is beyond the scope of this dissertation, this appendix aims to present enough detail for one to grasp the unique issues that each model raises with respect to reusable subsystems. The description of each model is divided into a basic introduction, a more detailed overview of the model or language’s key features, and a presentation of the advantages and disadvantages of that model with respect to software subsystem construction, reuse, and understanding.

A.1 The 3C Model

One of the more recent attempts to characterize reusable software components is the “3C model.” Unlike the other models discussed in this appendix, the 3C model of reusable software components was conceived as an abstract, language-independent framework for discussing the features of software parts. It was originally developed at the “Reuse in Practice” workshop, held July 11–13, 1989 in Pittsburgh, Pennsyl-
vania [62]. The 3C model is an enhancement of the Concept/Context model initially proposed by Will Tracz in his dissertation work at Stanford [63]. This model is thus based on previous work embodied in LIL [15, 19] and OBJ [21].

A.1.1 An Overview of the 3Cs

The 3Cs stand for:

**Concept**—The abstraction captured in a component. This embodies *what* the component does. Tracz describes this term as follows:

The “concept” behind a reusable software component is an abstract canonical description of “what” a component does. Concepts are identified through requirement or domain analysis as providing desired functionality for some aspect of a system. A concept is realized by an interface specification and an (optionally formal) description of the semantics (as a minimum, the pre- and post-conditions) associated with each operation. An Ada package specification with its behavioral semantics described in Anna is an example of a reusable software concept. [62, p. 11]

**Content**—The implementation of that abstraction. This embodies *how* the component does what it does.

The “content” of a reusable software component is an implementation of a concept, or “how” a component does “what” it is supposed to do. The basic premise is that each reusable software component can have several implementations that obey the semantics of its concept. The collection of (28) stack packages found in Grady Booch’s components is an example of a family of implementations for the same concept (a stack). [62, p. 11]
**Context**—The environment in which the component is designed to operate. This embodies the constraints that must be satisfied in the environment in which the component will be reused.

The “context” of a reusable software component is 1) the environment that the concept is defined in ("conceptual context"), and 2) the environment it is implemented under ("contentual context"). It is very important to distinguish between these two types of contexts because different language mechanisms (inheritance and genericity) apply differently to each. Furthermore, these two contexts clearly distinguish between type inheritance and code inheritance. [62, pp. 11-12]

The focus of this model is on separating the concept, content, and context of a component in order to enhance reusability.

Most programmers are already familiar with the notion of a component’s “concept”; it is usually identified with some form of module interface specification. It is important to distinguish the notion of a concept from how it is represented in a particular programming language, however [5, p. 5]. The concept, by definition, encompasses a complete description of the component’s behavior. An interface specification in a given language, on the other hand, may only capture some aspects of this description (e.g., the parameter profile) in a formal way. This is certainly useful, since it provides some measure of automatic checking, but it should not be mistaken for the richer abstraction of the component’s concept if the language does not permit a complete description of the component’s abstract behavior.

In a similar vein, software engineers usually identify the content of a component with an encapsulated module implementation. Naturally, there can be many such implementations that all provide the same externally observable functional behavior.
Presumably, if the implementation language allowed, each such implementation could be checked to ensure conformance with the corresponding concept description for the component (or, as much of the concept as can be expressed in that language).

The remaining “C” in the 3C model is the context—the external environment that is relevant to the definition and operation of a given component. Most often, one might view the modules upon which another component’s implementation is built to be context. Similarly, modules that define abstract types or other information needed to describe a given component’s specification are also context. In an OO language, when one uses an existing module as the basis for the definition of a new module through inheritance, the first serves as part of the context for the second.

By making a clear distinction between external dependencies and new concepts or implementations, one can more effectively separate changeable aspects of a component from constant aspects. One can even go a step further by decoupling direct external dependencies through the use of some form of parameterization. In most procedural languages, the obvious choice for providing fixed external dependencies is through a static importation mechanism (for example, Ada’s with statement, Modula-2’s import, or C’s #include). Alternatively, the client can be given the capability of binding in actual external dependencies through generic parameters. Potential clients of the resulting component can then configure it for use in a variety of environments (contexts). In some sense, the “parameters” that represent this context form an abstract description of the dependencies a component has on its environment.
Further, one can consider trying to separate external dependencies from either a component’s concept or from its implementation. With this in mind, the term \textit{conceptual context} will be used to refer to the external dependencies of a component’s concept. The term \textit{implementation context} will be used in a similar fashion to refer to the context of a particular implementation. Regardless of where a given external dependency is used, the component writer may decide to hard-code a specific reference—making this part of the context \textit{fixed}—or decouple it through some mechanism (e.g., parameterization) controlled by the eventual reuser—making this part of the context \textit{variable}.

The benefits of using this framework to view the problem of creating reusable software are summed up as follows:

Defining reusable components in a programming language can thus be viewed as the task of separating \textit{context} from \textit{concept}, \textit{concept} from \textit{content}, and \textit{content} from \textit{context}. While this does not answer the question of how to design reusable components in general, it does provide a new perspective on the question of how to represent such components. Designing the component involves forming an abstraction and identifying the \textit{concept} and the \textit{context}, then separating them to achieve the best change control and reusability. Once this is done, one can concretely represent this abstraction in a given programming language. [emphasis in original] [5, p. 9]

A more detailed description of the 3C model is provided by Edwards [5], who uses this model to develop a set of programming guidelines for creating reusable Ada software. This model evolved through “bottom up” efforts to characterize current knowledge about how module-level reuse is best performed today. Additional work
with the 3C model has also been aimed at modeling larger grained reusable parts [38].

A.1.2 Issues Raised by the 3Cs

The 3Cs themselves form a nice conceptual framework for discussing many aspects of reusable software. Perhaps the greatest contribution of this model is the way it frames the notions of concept, content, and context as three distinct facets of a component that have equal importance. Effectively separating these parts within a given component is critical for reuse. Unfortunately, these notions have blurry definitions, in part because of the goal of language independence. The following subsections reflect the issues the 3C model raises in the context of this research by describing the advantages and disadvantages this model has with respect to reuse.

Advantages

The 3C model has the following advantages:

1. It is language-independent.

2. It recognizes the notion of an abstract description of component behavior (i.e., a concept), and advocates separating this abstraction from both the component’s implementation, and the component’s external dependencies.

3. It recognizes the notion of a component implementation (i.e., content), and advocates separating it from both the component’s abstract description of behavior and the component’s external dependencies.
4. It recognizes the notion of a component’s external dependencies, and advocates separating them from both the component’s abstract description of behavior and the component’s implementation. It also supports separating external dependencies required by a component’s abstract description from the external dependencies required by its implementation.

5. It allows for the formal specification of the behavior provided by a module (the concept).

6. It permits one to consider generalizing abstract descriptions of component behavior, although it does not specifically address such generalizations.

7. It advocates the use of multiple implementations for a given abstraction—i.e., multiple contents for one concept.

8. While it advocates separating context from concept and content, it does not specify (or limit) what language mechanisms should be used for this task.

9. While advocating generic parameter mechanisms for component abstractions and implementations, it does not prescribe a particular parameter conformance enforcement policy.

10. It does not require that concepts be simple modules—concepts could be used to encapsulate large subsystems of interconnected modules bundled in a single package, or even a component generator, where the parameters to the generator are viewed as context provided via a particularly rich formalism.
Disadvantages

Unfortunately, the 3C model also has the following limitations:

1. It is not formally defined, and there are occasional disputes over definitions for the most basic terms (e.g., “context”).

2. Because it is not well defined, it is very difficult to communicate to others.

3. Also because it is not well defined, it fails to address the needs of one who wishes to reason about the behavior of a (possibly complex) composition of components.

4. Because of its bottom-up origins, it is often criticized as being insufficient for scaling up to components of “realistic” sizes.

5. “The 3C model is a qualitative model, similar to the Greek ‘earth, air, fire, water’ model of physics.”—Don Batory, WISR’91 [37].

6. Because it does not specifically address the notion of general-purpose abstract descriptions of behavior (e.g., common interface models), it also fails to address the need for correspondence mechanisms between a given component and such a general description.

7. While it advocates many-to-one correspondences between implementations and concepts, it does not address the question of whether this relationship should be many-to-many.
8. It does not explicitly address the need for structuring the space of contextual parameters required by a concept or implementation.

9. It does not explicitly address the need for mechanisms for defining new modules as extensions or enhancements of existing ones.

A.2 OBJ

OBJ was originally designed by Joseph Goguen in 1976 [18] as a tool for extending algebraic abstract data types. Its design and early evolution were heavily influenced by Goguen’s work with Clear [3], and OBJ took on the form of a specification language that would support the testing of specifications by making them executable [17, 16]. Thus, an OBJ specification of an abstraction can be considered as a set of equations or axioms that define the semantics of the exported operations, or it can be considered as a program that can be executed by interpreting the defining equations as rewrite rules.

OBJ was also designed to support programming at a higher level of abstraction than traditional high order languages by focusing on the interconnection and composition of programming modules, particularly reusable modules. It thus borrowed from Clear many ideas for programming with parameterized modules. OBJ3, the most recent version of OBJ [21], reflects Goguen’s thoughts on the subject of parameterized programming.

Goguen describes the purpose of parameterized programming as follows:

It may happen that there is a software part we want to reuse, but it is not in exactly the right form, or perhaps we have combined some
modules and now we want to improve the efficiency of the combination. . . . Parameterized programming allows us to achieve such goals by modifying parameterized modules, either before or after combination or instantiation, so that they can fit a wider variety of applications. [21, p. 160]

He further characterizes the key types of modifications that can be made as the following [21, p. 160-161]:

- Extend a module by adding to its functionality.
- Rename some of its external interface.
- Restrict a module, by eliminating some of its functionality.
- Encapsulate some existing code.
- Combine (add) two or more modules.
- Modify the code inside a module.

OBJ is designed to allow all of these forms of modification while guaranteeing that some selected set of module properties are preserved by the modification.

A.2.1 An Overview of OBJ

OBJ is a functional language. All operations are functions, and their behavior is specified by sets of axioms or equations. Because OBJ specifications can be interpreted using a rewrite approach, it is very desirable that the set of axioms provided by a specification have the Church-Rosser property (i.e., independent of the order in which rewrites are applied) and be terminating. Unfortunately, the combination of a functional style with an algebraic specification approach can be bad for reuse:
In fact, because of the functional style of an algebraic specification, it is easier to define behavior in that style that cannot be implemented efficiently than to define behavior that can be. [67, p. 24]

Further, algebraic specifications do not effectively support the needs of clients or designers in reasoning about components. This is because they describe behavior, without providing a specific conceptual model of the values being manipulated. Chapter II shows why such a model is critical for supporting (human) reasoning.

OBJ’s theoretical roots lie in order-sorted algebra (OSA), which overcomes some of the expressive difficulties with many-sorted algebra approaches [21]. OSA is a rigorous mathematical theory, and it is used as the basis for an abstract denotational semantics for OBJ. The language also has a more concrete operational semantics based on order-sorted rewriting, as one might expect.

There are four high-level kinds of entities in OBJ: modules, views, module expressions, and reductions. Further, modules can be either objects or theories. These classes of entities will each be described in turn.

The name OBJ is derived from the keyword it uses to introduce a module of executable operations, called an object.

In OBJ, an object is a module, which may export types, operations, and axioms. An object is not the same as the notion of a “data object,” which is how the term is used in most other languages. Instead, an OBJ object is a description of an “abstract software component” that is available to serve as a building block when constructing larger software subsystems. Objects can thus be viewed as executable code.
An OBJ theory, on the other hand, is an abstract description of behavior or of an interface. A theory has the same structure as an object, but the difference is that objects are executable while theories simply define properties that other modules may have. The methods by which modules are manipulated, combined, and modified are applicable to both objects and theories.

Note that a particular theory is not a priori bound to any particular object. Instead, theories implicitly define families of related modules (objects and other theories) that have the specified properties. This is different than the specification for a single object, which defines the interface of a particular module—the types, operations, and behavior exported by that module.

Most often, a theory is used to describe the semantic requirements placed on a generic parameter to another module. Such a theory would describe the semantic characteristics required of any "actual" supplied for the corresponding "formal" parameter in an instantiation. Any module that could be shown to comply with this theory—i.e., any module that meets the requirements of the parameter—could be passed in as the corresponding parameter value during an instantiation.

An OBJ view is a correspondence between an OBJ module (an object or theory) and an OBJ theory. For example, a view can be used to express the mapping between an object and a theory, showing how that object meets the requirements of that theory. Also, a view can be used to show how one theory meets the requirements of another and how everything that corresponds to the first also meets the requirements of the second. Any such view can be given a name and reused in many situations.
In OBJ, objects, theories, and views can all be generic. This allows the context of the corresponding entity to be explicitly identified: "For code to be reusable, it and anything that it relies upon (its context) must be known" [21, p. 184]. It also allows for dependencies on context to be decoupled from modules, and thus lies at the heart of parameterized programming.

An OBJ module expression is a statement which actually describes the composition of two or more modules. Module expressions are the means by which a programmer can actually specify the instantiation or aggregation of modules, and the renaming, addition, elimination, or replacement of entities exported by a module. Theories can be composed to form more sophisticated composite theories, and collections of objects and theories can be composed to form composite objects. The evaluation process includes ensuring that any formally stated program properties, specified through theories and views, are maintained by the product of the composition.

Finally, a reduction is a statement that evaluates a given OBJ expression in the context of a particular object. The expression is evaluated by interpreting the axioms within the object(s) as rewrite rules. Thus, a reduction is analogous to a top-level program execution, where the body of the program is simply an expression—as would be expected in a functional language.

A.2.2 Issues Raised by OBJ

Goguen's work with OBJ has provided the theoretical foundations behind many of the component level reuse research efforts in the last decade. It provides programming language concepts that support reuse and which have a mathematically rigorous basis.
The advantages and disadvantages of OBJ with respect to reuse are described in the following subsections.

**Advantages**

As expected, OBJ has a huge number of advantages for supporting reuse and the manipulation of subsystems, mostly stemming from the use of parameterized programming. Among these advantages are:

1. It allows for the formal specification of the behavior provided by a module (the concept).

2. Theories support the notion of generalizing properties shared by many modules.

3. It supports the notion of a formally verifiable correspondence between a given module and a given theory—called a view.

4. It supports many-to-many mappings between modules and the theories that they satisfy.

5. It provides a mechanism for separating context from concept (e.g., generic parameterization).

6. It provides an intensional parameter conformance mechanism for use with generic parameters [7]—any module that can be mapped to the requirement theory defining a formal parameter can be substituted as the corresponding actual.

7. It provides a mechanism for structuring the space of contextual parameters required by a concept (e.g., theories).
8. It provides a mechanism to define new modules as extensions or enhancements of existing ones (importation/inheritance).

Disadvantages

Unfortunately, OBJ also has the following limitations:

1. It appears to be poorly understood, except in theoretical circles.

2. It does not clearly separate specification from implementation, since an OBJ object defines both the specification and implementation simultaneously.

3. OBJ views allow behavior to be added to a given module in order for it to conform to a theory. While this added behavior simply takes the form of additional axioms in OBJ, it would also necessitate implementation code in a more conventional language. This code would need to be encapsulated in some structure.

4. It lacks a distinct notion of "context"—it indirectly addresses this need instead by defining some language mechanisms that can be used to provide context (e.g., generic parameterization and importation).

5. It is based on algebraic rather than model-based specification techniques. Weide et al. discuss why this is not the best choice for reuse [67, p. 10-20, 24].

6. It fails to provide the support necessary for dealing with module-level state and objects that have their own local state [21, p. 180]. This is a natural outgrowth of the fact that OBJ is a functional language. In LIL [15, 19], Goguen remedies this situation to some extent.
7. OBJ does not allow nested modules in its current version, although Goguen recognizes the need for them and envisions them as part of OBJ in the future [21, p. 199].

8. Goguen hints at the need for "dependent types"—i.e., nested generic modules—but implies that there is not yet a developed formal semantics for such objects [21, pp. 209–210].

9. OBJ3 includes a built-in module called TUPLE that is an n-ary parameterized module. This is one example of the need for "variable-length" module parameterization, a need that has arisen in other languages as well. Basically, one would like to have a "vector" of modules, all matching the same requirement theory, be a parameter to another module. Unfortunately, OBJ3 only provides this capability for this one built-in module, and there is no generalized way for users to parameterize their own modules in this fashion.

A.3 RESOLVE

RESOLVE [67, 33, 27, 24, 57, 31], the REusable SOfware Language with Verifiability and Efficiency, is the product of ongoing research efforts by the Reusable Software Research Group (RSRG) at The Ohio State University. This language consolidates many of the features and concepts that support reuse from existing programming languages into a single uniform framework. As noted by Weide et al., RESOLVE supports many of the basic notions from the 3C Model [67, p. 4–8, 60–61]. The
language constructs, along with their associated proof rules [33], implicitly describe a formal model of reusable software.

A.3.1 An Overview of RESOLVE

The structure of RESOLVE components nicely fits into the framework of the 3C model:

- The abstract functional behavior of a piece of software is explicitly separated from the implementation of that behavior; i.e., concept is separated from content.

- For a particular abstract behavior there may be multiple implementations that differ in time/space performance or in price, but not in functionality; i.e., a given concept may have more than one content that realizes it.

- The external factors that contribute to the explanation of behavior are separated from that explanation; i.e., the context of concept is separated from the concept itself.

- The external factors that contribute to the implementation of behavior are separated from implementation code; i.e., the context of content is separated from the content itself.

[67, p. 7]

In addition, RESOLVE adds some contributions that are not described in previous work.

The reasoning behind many of the contributions of RESOLVE is highlighted by considering the following question: “What could be worse than not reusing software?”

1. “An inappropriate component might be chosen—one whose actual behavior is misunderstood by the client programmer” [67, p. 10].
2. "An apparently reusable abstract component may be designed poorly from the standpoint of reuse." In particular, it might be prohibitively inefficient when applied by some clients [67, p. 21].

3. "A not-quite-suitable component might be available—one with no hope of adaptation to the specific needs of the client" [67, p. 45].

4. "An incorrect concrete component might be chosen—one that is not a correct realization of the corresponding abstraction" [67, p. 53].

RESOLVE addresses each of these ways that reuse can go awry.

To address the first type of mistake, choosing an incorrect component, RESOLVE was designed with clarity and abstractness of specifications in mind. In this case, clarity means that a component specification must be "clear, unambiguous, and understandable to a potential client and to a potential implementer" [67, p. 11]. Abstractness means that the specification "must be free of implementation details in order to support a variety of concrete [implementations]." To encourage specifications with these properties, RESOLVE has a sublanguage for writing formal specifications using a model-based rather than an algebraic approach. The idea that a single abstract specification may have many alternative concrete implementations is central to RESOLVE's notion of a module.

The second type of mistake involves poorly designed components, which may be excessively inefficient in some circumstances.

An abstract component with only inefficient concrete components to implement it tempts a client to "roll his/her own." A poorly designed ab-
abstract component may even inherently rule out efficient realizations. [67, p. 21]

Further, Weide et al. [67, p. 21] claim there is no theoretical reason to believe that there is an intrinsic tradeoff between generality and efficiency. RESOLVE has many features aimed at eliminating design choices that restrict efficient component implementations. This is one reason why RESOLVE uses model-based specifications rather than algebraic techniques—the functional style of algebraic specifications make it easier to define specifications that cannot be implemented efficiently [67, p. 24]. Further, RESOLVE advocates the use of swapping rather than copying as the basic mechanism for moving data objects around in programs [25].

To address the third type of mistake, inflexible components, RESOLVE provides language mechanisms to parameterize context through genericity. The language recognizes the distinction between fixed and variable (e.g., parameterized) context, and the distinction between the context of the concept and the context of the implementation (content). Thus, RESOLVE allows components to have separate generic parameters for specifications and for implementations. This generic mechanism allows instances of concepts to be parameters to other concepts or implementations. For maximum reusability, the RESOLVE discipline encourages fully parameterized components that have no fixed context [51, p. 25]. Finally, RESOLVE supports a limited form of type or specification inheritance originally called "enhancement," and later [9] termed "re-exporting."

To address the fourth type of mistake, incorrect components, RESOLVE requires that it be possible to certify that any component implementation correctly supplies
the functionality described in the corresponding specification. The proof rules for RESOLVE [33] ensure that the verification process factors well along component boundaries, so that components can be independently verified. This modularizes the verification of large software systems, and contradicts earlier beliefs that "there is no reason to believe that a big verification can be the sum of many small verifications" [4].

The net effect of this framework is that RESOLVE requires complete information hiding and encapsulation, rather than the leaky encapsulation supported or even encouraged by other conventional programming languages, such as Ada. This works to the programmer’s benefit, whether she be a client or an implementer of a component, by eliminating many of the problems caused by "chasing efficiency at the expense of correctness" [67, pp. 43–45].

A.3.2 Issues Raised by RESOLVE

RESOLVE stands out among the languages discussed in this appendix in that it was specifically designed to support software reuse above all other concerns. Further, its goal of modular verifiability has brought forward many additional insights. It meshes extremely well with the framework of the 3C model, and arguably provides a model of software construction that is scalable to very large systems. The following subsections describe its advantages and disadvantages in more detail.
Advantages

RESOLVE has the following advantages for constructing reusable software:

1. The abstract functional behavior of a piece of software is explicitly separated from the implementation of that behavior.

2. A particular abstract description of behavior may have multiple implementations.

3. The external factors that contribute to the explanation of behavior are separated from that explanation.

4. The external factors that contribute to the implementation of behavior are separated from implementation code.

5. RESOLVE requires descriptions of behavior to be formally specified.

6. Functional specifications are provided using a model-based rather than an algebraic approach. This is important for both efficiency and human factors reasons, and naturally supports RESOLVE's emphasis on data objects that have their own local state.

7. In RESOLVE, all modules are generic, and parameterization mechanisms are provided for expressing context dependencies.

8. The language recognizes the distinction between fixed and variable (e.g., parameterized) context, and the distinction between the context of the concept and the context of the implementation (content).
9. RESOLVE's generic mechanism allows instances of concepts to be parameters to other concepts or implementations, allowing for the structuring of the space of contextual parameters.

10. A limited form of inheritance is provided to allow new modules to be defined as enhancements or extensions of existing modules.

11. The correspondence between a component implementation and a concept is clearly defined and formally verifiable.

12. The verification rules for the language ensure that the verification process can be truly modular, where component implementations can be verified once independently of any given contextual environment.

13. The language implicitly enforces complete information hiding and encapsulation, discouraging the weaker encapsulation that leads to "leaky" abstractions in more conventional languages.

Disadvantages

The RESOLVE work does have the following limitations, however:

1. It appears to be poorly understood outside of the RSRG.

2. It is difficult to communicate to others.

3. Because of its bottom-up origins, the techniques used in RESOLVE are often criticized as being insufficient for scaling up to components of "realistic" sizes.
This is a particularly prevalent (albeit invalid) criticism of the verification tech-
niques.

4. It lacks a way to describe general properties shared by many module concepts.

5. The definition of conformance for generic parameters is restrictive compared
to that provided in OBJ—module parameters are provided via an extensional
mechanism while more primitive parameters are provided intensionally. This
can be a critical issue when it comes to composability requirements [6].

6. The component model is implicitly defined as part of the language and verifica-
tion rules rather than being explicitly separated. This means that it is difficult
to extract a language-independent model with the same formal basis. Further,
ectangling the language features and the component model in this way makes it
difficult to use the model for many of the purposes to which it might be applied.

7. The use of inheritance for defining new modules in terms of existing ones is con-
servative. This is related to the requirement for independent formal verifiability
of the resulting modules, but a less conservative approach to inheritance could
still permit modularity.

8. RESOLVE does not allow nested modules, generic or otherwise.

9. "Variable-length" contextual parameters are not supported.
A.4 Eiffel

Eiffel is one of the more recent object-oriented (OO) languages to be used on industrial development projects. OO approaches in general, and Eiffel in particular, bring important additional insights to the software reuse problem that are not highlighted by the other approaches described here.

As with most OO languages, Eiffel is aimed at achieving quality software by focusing on the extendibility, flexibility, and adaptability of software components. Bertrand Meyer observes that over 40% of maintenance costs result from changes in user requirements, remarking that:

The magnitude of this proportion seems to reflect the lack of extendibility of commonly implemented software: systems are much harder to change than they should be. [46, p. 8]

His proposed approach to software construction focuses on achieving the key qualities of correctness, robustness, extendibility, reusability, and compatibility. These qualities are intended both to improve the overall quality of software and to reduce the cost of maintaining it.

While the OO paradigm embodied in Eiffel is not formally defined, the language itself can be taken as an expression of an implicit model of software construction.

A.4.1 An Overview of Eiffel

Bertrand Meyer [46, p. 12] defines five criteria for evaluating a software construction method with respect to modularity:
• Decomposability—a problem can be decomposed into several subproblems that can be solved independently.

• Composability—the method encourages producing software elements that may be freely combined with each other, possibly in very different environments, to produce new systems.

• Understandability—the method helps in producing software elements that can be separately understood by people.

• Continuity—a small change in the problem to be solved results in a change to one, or just a few modules in the system.

• Protection—error conditions that arise at run-time in one module are contained within that module without being propagated throughout the system.

These five criteria reflect the properties of modular code that Meyer believes contribute to achieving the key qualities of correctness, extendibility, reusability, and so on. Further, those key qualities also contribute to more maintainable code.

Another principle that Meyer considers to be crucial for good software is the “open-closed” principle. Meyer states that modules should be both open and closed [46, p. 23-25]:

**Open**—“A module is said to be open if it is still available for extension” [46, p. 23].

This is directly aimed at planning for the unplanned—preparing for the eventuality of changing requirements.
**Closed**—A module is closed if it is ready for use by other modules. Meyer equates this with having a stable, well-defined interface. For program modules, a compiled implementation available for use by clients is also necessary.

The open-closed principle ensures that software objects are always ready for use, but also always ready for adaptation, enhancement, or change.

Meyer [46] describes the rationale behind Eiffel, and how it and the OO techniques he proposes measure up as a software construction method against these criteria. Much of the emphasis on extendibility and flexibility led Meyer to design Eiffel as a class-based OO language that relies on inheritance as a primary mechanism for relating modules (classes).

In Eiffel, classes are the central programming concept. A class is a combination of a module and a type: it defines both a type and the operations available on that type. Further, a class can inherit from other classes, allowing modules to be defined by difference. In this case, the superclass is termed the “ancestor” class, while the subclass is termed the “heir.”

Each Eiffel class defines certain “features”: attributes or operations that are defined for that class. Only features that are explicitly exported are available for use by clients, although all features are available to any heir classes. Eiffel further allows features to be constrained by “assertions,” which can be used to record the pre- and postconditions of operations, class (module) invariants, and loop invariants. To encourage correctness, class-level assertions apply to all heirs of the given class (and thus all redefinitions of class features).
Because Meyer realized the necessity of protecting software components from variability of the types that are operated on, he included a form of genericity in Eiffel. Eiffel classes can be generic, with one or more formal class parameters. Meyer refers to this as simple, "unconstrained" genericity, and describes ways that inheritance might be used to simulate other forms of genericity. There is no provision for any direct form of procedures as generic parameters.

Note that in Eiffel, there are two kinds of types: "simple" types (one of the built-in scalar types INTEGER, BOOLEAN, CHARACTER, or REAL), and class types. While variables of "simple" types are defined in the language with the usual semantics, variables of non-simple types are defined to be references to objects to support explicit sharing. Thus, there is a deliberate mixture of reference and value semantics inherent in the language.

Meyer also highlights two other features of Eiffel as being critical for an effective OO language: polymorphism and dynamic binding. Polymorphism allows a given program entity—e.g., a variable—to refer to instances belonging to different classes at run-time. Dynamic binding is an implementation technique whereby the run-time system automatically selects the version of an operation (that may have multiple definitions or implementations) necessary for operating on the current object. Meyer argues for these features in order to make software more extendible and representation-independent. By making software components immune to implementation changes, even at run-time, client software can use multiple implementations of
a given data structure without needing to know which implementation will be used for any particular object of that data type.

A.4.2 Issues Raised by Eiffel

The following subsections provide an overview of the advantages and disadvantages of this language, which capture the issues that it raises with respect to reuse.

Advantages

Eiffel has the following advantages:

1. It provides for polymorphic variables.

2. It provides inheritance for defining new concepts in terms of old, defining new implementations in terms of old, enforcing subtype relationships between data abstractions, defining polymorphic relationships, and more.

3. It explicitly supports unplanned, adaptive reuse through inheritance.

4. It provides for semi-formal specification of the behavior of abstractions.

5. It provides strong encapsulation of the implementation details of an abstraction.

Disadvantages

Eiffel also has several limitations:

1. In Eiffel, the notion of a user-defined type is tightly tied to the notion of a module.
2. Polymorphism in Eiffel is tied to both the typing system and to the (single) inheritance mechanism.

3. The many conflicting uses of inheritance [8] are co-mingled in a single language mechanism.

4. Variables of user-defined types are explicitly defined to be references to objects. Explicit aliasing is not only allowed, it is encouraged. Further, built-in assignment and equality operations have reference semantics. The language also defines **Clone** and **Equal** features for every object, which do not have reference semantics, although they only implement shallow copying or equality testing\(^1\).

5. Eiffel supports automatic initialization of objects using programmer-defined features from the corresponding class. It does not, however, support any finalization of objects.

6. Eiffel provides automatic garbage collection for unreachable objects as the primary means of storage reclamation.

7. Eiffel's specification-like assertion slots were not designed with formal specification or verification in mind, leading to many limitations. Eiffel's assertions are boolean expressions that must be evaluable at run-time. There is no provision for universal or existential quantification in assertions. Assertion expressions must be written in terms of an object's state (i.e., attributes of the

\(^1\)Newer versions of Eiffel that post date [46] may have "deep" versions of the **Clone** operation.
implementation), so there is no provision for abstraction of specifications if one wishes to use these assertions as a specification tool.

8. The specification and implementation of a class are defined together, making it difficult to effectively separate them. It is possible to separate them by placing the specification in a deferred class from which the implementation inherits, but this is simply a programmer convention.

9. Although classes can be generic, only other classes are allowed to be generic parameters.

10. Class-level invariants are only enforced after object creation and after any call to a routine of that class [46, p. 479]. Since routines of a class can call other routines of the same class, there is no guarantee that invariants will hold before a given routine is called.

11. Eiffel does not distinguish the notion of context, and provides little support for separating it from either concept or implementation.

A.5 Standard ML

ML was introduced in 1977 to be used as the Meta Language for use with a theorem prover called Edinburgh LCF (Logic for Computable Functions) [54, p. 9], which was more akin to a programmable proof checker. The inference rules and proof methods used by Edinburgh LCF were represented as functions, and ML was intended as a vehicle to allow users to express their own proof strategies in the same functional
style. In addition to the full power of higher order functions, ML was given an innovative module system for structuring code. ML was also given imperative features for pragmatic reasons, although its main focus is on declarative programming. Now, ML has a life of its own as a general purpose programming language.

Although several dialects of ML have evolved, the ML community has banded together to promote a common dialect now known as Standard ML. The formal definition [48] includes the denotational semantics of the language, which constitutes a rigorous description of a model of software component construction. Within this dissertation, the name "ML" will always refer to Standard ML.

A.5.1 An Overview of Standard ML

The presentation of ML is often split into the presentation of the "core" language used to define types and values, and the module system [54, 48, 47]. Note that because ML is a functional language, its notion of "value" includes functions, so that defining values in the core language subsumes the notion of defining functions, and defining types subsumes the notion of defining classes of functions. In this section, however, it is the module system that provides the majority of ML's contributions to the reuse problem: "ML's modules may be the most advanced of any language" [54, p. 1], or at least of any non-academic, general purpose programming language.

The terminology for ML's module system is derived from abstract algebra [54, pp. 275–276]. In abstract algebra, one frequently packages a set of objects together with its operations to form a more abstract object—an algebraic structure. Groups, rings, fields, and so on are common examples of such structures. Further, one can
simply describe the names of the required operations and the laws that they must satisfy, forming an algebraic signature. For example, the typical definition of what it means to be a group (a non-empty set along with a binary operation on its elements, where the set is closed under the operation, the operation is associative, the operation has an identity element in the set, and so on) defines an algebraic signature that all groups must meet. Finally, one can consider functors that map algebraic structures to other algebraic structures.

Proceeding from this analogy, an ML module is either a structure or a functor. A structure is simply a collection of declarations, usually of items that are related. These declarations may include types, values, and other structures. This means that structures can be grouped into larger structures, forming complex but self-contained subsystems that can be manipulated as a single unit.

Similarly, an ML functor is a mapping from structures to structures. The body of the functor defines a structure parametrically in a manner similar to the generics of OBJ or RESOLVE. In the abstract, applying such a functor means substituting an actual structure into the functor's body in place of the corresponding formal parameter. Laurence Paulson [54, p. 1] describes functors as an "extension" of the usual notion of generic modules as it exists in languages like Ada, since the parameters to a functor may be arbitrarily complex structures, rather than simple types and operations.

In addition to modules, ML also offers signatures, which specify type-checking information about a set of exported declarations. A signature lists the types, the
name and type of each value, and the signature of each substructure that is exported. A given structure can then be an instance of this signature if it defines all of the exported entities described in the signature by using conforming declarations. An instance of a signature can declare items that are not specified in the signature, and just as different values can have the same type, different structures can have the same signature. Thus, signatures intensionally define entire classes of structures that share a given syntactic interface.

Note that structures are only a means of grouping declarations, and do not provide any means of hiding information. Viewing a structure through a specific signature provides a way to hide extra details. When a structure is declared, it may be given an explicit signature to hide some of its details, or a signature constraint may be applied later to restrict access to those details. In fact, different signatures could be applied to the same structure at different points to provide views of that structure at different levels of abstraction. A structure declaration need not explicitly mention a signature, however. In this case, ML will infer a signature for the structure that contains the maximum amount of type-checking information and that hides no details.

Structure, functor, and signature declarations in ML are all made in the context of some "environment" of currently active definitions. Simplistically, this environment is the set of declarations that have been made prior to the current declaration. Hence, all three entities can make direct references to external definitions in that environment. To aid in modularity, the original designer of ML's module system proposed the "Signature Closure Rule": signatures can only refer to structures specified within
themselves, but not to free-standing external structures [54, p. 247]. Every signature that obeys this rule is completely self-contained, carrying around inside itself the definitions on which it depends. While this rule has been relaxed in most implementations to allow more freedom for programmers, it is enforced by some compilers on separately compiled modules to simplify the compilation process.

While functors seem to be an elegant implementation of generics, they are interesting because of the novel ways in which they are used by ML programmers. Because a functor takes a structure as an explicit parameter, it is common for all of a functor's external dependencies to come in through that parameter, promoting a very modular form of programming. This also allows modules to be developed and compiled independently. Once the functors are constructed, they can be linked together simply by applying them. Because parameters to functors are specified through signatures, there are no direct dependencies between program units, and it is easy to plug together alternative configurations. This all-functors style of programming [54, p. 270] has very powerful benefits for reuse.

ML also provides sharing constraints, which allow functors to require that certain subparts of their parameters are identical—for example, requiring that a list abstraction and a tree abstraction both operate over the same type. This is very critical for complex contexts, where a functor imports many subsystems through its parameter mechanism. If the functor expects the same key building block to be used by several of the subsystems in its context, it must be able to express this restriction.
Finally, ML also supports a form of polymorphism based on type schemes [54, p. 55]. A type scheme is a template or pattern that describes a relationship among one or more parameters that are also types. For example, the identity function in ML has the type scheme $\alpha \to \alpha$, where $\alpha$ can be considered as a variable representing some ML type. This identity function could be applied to an object of any type, and from the type scheme of the function, we know the range of the function will always be the same type as its domain. This form of polymorphism is interesting because it is amenable to static type checking and prevents run-time type errors, while at the same time allowing a great deal of the flexibility programmers have in polymorphic languages that do not support static type checking.

A.5.2 Issues Raised by Standard ML

Like OBJ, ML provides an elegant and powerful module abstraction for packaging functional definitions. ML also provides support for a novel form of polymorphism that is distinct from the usual OO interpretation. Further, the denotational emphasis of the Standard ML definition ensures that all of these features have a well-documented mathematical meaning which can serve as the basis for language-independent developments. Taken together, ML provides many insights for constructing reusable software, and the issues it brings to this research are reflected in the advantages and disadvantages presented in the following subsections.
Advantages

Standard ML provides the following advantages for constructing reusable software components:

1. It supports the notion of an abstract description of a component’s interface through signatures.

2. Because the formal semantics of ML define a subsumption relationship over signatures, signatures can also be used to represent more generalized descriptions of component interfaces.

3. It supports the notion of a component implementation through structures.

4. It allows multiple structures to match the same signature, thereby permitting multiple implementations for the same concept.

5. It supports many-to-many mappings between structures and the signatures that they satisfy.

6. It allows contextual dependencies to be separated from a component through generic parameters.

7. The Signature Closure Rule advocates complete separation of contextual dependencies from modules.

8. The all-functors programming style supported by ML requires the complete separation of contextual dependencies from all modules.
9. Because generic parameters to an ML functor can be structures, a mechanism is provided to structure the contextual space.

10. ML provides an intensional parameter conformance mechanism for use with generic parameters [7]—any structure that can be mapped to the signature defining a formal parameter can be substituted as the corresponding actual.

11. It provides direct support for nested modules, although it does not allow nested functors.

Disadvantages

Standard ML also has the following disadvantages:

1. Signatures only capture the syntactic aspects of a component's interface.

2. The mechanism by which structures correspond to signatures is based only on syntactic information, and ignores the behavior of the types, operations, and substructures involved.

3. Because of the way signature-to-structure conformance is defined, it is not possible to separate the generic parameters representing conceptual context from those that represent context of the implementation.

4. ML allows structures to serve as a description of their own (implicit) signature, so that the two notions of concept and content can be co-mingled in the same entity.
5. It lacks a distinct notion of “context”—it indirectly addresses this need instead by defining some language mechanisms that can be used to provide context (e.g., generic parameterization and importation).

6. It does not provide any mechanisms for defining new modules as extensions or enhancements of existing ones (inheritance).
Evaluating the Comprehensiveness of a Software Model

Given the collection of previous models described in Appendix A, the next step is to devise an operational means of determining how “comprehensive” each is—the extent to which it addresses the intuitive notions about software construction and reuse that practitioners have observed. This necessarily involves capturing these “intuitive notions” in a more rigorous form. This is done here by using a “structured check list” of properties that a universally applicable model of software should have. Given such a check list, one can then assess the comprehensiveness of any previous, or future, model. This appendix describes the method used to construct the structured check list used in this dissertation, and the check list so produced. The formal definition of ACTI, described in Chapter IV, was derived using this check list as a statement of requirements.

B.1 Method for Deriving the Structured Check List

In order to ensure that the constructed check list captures as much current knowledge as possible, two distinct approaches are used to generate the individual properties:
Table 2: Matrix of Viewpoints

<table>
<thead>
<tr>
<th>Group</th>
<th>Role</th>
<th>Perspective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development</td>
<td>Client/User</td>
<td>One Subsystem in Isolation (from inside)</td>
</tr>
<tr>
<td></td>
<td>Spec. Designer</td>
<td>One Subsystem in Environment (from outside)</td>
</tr>
<tr>
<td></td>
<td>Implementer</td>
<td>Multiple Interrelated Subsystems (from outside)</td>
</tr>
<tr>
<td></td>
<td>V &amp; V/Tester</td>
<td></td>
</tr>
<tr>
<td>Asset Mgmt.</td>
<td>Librarian</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maintainer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>QA/QC</td>
<td></td>
</tr>
</tbody>
</table>

coverage of previous models, and viewpoint analysis. Each property is cross-checked
to ensure that it is considered from both angles.

To ensure coverage of all five previous models, the strengths and weaknesses of
each of the five are used to generate check list properties. These strengths and
weaknesses were presented in Appendix A. In a complementary fashion, viewpoint
analysis [59, p. 68–70] begins by identifying the various roles that software engineers
may play while building, using, and managing software components. These roles are
then used to generate check list properties from each of these distinct perspectives.
By iterating between these two techniques, and ensuring that each generated property
is approached using both, a more comprehensive list can be generated.

Table 2 presents a two-dimensional matrix defining the viewpoints used for the
viewpoint analysis. Along the vertical dimension it shows the various development
and asset management roles individuals may play with respect to reusable software.
Along the horizontal dimension, it shows three distinct perspectives from which
reusable subsystems may be viewed. The intersection of each row and column thus
designates a particular viewpoint: an individual in a specific role looking at a reusable
subsystem from a given perspective.

As is common in computer science [59, p. 68], this viewpoint analysis is carried out
as a mental exercise by the author, rather than by actually interviewing professionals
who play these roles, as is more typical in qualitative research [1, p. 32]. Thus, the
reader should be aware that the resulting check list may be affected by the author’s
personal experience and biases, as opposed to the experience and biases of those who
might have been interviewed.

B.2 The Structured Check List

This section describes the structured check list derived using the approach presented
in Section B.1. The check list consists of an organized list of “yes or no” properties,
or check list items. These properties are stated in the positive: “A model should
\ldots,” rather than the negative. Ideally (or perhaps naively, in some cases), for a given
model it should be clear whether it possesses a given property.

Each check list item consists of a short description, a longer justification for the
item, a description of how concerns for this item are raised at the relevant viewpoints,
and a discussion of how the previous models support inclusion of this item. The items
are grouped in four categories:

1. Properties about abstractions.

\footnote{For more information about viewpoint analysis from a qualitative research perspective, refer to
[61], which gives an in-depth presentation of standpoint theory.}
2. Properties about separation of concerns.

3. Properties about mappings between abstractions.

4. Properties about deriving new abstractions from old ones.

B.2.1 Abstractions

Property Category 1 (Abstractions)—This category of check list properties includes properties that deal with the abstract entities present in a model.

Property 1.1 (Type Abstractions)—The model should provide a method for capturing encapsulated data abstractions, or types. Typically, such a type is considered to be the name of a mathematical set or domain of values.

Justification

Types are more than just a safety feature of programming languages, they are the cornerstone of supporting data abstraction [59, p. 324]. Without being able to identify types, it is extremely difficult to effectively support abstractions of data structures and information hiding with respect to their representations.

Viewpoints

Type support is beneficial from every viewpoint in Table 2.

Support From Previous Models

All of the languages discussed in Appendix A provide the ability to describe type abstractions.
Property 1.2 (Types May Have Invariant Properties)—The model should provide a method for expressing invariant properties that all values of a given type possess.

Justification

It is possible for the values of a type to contain redundant information, and the interrelationship of such redundancies should be spelled out [40, pp. 73]. Effectively, such a "type invariant" statement should be viewed conceptually as a further restriction over the set of values which make up the type [46, p. 125].

Viewpoints

All of the development roles, the maintenance role, and the QA/QC role, regardless of perspective, observe benefits from recording conceptual information about properties that "allowable" values of types must have.

Support From Previous Models

Both Eiffel and RESOLVE offer explicit support for recording type invariants, although Eiffel's mechanism requires these invariants to be computable assertions. Unlike RESOLVE, however, Eiffel lacks a facility for recording the abstract model of a class, so invariants cannot be expressed in abstract terms. In OBJ Axioms in theories and objects do allow such invariants to be expressed.

Property 1.3 (Types Have Abstract Models)—The model should provide a method for describing an abstract, conceptual model of the values that make up each type. This entails providing true abstract types in the sense described by Harms [24]—types
where there is an explicitly described abstract model that is independent of the actual representation of the type.

**Justification**

People’s views of a given software part which they are asked to learn depend critically on the conceptualizations they use [49, p. 241]. Further, it is clear that when faced with a new device or target system, people will formulate a hypothesized mental model, which will naturally evolve with their interaction experience [49] [50, p. 17]. Unfortunately, these “user-formed” mental models tend to be incomplete, unstable, unscientific, and even “superstitious” [49, p. 241].

As a result, it is in the best interests of software clients, designers, implementers, testers, and anyone else who will read the specification for a particular software part to *explicitly* provide a documented, easy-to-understand conceptual model of the part. If this model can be easily assimilated by the reader of the specification, then he or she will be provided with a ready-made conceptualization—one which is documented in a way that can be referred to later, helping to promote consistency, stability, and correctness in the reader’s view of the software part.

Similarly, the reader should be provided with a useful conceptualization of the domain of values that characterize a given type. Typically, this does not mean implicitly defining the domain of values making up a type as “all values that support the following operations . . .,” which is typical in algebraic specifications. Instead, model-based specifications where the reader is provided with an
explicit, rather than implicit, characterization of the structure of values that make up a type is more psychologically suitable.

Viewpoints

All roles that involve reading type specifications benefit from this property, which necessarily includes all of the viewpoints under consideration here.

Support From Previous Models

Of the models presented in Appendix A, only RESOLVE addresses this concern. RESOLVE not only supports, but requires, abstract models for all types. This is a direct result of the model-based specification approach used in the language.

Property 1.4 (State Abstractions)—The model should provide a method for capturing encapsulated state abstractions.

Justification

State abstraction in the form of variables is a staple of procedural programming languages. While it is possible to write programs which do not refer to any explicit state—indeed, both OBJ and Standard ML support this—any model of programming that strives to be comprehensive must be capable of modeling subsystems that encompass state.

Viewpoints

Abstraction of state values is most useful to specification designers, who can use it to more naturally and concisely describe some behaviors [59, p. 158], and
to component implementers, who can use state within a component to provide
a concrete realization of higher-level abstract behavior.

Support From Previous Models
RESOLVE, Eiffel, and Standard ML all provide support for state abstraction
within a module or class. OBJ, which is based on a functional programming
style, does not.

Property 1.5 (Operation Abstractions) — The model should provide a method for
capturing encapsulated operation abstractions.

Justification
Operation abstractions in the form of procedures or functions are one of the
defining elements of imperative programming. Any model of software that
strives to be comprehensive must be capable of modeling subsystems that pro-
vide abstract operations.

Viewpoints
Because of their important role in imperative programming, operation abstrac-
tions are used by all of the development roles described in Table 2, regardless
of perspective.

Support From Previous Models
All of the languages presented in Appendix A provide support for operation
abstraction.
Property 1.6 (Operations Entail Behavior)—An abstraction of an operation necessarily entails the behavior demonstrated by the operation, not just the domain(s) and range(s) of the operation. The model should provide a method for capturing a description of the behavior of each operation abstraction.

Justification

The behavioral description of an operation is the analog of an abstract description of a type. For psychological reasons, it is important to provide the reader of a specification an explicit, rather than implied, description of what each operation actually does.

Further, specifications, including the behavioral specifications of operations, are essential for achieving program modularity [40, p. 157]. Without them, it is impossible to predict, in advance, the effect of a series of operation invocations without knowing how the operations are actually implemented.

Viewpoints

As with abstract descriptions of types, all of the roles and perspectives shown in Table 2 observe benefits from recording conceptual information about the behavior of operations.

Support From Previous Models

Of the languages presented in Appendix A, only RESOLVE provides for explicit representation of the behavior exhibited by a program operation. OBJ allows for the behavior of operations to be defined implicitly by a set of axioms. Eiffel
provides for executable pre- and postcondition assertions which can be used to explicitly represent the complete behavior of many operations. Standard ML does not provide any method for capturing behavioral descriptions of operations.

**Property 1.7 (Subsystems: Collections of Definitions)**—*The model should provide a method for collecting and encapsulating a group of related definitions, including type, state, and operation definitions.*

**Justification**

If there is no way to group together related definitions, it is impossible to provide meaningful encapsulation or information hiding, which are critical tools for managing complexity in software systems. Without being able to distinguish some form of “boundary” between representations or implementations that are defined and the other software artifacts they are hidden or encapsulated from, such protection measures cannot be used.

**Viewpoints**

The complexity control benefits of encapsulation and information hiding extend to all of the viewpoints in Table 2.

**Support From Previous Models**

All of the languages presented in Appendix A provide some notion of a “collection” of definitions: RESOLVE provides concepts and realizations; OBJ provides objects and theories; Standard ML provides structures and signatures; and Eif-
fel provides classes. The 3C model also includes the notion of "concept," which is essentially the idea of a software subsystem.

**Property 1.8 (Subsystems May Have Invariant Properties) — The model should provide a method for expressing invariant properties of a subsystem as a whole.**

**Justification**

Just as a single object (a value of a type) may contain redundant information, subsystems that include state information can contain redundant state information. The interrelationships of the various state abstractions within a single subsystem should be clearly spelled out. Such a "subsystem-level invariant" statement should be viewed conceptually as a further restriction over the set of legal composite states which the subsystem may take on.

**Viewpoints**

Just as with type invariants, all of the development roles, the maintenance role, and the QA/QC role, regardless of perspective, observe benefits from recording conceptual information about properties that subsystems exhibit.

**Support From Previous Models**

Both Eiffel and RESOLVE offer explicit support for recording subsystem-level (or module-level, or class-level) invariants. In addition, axioms in OBJ theories and objects allow such invariants to be expressed.

**Property 1.9 (Nested Subsystems) — The model should allow subsystems to contain other subsystems.**
Justification
Hierarchical structuring of collections of program objects is one of the time-tested tools of managing complexity in software. A comprehensive model of software structure must certainly be capable of modeling software parts with hierarchical substructuring.

Viewpoints
Because of the role hierarchical structuring plays in the presentation and understanding of specifications of software parts, all of the roles and perspectives described in Table 2 directly benefit from its proper use.

Support From Previous Models
Standard ML provides full support for nested structures (modules) and signatures (interfaces). Nested modules are also part of OBJ on paper, but are not yet implemented in the latest version of the language [21, p. 199].

Property 1.10 (Subsystem Abstractions)—The model should provide a mechanism for expressing structural and/or behavioral properties shared by one or more subsystems. Such a description might be viewed as a “Subsystem interface description” that is independent of any single component.

Justification
A description of such properties forms an abstraction of a given subsystem, one that might capture commonalities among a whole group of components [7].
Thus, this description implicitly defines a family of related subsystems. Such a description may be used for expressing requirements on import interfaces.

**Viewpoints**

Abstractions over families of related subsystems are useful for all of the roles under consideration, across all perspectives. For clients, such abstractions make it possible for one to understand a single specification, and then reuse that understanding across multiple subsystems. For specification designers, such abstractions make it possible to organize and describe the conceptual similarities between subsystem specifications. For implementers, such abstractions make it possible to create an implementation that depends only on an abstract definition of an entire family of possible supporting modules, rather than building one-off implementations hard-coded to depend on a single underlying module. For testers and QA/QC personnel, such abstractions make it possible to effectively develop test suites applicable to entire families of components. For component library managers, such abstractions indicate possible ways to group and organize collections of subsystems. For maintenance personnel, such abstractions describe families of effectively interchangeable components, allowing plug-compatible changes that fix bugs, improve performance, or enhance functionality to be more easily identified.

**Support From Previous Models**

In OBJ, theory modules are used for this purpose, implicitly defining an entire family of related theories and objects that share the behavioral traits specified
therein [21, p. 186]. In Standard ML, signatures are used to implicitly define families of structures, although only syntactic commonalities can be described. In RESOLVE, concepts effectively define the syntactic and behavioral commonalities shared by a family of related realizations, but the scope of this mechanism is more limited than in either OBJ or Standard ML. In Eiffel, superclasses are used to describe the common syntactic and behavioral traits shared by a collection of subclasses. Finally, in the 3C model, concepts encompass the role of abstractions over families of subsystems.

Property 1.11 (Parameterized Subsystems)—The model should provide a mechanism to express parameterized subsystems, i.e., subsystem templates with formal parameters, which can be used to generate actual subsystems through some form of parameter substitution mechanism.

Justification

Parameterized subsystems are necessary to effectively model software architectures based on parameterized programming (see Section A.2). Intuitively, “A parameterized abstraction is really a class of related abstractions defined by a single specification” [40, p. 47].

Viewpoints

Although it initially appears that parameterized subsystems are most useful to specification designers and component clients, the technique actually provides benefit for all of the viewpoints listed in Table 2. Clients can clearly
use them to employ parameterized programming techniques to structure their software, while specification designers can design a single specification covering an entire class of closely related components. Specification designers and component implementers alike can use parameterized programming techniques to make specifications and implementations more independent of lower-level units [5, p. 39][31, pp. 45-46][51, p. 5]. For languages that allow parameterized modules to be verified to be correct independent of any parameter values (such as RESOLVE, but not Standard ML or Eiffel), testers, QA/QC personnel, and maintainers can all look at a parameterized subsystem in isolation for testing, debugging, certification, and repair activities. Of course any effort placed into these activities will naturally apply any time the parameterized subsystem is instantiated. Finally, parameterized subsystems stand as an exemplar in place of a class of subsystems for library management purposes, and if parameterization is used to eliminate hard-coded context dependencies as advocated in [5, 31, 51], managing dependency relationships in the library can be considerably simpler.

**Support From Previous Models**

OBJ was the first language to advocate parameterized programming, and to date supports it the most wholeheartedly. RESOLVE and Standard ML also provide good support for parameterized programming. Eiffel provides parameterized classes [46, p. 105-110], although Meyer considers parameterized units to be of much more limited use [45]. Because the 3C model grew in part out of work based on OBJ, it also advocates parameterized modules. Further, the 3C model,
RESOLVE, and Standard ML all advocate the idea of completely parameterizing all external context dependencies to make software parts more flexible.

**Property 1.12 (Explicit Context Interfaces)**—*The model should provide a mechanism for explicitly expressing a subsystem's dependency on any definitions or entities external to the subsystem.*

**Justification**

For code to be reusable, it and anything that it relies upon (its context) must be known. [21, p. 184]

In order to deal with groups of interacting subsystems, it is necessary to have a clear picture of their interrelationships. Without an explicit indication of what external resources subsystems depend on, and which they might interact through, it is extremely difficult to get an effective handle on the behavior of the subsystems, in isolation, in combination, and in varying environments.

Note that an explicit contextual interface can be much more refined than simply "module A depends on module B." Ideally, one would like to know exactly what entities (within B) A depends on, and perhaps even the specific properties of those entities A requires to be true (rather than just "they exist"). Thus, an expressive form of context interface might go much farther than a simple subsystem dependency graph.
Viewpoints

Explicit context is relevant to the roles of client, tester, librarian, maintainer, and QA/QC personnel. It is relevant from the perspectives of a single subsystem in a given environment, or multiple interacting subsystems in a given environment. To the client, explicit context spells out how a given subsystem depends on services provided by its environment, allowing the client to decide whether that subsystem can be integrated into the overall software system he is constructing. Testers, of course, must ensure that a subsystem lives up to its external specification, which includes not only the advertised behavior the subsystem provides, but also the assumptions the subsystem makes about its external environment. Librarians must maintain information about the capabilities of each software asset, and about the interdependencies of such assets. Maintainers are responsible for diagnosing and fixing software problems, many of which arise exactly because a subsystem makes certain assumptions about its external environment that are not valid in the program at hand. QA/QC personnel, much like testers, are concerned with the quality and robustness of a subsystem, which critically hinges on the number and nature of assumptions made about the operating environment by the subsystem. For all of these roles, explicit defining the external dependencies exhibited by a subsystem makes certain assumptions clearly visible, instead of implicit in the code itself.
Support From Previous Models

The 3C model is the primary motivator for explicit contextual interfaces, especially because it advocates separation of contextual dependencies from both content and context. In this vein, RESOLVE not only provides syntactic support for a form of contextual interface, but requires this to be the only means of external referencing for concepts and realizations. In OBJ, parameterized modules have explicit contextual interfaces, but the importation mechanisms only make simple module dependency graph information explicit. In Standard ML, functors (parameterized modules) have explicit contextual interfaces, but all modules are also freely allowed to refer to external definitions that do not come in to the local scope through any form of interface. The Signature Closure Rule (see Section A.5) advises against using such external references, however. Finally, Eiffel only allows one to express simple module-level dependencies.

B.2.2 Separation of Concerns

Property Category 2 (Separation of Concerns)—This category includes check list properties that ensure there is appropriate separation between various kinds of abstractions, between things that change and things that are constant, and between abstractions that might be written with conflicting goals in mind.

Property 2.1 (Specification Versus Implementation)—The model should separate specifications of subsystems from their implementations, both physically and semantically.
Justification

Specifications are tools for defining abstractions, while implementations are tools for describing particular representations of abstractions:

A specification is distinct from any implementation of the abstraction it defines. The implementations are all similar because they implement the same abstraction; they differ because they implement it in different ways. The specification defines their commonality. [40, p. 43]

It is critical that abstractions have precise definitions if the advantages of locality and modularity are to be achieved [40, p. 42]. The tool for providing this definition is the specification. Further, it is critical that the abstraction not be overspecified, in order to preserve flexibility and reusability [40, p. 151]. In particular, it is important to ensure that the specification is not based on a particular implementation, which promotes overspecification [46, p. 53]. Physical, syntactic separation of specifications from implementations is an important technique for promoting this distinction.

Viewpoints

The strict separation of specification and implementation is of primary benefit to clients, specification designers, implementers, testers, and maintainers, from the perspective of a single subsystem. Clients benefit from the separation because eliminating implementation details from subsystem specifications makes the specifications easier to understand, and may allow for a wider variety of realizations with alternative performance profiles to choose from. For
specification designers, the separation encourages and supports the mental distinction between specification details and implementation concerns, promoting the design of more general, flexible, and reusable specifications. For component implementers, the separation helps to ensure that abstractions are not over-specified, allowing more freedom for clever and creative implementations. For testers, the separation ensures that there is a physically isolatable description of the expected externally visible behavior of a subsystem to serve as the final word on test results. For maintainers, the separation encourages specifications that are free of implementation details, promoting the availability of greater numbers of truly plug-compatible implementations.

Support From Previous Models

The 3C model presumes that concepts are distinct from content. RESOLVE also separates concepts (specifications) from realizations (implementations) both physically and semantically. OBJ, on the other hand, does not explicitly include the notion of an "implementation," which is a byproduct of its roots as a language for writing specifications. Semantically, however, one can interpret OBJ objects as implementations and OBJ theories as specifications, which are distinct flavors of modules.

Standard ML does provide separate constructs for defining module interfaces (signatures) and implementations (structures). Declared signatures in Standard ML are separated both physically and semantically from any structure(s) they might describe. A structure can be declared without an explicit signature,
however, in which case it is considered to implicitly define a signature of its own which has no syntactically separate presentation.

Finally, Eiffel distinguishes class specifications from class implementations in its semantics, but there is no syntactically separate presentation of specifications in the language. In fact, Meyer acknowledges that the class construct in Eiffel is, strictly speaking, for embodying abstract data type implementations only, not ADT specifications [46, p. 59].

**Property 2.2 (The Specification/Implementation Relationship)**—Separating specifications from implementations implies that there is some relationship between each implementation and the specification it fulfills. The model should provide a means of expressing this relationship as a distinct entity separate from both the implementation and the specification involved.

**Justification**

Specifications are abstract descriptions of behavior, while alternative implementations may realize this same behavior in different ways. In order to view a given implementation as fulfilling the described behavior of a given specification, one must have a way of interpreting the lower-level constructs and behaviors of the implementation in terms of the higher-level abstraction presented in the specification. Indeed, writing down critical components of this "interpretation," such as type- and module-level abstraction functions or representation invariants [40, pp. 70–74], is necessary for effective verification. But is this interpretation information part of the specification or the implementation?
Associating the implementation-to-specification interpretation with the specification makes that specification implementation-specific, since other alternative implementations will necessarily require other interpretation information. Alternatively, associating it with the implementation makes that one implementation specification-specific, since other specifications describing behavior this implementation also fulfills will necessarily require other interpretation information. The only way to properly allow the freedom for a many-to-many specification-to-implementation relationship [21, p. 186] is to completely separate the relationship itself from both the specification and the implementation involved.

**Viewpoints**

Separating the specification-to-implementation relationship from both of the pieces involved is most beneficial to implementers and to component clients, from the perspective of a single subsystem. It gives implementers the freedom to reuse a single code unit to provide an effective implementation of many distinct specifications. Similarly, it means that clients may have a larger variety of alternative implementations to choose from for a given specification.

**Support From Previous Models**

If one interprets OBJ objects as implementations and OBJ theories as specifications, OBJ fully supports describing the relationship between the two through views. Views can be separately named and reused, and it is possible to define
several distinct views between the same object and theory, if it is possible to interpret the object as fulfilling the theory in alternative ways.

In the Standard ML semantic definition, the relationship between a signature and a structure is a distinct entity—termed a realization [48, p. 33–34]—although realizations are not visible in the syntax of the language. Standard ML realizations are much more limited forms of relationships than OBJ views, however.

Neither RESOLVE nor Eiffel, nor the 3C model include the specification/implementation relationship as a distinct entity separate from the implementation and specification involved. RESOLVE at least has a place to record the critical information about this relationship, however, as part of the implementation module [2, pp. 47–49].

**Property 2.3 (Information Hiding)** — *The model should provide some mechanism for hiding information—that is, ensuring that some information inside a subsystem cannot be accessed from outside that subsystem.*

**Justification**

The use and importance of information hiding was first presented by Parnas [53]. Information hiding strategies are important for managing complexity, because they aim at protecting hidden information from corruption by outside sources, making assumptions about the state of that information by internal operations safer and more viable. Further, if information hiding is enforced with some
form of encapsulation barrier, it then becomes feasible to easily change the representation or organization of hidden information, a necessity for building plug-compatible software components.

**Viewpoints**

All of the viewpoints shown in Table 2 derive benefits from information hiding, from all three perspectives.

**Support From Previous Models**

All of the models described in Appendix A provide support for information hiding.

**Property 2.4 (Context Versus Specification)**—The model should allow context dependencies within a specification to be explicitly separated from that specification.

**Justification**

In order for code to be reusable, it is necessary to know exactly what it depends on [21, p. 184]. Providing support for explicitly separating a declaration of dependency from the actual use of external definitions in the specification allows this information to be spelled out up front. Further, by encouraging specification designers to explicitly define the assumptions about external entities their specifications make, those designers might similarly be encouraged to reduce or eliminate those dependencies for greater reusability where possible.
Viewpoints

This separation is most useful to clients and to specification designers. To the client, explicit context spells out how a given subsystem depends on services provided by its environment, allowing the client to decide whether that subsystem can be integrated into the overall software system she is constructing. To the specification designer, it can force external dependencies to be written out in full, exposing them to greater scrutiny and encouraging their removal where reasonable.

Support From Previous Models

The 3C model explicitly advocates this separation. RESOLVE carries this advocacy into action by requiring that all context dependencies within a specification be explicitly defined as part of the context interface. OBJ, Eiffel, and Standard ML, on the other hand, give no mechanism for providing this separation unless all context dependencies come in as parameters to a parameterized module.

Property 2.5 (Context Versus Implementation) — The model should allow context dependencies within an implementation to be explicitly separated from that implementation.

Justification

This check list property is the dual of the “Context Versus Specification” property, and has exactly the same rationale, applying to component implementers rather than specification designers.
Viewpoints

This separation is most useful to clients, testers, and maintainers. To the client, explicit context spells out how a given subsystem depends on services provided by its environment, allowing the client to decide whether that subsystem can be integrated into the overall software system she is constructing. Similarly, testers who must ensure that a subsystem lives up to its external specification, including the assumptions the subsystem makes about its external environment, will have an easier time if those dependencies are spelled out explicitly. Maintainers can use this same information to advantage when diagnosing and fixing software problems, many of which arise because assumptions about external dependencies are no longer valid due to the evolution of the system under consideration.

Support From Previous Models

As with specifications, the 3C model explicitly advocates this separation and RESOLVE requires all context dependencies within an implementation to be explicitly defined in its context interface. The other languages provide no mechanism for this separation except in the case of parameterized modules, where context dependencies can be phrased as parameters.

Property 2.6 (Specification Information Versus Program Code)—The model should allow definitions and other information used solely for specifying or describing behavior to be separated from definitions and declarations that give rise to executable artifacts (e.g., code).
Justification

The justification here is threefold: to aid specification designers and component implementers psychologically; to aid language designers; and to aid compiler writers.

Specification designers and component implementers alike must be concerned with writing both programmatic constructs and abstract specifications of their behavior. Implementers also share the duty of defining correspondence relationships between their implementations and the specifications they fulfill. Allowing these individuals to clearly separate definitions and entities created expressly for explaining abstractions to other people or analysis tools, from those which are meant to describe actual machine execution instructions, can aid them in maintaining this distinction in their own heads.

Language designers should also be concerned with this division, which is one facet of the phase distinction [26] between the compile-time elaboration phase and the run-time execution phase for a given programming language. "This phase distinction is at the heart of efficient execution for any typed language" [emphasis in original] [47, p. 26]. Keeping a clear distinction between which elements of language semantics are handled by the two distinct phases is important for maintaining a clear phase distinction. The two types of definitions described in this check list property naturally fall on the opposite sides of this phase distinction.

Finally, if the phase distinction is maintained clearly and meticulously in the
definition of a programming language, then it is much easier for compiler writers to implement the language. At the same time, the information needed to build program verifiers or other static analysis tools can easily be placed in the model.

**Viewpoints**

The viewpoints supported are specification designers and component implementers, from the perspective of a single subsystem.

**Support From Previous Models**

RESOLVE provides this separation in both its syntax and semantics. Standard ML respects the phase distinction separating its static semantics from its dynamic semantics, but does not provide any method of representing behavioral descriptions of abstractions.

**Property 2.7 (Derivations Versus “Is-A” Relationships)**—*The model should separate the mechanisms it provides for deriving new entities from existing ones from those it provides for defining behavioral relationships between abstractions.*

**Justification**

Most modern OO programming languages, including Eiffel, provide an inheritance mechanism that is simultaneously used for “programming by difference”—defining new software parts from existing ones—and for describing subtyping relationships. Edwards [8] describes the many different uses to which this single mechanism may be applied, and how conflicting uses can lead to serious problems, even loss of encapsulation protection. The simplest way to elimi-
nate these problems is simply to disentangle the many uses of inheritance by providing separate mechanisms.

**Viewpoints**

Because of the numerous problems that can be caused by using a single inheritance-like mechanism for many conflicting purposes [8], all of the viewpoints described in Table 2 see the benefits of providing this separation.

**Support From Previous Models**

None of the models presented in Appendix A explicitly provide such a separation.

**B.2.3 Mappings and Bindings**

**Property Category 3 (Mappings and Bindings)**—This category includes check list properties that deal with mappings, bindings, or interpretations between abstractions.

**Property 3.1 (Specification-to-Implementation Mappings)**—The model should provide a mechanism for expressing how a given subsystem implementation “corresponds” to a given specification. It should be possible to check whether such a correspondence is valid.

**Justification**

Any implementation of a given specification must state how the implementation satisfies the specification, including how representations of object values and
lower-level state correspond to the abstract models of those values and state [40, p. 70]. Without recording this information, which is often presented in terms of type-level and module level abstraction functions and invariants, it is impossible to reason effectively about the correctness of any claim that the implementation actually satisfies the given specification. Check List Item 2.2 indicates that the model should separate this information from both the specification and implementation involved, and this item indicates that there should be some mechanism for actually representing this information.

**Viewpoints**

The specification-to-implementation relationship is of primary concern to component implementers, testers, and maintainers, when considering a single subsystem. Implementers need to write down their assumptions about how their implementation actually satisfies a given specification in order to ensure that they do not violate any of the assumptions of this correspondence while writing the code. Later, both testers and maintainers need visibility over these assumptions to ensure that the claimed correspondence is actually valid for the implementation, and also ensure that it remains valid under future changes and evolution of the software part.

**Support From Previous Models**

The 3C model presumes that there is a relationship between alternative implementations (content) for a given specification (concept). In OBJ, views serve this purpose. RESOLVE provides syntactic support for expressing specification-
to-implementation relationships, and requires that every such relationship be documented and formally verifiable. Standard ML includes the notion of the relationship between specifications and implementations in its semantic definition, although only in the form of simple identifier maps which do not show up in the syntax of the language at all. Eiffel does not address specifications, and hence does not address the relationship between specifications and implementations.

Property 3.2 (Specification-to-Specification Mappings) — The model should provide a mechanism for expressing how a subsystem specification "corresponds" to an abstraction over many subsystems. It should be possible to check whether such a correspondence is valid.

Justification

If the model provides abstractions over classes of subsystems as advocated by Check List Item 1.10, then it is logical to require that it also support the relationships between a given subsystem and one of these abstractions. This relationship is very similar in information content with the relationship between an implementation and a specification that it satisfies, and might be presented in terms of type-level and module-level abstraction functions and invariants. Without recording how one specification is viewed as satisfying another, it is impossible to effectively reason about the correctness of their purported relationship.
Viewpoints

Specification-to-specification relationships are of primary concern to clients, specification designers, and library personnel, when considering a subsystem in isolation.

Support From Previous Models

The 3C model includes the notion that concepts might describe an abstract view of the behavior provided by other concepts. In OBJ, views between theories serve this purpose. In Standard ML, signatures are partially ordered and the nature of their relationships is described in its semantic definition.

Property 3.3 (Binding Context) — The model should provide a mechanism for binding actual subsystems/values to an explicit context interface.

Justification

If the model provides a means for explicitly defining the context dependencies in a subsystem as advocated in Check List Item 1.12, it should also provide a method for expressing how those external dependencies are fulfilled by sources outside of the subsystem.

Viewpoints

Clients, looking at a single subsystem against the background of its prospective environment, need a mechanism for tying the two together properly.
Support From Previous Models

Of the languages presented in Appendix A, only RESOLVE provides for explicit context interfaces, but RESOLVE implicitly includes the binding of actual contextual values within the context interface (fixed context); there is no distinct mechanism for doing so. All of the languages do provide mechanisms for binding actual values in place of the formal parameters to a parameterized module, though, which is discussed under Check List Item 3.5.

Property 3.4 (Intensional Contextual Binding)—The model should provide an intensional mechanism for binding actual subsystems/values to an explicit context interface.

Justification

As described by Edwards [7], intensional mechanisms describe groups of objects, in this case external entities that meet the assumptions made within the subsystem, by the properties that those objects must possess, rather than by explicitly naming the objects or a collection of them. Because intensional requirements are more general than extensional ones [7], they increase reusability.

Viewpoints

The client role, from the perspective of a subsystem considered with its environment, is the primary beneficiary of this mechanism. As explained by Edwards [7], it allows greater flexibility and freedom in what actual environmental entities may be used to fulfill the context dependencies of the given subsystem.
Support From Previous Models

None of the models presented in Appendix A explicitly provide such a mechanism. It is worth noting that one of OBJ’s descendants, LIL, does provide such a mechanism, however [20].

Property 3.5 (Binding Subsystem Parameters) — The model should provide a mechanism for binding actual subsystems/values to the formal parameters of a parameterized subsystem in order to create an instance.

Justification

Without the ability to provide actual values in place of formal parameters, parameterized subsystems are of little use. As a result, this property is a direct corollary of Check List Item 1.11.

Viewpoints

As in Check List Item 1.11, all of the viewpoints listed in Table 2 benefit from this property.

Support From Previous Models

All of the models discussed in Appendix A include some mechanism for providing the parameter values of a parameterized subsystem.

Property 3.6 (Intensional Subsystem Parameter Binding) — The model should provide an intensional mechanism for binding actual subsystems/values to the formal parameters of a parameterized subsystem in order to create an instance.
Justification

Just as for Check List Item 3.4, intensional parameter requirements are more general than extensional ones, thereby increasing flexibility and reusability.

Viewpoints

As in Check List Item 3.4, the client role, from the perspective of a subsystem considered with its environment, is the primary beneficiary of this mechanism.

Support From Previous Models

Both OBJ and Standard ML provide completely intensional parameter binding mechanisms for generic subsystems. The 3C model was also built presuming this form of parameter binding. RESOLVE provides intensional binding for some kinds of generic parameters, but not for module-valued parameters.

Property 3.7 ("Is-A" Type Relationships)—The model should provide a mechanism for expressing behavioral relationships between type or subsystem abstractions, including "subtype" relationships.

Justification

Behavioral relationships between types or subsystems, including “is-a” relationships, are useful in understanding and applying existing abstractions [8]. An is-a relationship is a strict subtyping relationship—a class B is a subtype of another class A (B is-a A) if any instance of B can be used wherever instances of A are required [36]. Subtyping relationships are a cornerstone of OO design, programming, and analysis techniques.
Further, as indicated in Check List Item 2.7, it would be beneficial for the model to distinguish between mechanisms for deriving new entities from those for defining behavioral relationships. In order to provide this separation, the model clearly needs mechanisms for both purposes.

Viewpoints

Both clients and specification designers can effectively use is-a relations between types or modules. Designers can use them to organize and relate collections of subsystems or collections of types. Clients can then use these relationships to reduce the amount of learning necessary to master the collection(s), and potentially to write more flexible client software.

Support From Previous Models

Eiffel, as with most OOPLs, provides an inheritance mechanism for defining subtype relationships between classes. Alternatively, OBJ provides an importation mechanism that allows for specification inheritance between modules, thus providing a kind of module-level subtyping. RESOLVE also provides for a form of specification inheritance.

Property 3.8 (Types With Many Representations) — The model should allow the description of data abstractions with varying run-time representations, possibly of varying sizes (polymorphic representation).
Justification

Hilfinger [30, pp. 164–168] suggests providing a Universal type "that (abstractly) contains any value that will fit into its representation, and which keeps track of the type of that value, so that it may be used only in the proper contexts." This capability is critical for table-driven programming, and for frameworks or subsystems that use a call-back style to allow for the automatic execution of user-defined actions [5, pp. 53–61]. Many OOPLs, including Eiffel, provide for polymorphism in some form specifically to allow for the call-back style of programming it permits.

Viewpoints

The ability to provide data abstractions with polymorphic representations is most important to component implementers. Unlike clients, who rely on the conceptual relations between types (Check List Item 3.7, implementers need representation-level flexibility in order to use table-driven, event-driven, or call-back style programming techniques in implementing abstractions.

Support From Previous Models

Of the models presented in Appendix A, only Eiffel addresses this concern at present, although support for polymorphic representation has been suggested for RESOLVE.
Property 3.9 (Late Binding)—The model should allow the binding between an operation's name and the actual program code that will be executed to be delayed as late as desired.

Justification

Most often, late binding is realized by dynamic binding in languages that strive to support polymorphism—by delaying binding until run-time, language implementations can automatically select the version of an operation appropriate to the corresponding polymorphic representation of a given object. More generally, delaying the binding of logical control and data abstractions to their representations as late as possible is a successful information hiding strategy underlying OO techniques [59, p. 204].

Viewpoints

As with polymorphic data representations, the component implementer role is the primary beneficiary of this property.

Support From Previous Models

Of the models presented in Appendix A, only Eiffel addresses this concern.

B.2.4 Derivations

Property Category 4 (Derivations)—This category of check list properties covers mechanisms for deriving new abstractions from old ones, or “programming by difference.”
Property 4.1 (Specification Derivation)—The model should provide a mechanism for deriving new specifications from old ones.

Justification

This property is critical for supporting enhancements and specializations of existing specifications. As Meyer states, often the best way to expand on previous developments is to consider new parts to be imitations, refinements, or combinations, of existing ones [46, p. 217]. Such a mechanism would also provide an effective way to capture commonalities within similar specifications in a way that a single point of control can still be maintained.

Viewpoints

Specification designers working on new subsystems see the benefits of this mechanism.

Support From Previous Models

Of the models described in Appendix A, OBJ provides by far the most sophisticated module derivation mechanisms. Speaking of OBJ’s module expressions, Goguen states, “No other programming language that we know has such a feature in the language itself” [21, p. 197]. RESOLVE also provides a form of specification derivation [9, p. 18].

Interestingly, Eiffel, which was specifically designed to support programming by difference, does not provide for defining specifications by difference, since it fails to include the notion of specification in the language syntax.
Property 4.2 (Implementation Derivation) — *The model should provide a mechanism for deriving new implementations from old ones.*

**Justification**

As in Check List Item 4.1, this property is necessary for supporting enhancements, specializations, and refinements of implementations.

**Viewpoints**

Component implementers working on new subsystem implementations see the benefits of this mechanism.

**Support From Previous Models**

Of the models described in Appendix A, only Eiffel specifically addresses derivation of new implementations from existing ones.

Property 4.3 (Specification-to-Implementation Relationship Derivation) — *The model should provide a mechanism for deriving new specification-to-implementation relationships from old ones.*

**Justification**

As with Check List Items 4.1 and 4.2, this property is necessary for supporting enhancements, specialization, and refinement.

**Viewpoints**

Component implementers working on new subsystem implementations see the benefits of this mechanism.
Support From Previous Models

None of the models presented in Appendix A provide such a mechanism.

Property 4.4 (Simultaneous Specification/Implementation Derivation) —

The model should provide a mechanism for deriving new specification/implementation pair, along with the corresponding specification-to-implementation relationship, from an existing specification, implementation, and relationship between them.

Justification

This property is the logical combination of the mechanisms in Check List Items 4.1, 4.2, and 4.3. Most OOPLs provide a single derivation mechanism that is used for all three purposes. Often, this single mechanism is safely applicable, and hence easier to use than the separate fragments. While the separate derivations are necessary for appropriate separation of concerns, it should be possible for the model to capture their combined, simultaneous use when it is legitimate.

Viewpoints

Specification designers and component implementers see the benefits of this mechanism.

Support From Previous Models

None of the models presented in Appendix A provide such a mechanism, although the notion of “representation inheritance” covering this form of derivation has been suggested for RESOLVE [65].
B.3 Quick Summary of the Comprehensiveness of Previous Models

Table 3 presents a very brief summary of how each of the models introduced in Appendix A measures up against the structured check list presented in Section B.2. In this table, check marks ("✓") are used to show which properties are possessed by a given model. Similarly, asterisks ("*"") are used to show which properties are partially possessed by a given model. For more details on the individual entries in the table, consult the "Support From Previous Models" description for the corresponding check list item.
Table 3: Matrix of Check List Items versus Previous Models

<table>
<thead>
<tr>
<th>Check List Item</th>
<th>Previous Models</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3Cs</td>
</tr>
<tr>
<td>1. Abstractions</td>
<td></td>
</tr>
<tr>
<td>1.1. Type Abstractions</td>
<td>✓</td>
</tr>
<tr>
<td>1.2. Types May Have Invariant Properties</td>
<td>✓</td>
</tr>
<tr>
<td>1.3. Types Have Abstract Models</td>
<td>✓</td>
</tr>
<tr>
<td>1.4. State Abstractions</td>
<td>✓</td>
</tr>
<tr>
<td>1.5. Operation Abstractions</td>
<td>✓</td>
</tr>
<tr>
<td>1.6. Operations Entail Behavior</td>
<td>✓</td>
</tr>
<tr>
<td>1.7. Subsystems: Collections of Definitions</td>
<td>✓</td>
</tr>
<tr>
<td>1.8. Subsystems May Have Invariant Properties</td>
<td>✓</td>
</tr>
<tr>
<td>1.9. Nested Subsystems</td>
<td>*</td>
</tr>
<tr>
<td>1.10. Subsystem Abstractions</td>
<td>✓</td>
</tr>
<tr>
<td>1.11. Parameterized Subsystems</td>
<td>✓</td>
</tr>
<tr>
<td>1.12. Explicit Context Interfaces</td>
<td>✓</td>
</tr>
<tr>
<td>2. Separation of Concerns</td>
<td></td>
</tr>
<tr>
<td>2.1. Specification Versus Imple-</td>
<td>✓</td>
</tr>
<tr>
<td>mentation</td>
<td></td>
</tr>
<tr>
<td>2.2. The Specification/Imple-</td>
<td>✓</td>
</tr>
<tr>
<td>mentation Relationship</td>
<td></td>
</tr>
<tr>
<td>2.3. Information Hiding</td>
<td>✓</td>
</tr>
<tr>
<td>2.4. Context Versus Specification</td>
<td>✓</td>
</tr>
</tbody>
</table>
Table 4: Matrix of Check List Items versus Previous Models (continued)

<table>
<thead>
<tr>
<th>Check List Item</th>
<th>Previous Models</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3Cs</td>
</tr>
<tr>
<td>2. Separation of Concerns</td>
<td></td>
</tr>
<tr>
<td>2.5. Context Versus Implementation</td>
<td>✓</td>
</tr>
<tr>
<td>2.6. Specification Information Versus Program Code</td>
<td></td>
</tr>
<tr>
<td>2.7. Derivations Versus “Is-A” Relationships</td>
<td></td>
</tr>
<tr>
<td>3. Mappings and Bindings</td>
<td></td>
</tr>
<tr>
<td>3.1. Specification-to-Implementation Mappings</td>
<td>✓</td>
</tr>
<tr>
<td>3.2. Specification-to-Specification Mappings</td>
<td>✓</td>
</tr>
<tr>
<td>3.3. Binding Context</td>
<td></td>
</tr>
<tr>
<td>3.4. Intensional Contextual Binding</td>
<td></td>
</tr>
<tr>
<td>3.5. Binding Subsystem Parameters</td>
<td>✓</td>
</tr>
<tr>
<td>3.6. Intensional Subsystem Parameter Binding</td>
<td>✓</td>
</tr>
<tr>
<td>3.7. “Is-A” Type Relationships</td>
<td></td>
</tr>
<tr>
<td>3.8. Types With Many Representations</td>
<td></td>
</tr>
<tr>
<td>3.9. Late Binding</td>
<td></td>
</tr>
</tbody>
</table>
Table 5: Matrix of Check List Items versus Previous Models (continued)

<table>
<thead>
<tr>
<th>Check List Item</th>
<th>Previous Models</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3Cs</td>
</tr>
<tr>
<td>4. Derivations</td>
<td></td>
</tr>
<tr>
<td>4.1. Specification</td>
<td></td>
</tr>
<tr>
<td>Derivation</td>
<td>✓</td>
</tr>
<tr>
<td>4.2. Implementation</td>
<td></td>
</tr>
<tr>
<td>Derivation</td>
<td></td>
</tr>
<tr>
<td>4.3. Specification-to-Implementation</td>
<td></td>
</tr>
<tr>
<td>Relationship</td>
<td></td>
</tr>
<tr>
<td>Derivation</td>
<td></td>
</tr>
<tr>
<td>4.4. Simultaneous</td>
<td></td>
</tr>
<tr>
<td>Specification/Implementation</td>
<td></td>
</tr>
<tr>
<td>Derivation</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX C

How Does the ACTI Model Measure Up?

Given the formal description in Chapter IV, the ACTI model will be analyzed using the check list developed in Appendix B to determine how comprehensive it is. Section C.1 presents this analysis. Subsequent sections illustrate how each of the prior models described in Appendix A can be interpreted within ACTI to demonstrate its generality.

C.1 Comparing ACTI with the Check List

Section 3.1 presented three overriding concerns that guided the development of ACTI: defining it in mathematical terms, maintaining language independence, and achieving all of the properties described in the structured check list of Appendix B. While ACTI’s form addresses the first two, the third is addressed by ACTI’s content. This section provides a point by point explanation of how ACTI satisfies all of the properties described in the structured check list of Appendix B. Since this check list serves as an operational definition of the “underlying conceptual view of software architecture embedded in modern module-structured languages” described in the thesis, this section is the heart of defending the claim that ACTI does in fact capture this information.
C.1.1 Abstractions

Type Abstractions

A *Type Environment*, which appears in every abstract or concrete instance, maps type names to mathematical domains representing the allowable values for that type. Because any arbitrary domain can be used as the associated value of a type name in a *Type Environment*, it is possible for arbitrarily abstract or concrete sets of values to be associated with a type.

Typically, one would expect abstract characterizations of the domain associated with a type or data abstraction to appear in an abstract instance representing a subsystem specification. Similarly, a more concrete characterization of the representation of the object-level state present in a data abstraction to be captured within a concrete instance representing a subsystem implementation. The two could then be combined using a suitable interpretation mapping to form an encapsulated unit, as described in Section 4.13.3, combining the abstract description of behavior with their concrete but encapsulated representations.

Note that in some programming languages, type information is completely relegated to the typing system and considered part of the static semantics of the language. Since ACTI must deal meaningfully with the relationships between subsystem specifications and implementations, which necessarily includes dealing with the relationship between an abstract type specification and its low-level representation, the domain information associated with a type must be included in the model.
Types May Have Invariant Properties

In addition to associating a mathematical domain with a type name, each Type Environment also associates a CNSTR with the type. This predicate over the corresponding domain can be used to describe type invariants.

Types Have Abstract Models

As described above, the mathematical domain MD associated with a type name by a Type Environment can be any arbitrary domain. This allows the flexibility for choosing abstract domains for characterizing the values of a type, and such a domain serves as an explicit model of the type.

State Abstractions

The Variable Environment within each concrete instance effectively captures the notion of state. It is an association between names (variable names) and their respective types and values. For an abstract instance, its Variable Environment Signature simply describes the type of its state by associating names with type information only.

Operation Abstractions

Every concrete instance contains an Operation Environment component, which associates operation names with a corresponding Operation Meaning. This Operation Meaning includes an explicit representation of the input-to-output relation computed by the operation, as well as the effects of the operation on the state of the concrete instance.
Operations Entail Behavior

Every Operation Meaning associated with an operation includes an Operation Model, which is made up of an abstract characterization of when the operation is legitimately applicable ($DP$), and a relational description of the effects of the operation if it is applied legitimately ($EP$). As one would expect, the Operation Environment Signature in an abstract instance associates only an Operation Model with each operation name, rather than a complete Operation Meaning.

Subsystems: Collections of Definitions

Concrete instances serve as cohesive bundles of type, state, operation, and subsystem definitions that can be treated as a single composite structure.

Subsystems May Have Invariant Properties

Every abstract and concrete instance includes an Environment Invariant, which is a predicate over the entire subsystem. This predicate can be used to capture subsystem-level invariant properties.

Nested Subsystems

Every concrete instance has four subcomponents for modeling nested subsystems:

1. An Abstract Instance Environment for nested abstract instances.
2. A Concrete Instance Environment for nested concrete instances.
3. An Abstract Template Environment for nested abstract templates.

Abstract instances, of course, may only contain abstractions of concrete instances.

**Subsystem Abstractions**

An *Abstract Instance* is an abstraction over a collection of concrete instances. It is also an abstraction over a collection of other, more detailed abstract instances. In this way, abstract instances are similar to OBJ theories, which are abstractions over both OBJ objects and other OBJ theories.

**Parameterized Subsystems**

Templates fulfill this role. A *Concrete Template* is a function taking a single concrete instance as an argument and producing another concrete instance as its result. Since a concrete instance can contain arbitrarily many types, variables, operations, and nested subsystems, the template effectively has access to an unlimited number of arguments. Similarly, because the resulting concrete instance can contain arbitrarily many types, variables, operations, nested instances, and nested templates, a given template can simultaneously produce an unlimited number of results, including other templates. ACTI templates are higher-order functors, in the Standard ML sense [44].

Similarly, an *Abstract Template* is a function taking a single abstract instance as an argument and producing another abstract instance as its result. Together, abstract and concrete templates provide the heart of ACTI's support for parameterized programming.
Explicit Context Interfaces

The $CTXT$ component of every abstract and concrete instance serves as an explicit declaration of external dependencies.

C.1.2 Separation of Concerns
Specification Versus Implementation

The space of Abstract Instances is separate and distinct from that of Concrete Instances. Great pains have been taken to ensure that appropriate separation here is observed.

The Specification/Implementation Relationship

All relationships between specifications and implementations (or between pairs of specifications) are captured in the space of Interpretation Mappings. A Interpretation Mapping describes the correspondence between two instances independently of both of the instances involved.

Information Hiding

As described in Section 4.13.3, given an appropriate abstract instance and interpretation mapping $M$, a concrete instance can be “cut down” using the $\downarrow_M$ operator. Intuitively, this operator simply removes all information from the concrete instance that is not mapped through the abstract interface via $M$. This provides a useful way to capture information hiding and encapsulation.
**Context Versus Specification**

Every abstract instance has a \textit{CTXT} component which serves as an explicit declaration of external dependencies. All reference to entities that are not defined within the abstract instance must be imported through this context interface, thus effecting a separation between those external dependencies and the subsystem specification itself.

**Context Versus Implementation**

Every concrete instance has a \textit{CTXT} component which serves as an explicit declaration of external dependencies. All reference to entities that are not defined within the concrete instance must be imported through this context interface, thus effecting a separation between those external dependencies and the subsystem implementation itself.

**Specification Information Versus Program Code**

Both abstract and concrete instances contain \textit{Math Environments}, where all definitions made expressly for simplifying the specification can be placed. Here, they will be kept separate from the definitions of programmatic entities in the remainder of the subsystem.

**Derivations Versus “Is-A” Relationships**

In ACTI, is-a relationships are captured by \textit{Interpretation Mappings}, as discussed below in Section C.1.3. Derivations, however, are captured by operators that produce
new subsystems by combining, extending, or restricting existing ones. Thus, these
two classes of relationships are separated in ACTI.

C.1.3 Mappings and Bindings

Specification-to-Implementation Mappings

Specification-to-implementation relationships in ACTI are expressed as elements in
the space of Interpretation Mappings.

Specification-to-Specification Mappings

Specification-to-specification relationships in ACTI are expressed as elements in the
space of Interpretation Mappings.

Binding Context

Bindings between the CTXT component of an abstract or concrete instance and
its immediately containing subsystem (its environment) are defined as Interpretation
Mappings. This, combined with module embedding, provides an expressive way of
explaining how a given environment fulfills the external dependency requirements of
a given subsystem as described in Section 4.13.6.

Intensional Contextual Binding

Every Interpretation Mapping is intensional, since the mapping itself is independent
of both instances involved. Thus, the CTXT component of a given abstract or con­
crete instance is simply a statement about the minimum properties any corresponding
external environment must have, and any environment possessing these properties,
regardless of its name or how it was derived, can provide an effective parent environment for the given instance.

**Binding Subsystem Parameters**

Templates are essentially functions from instances to instances. Providing the actual parameter values corresponding to the formal subsystem parameters is simply mathematical function application.

**Intensional Subsystem Parameter Binding**

Each template has an associated “domain predicate” which is true for all instances to which that template may legitimately be applied. Both intensional and extensional parameter binding mechanisms can be modeled by such a domain predicate.

**“Is-A” Type Relationships**

In ACTI, is-a relationships are captured by *Interpretation Mappings*, which express how one abstract or concrete instance can be “interpreted as” another abstract instance. As part of expressing this relationship, the mapping must also explain how types in the first can be interpreted as types in the second in a behavior-preserving way. Thus, interpretation mappings effectively capture is-a relationships at the both the type and the subsystem levels.

**Types With Many Representations**

Because the mathematical domain of values associated with a type can be any arbitrary domains, a disjoint union of domains could be used as the space of values for a
given type. This domain could be used to model the many alternative representations for a given type.

Further, one could also provide a domain of uniform values as a model of the same type, at a higher level of abstraction. If the necessary type interpretation mapping between the two spaces can be defined, then a interpretation mapping between the two instances can be defined, and the $\downarrow_M$ operator can be used to encapsulate the lower-level representation details.

**Late Binding**

The domains associated with types are flexible enough to encompass operations—the space of *Operation Meanings*, for example, is just another domain which could be used to define the values associated with some type. Types which are modeled by highly structured domains containing one or more operations as part of each object value can be used to turn operation binding into a behavior modeled at run-time. Further, type-level or module-level tags attached to values can also be used for dynamic dispatch purposes.

**C.1.4 Derivations**

**Specification Derivation**

Derivation of one specification in terms of an already existing specification is easily defined as an operator over the space of abstract instances, or even abstract templates. This is described in Section 4.13.4.
Implementation Derivation

Derivation of one implementation in terms of an already existing implementation is easily defined as an operator over the space of concrete instances, or even concrete templates. This is described in Section 4.13.4.

Specification-to-Implementation Relationship Derivation

Derivation of one interpretation mappings in terms of an already existing interpretation mappings is easily defined as an operator over the space of interpretation mappings, just as for abstract and concrete instances.

Simultaneous Specification/Implementation Derivation

Given an abstract instance $AI$, a concrete instance $CI$, and a interpretation mapping $M$ describing their relationship, it is simple to define a composite derivation operator from the more primitive derivations for abstract instances, concrete instances, and interpretation mappings. The typical inheritance mechanism in an OOP language should be interpreted as just such a composite operator.

C.2 Interpreting the 3Cs within ACTI

Given that ACTI possesses all 22 properties in the structured check list, the next step is to see if each of the previous models can be "reinterpreted" within the framework of ACTI's mathematical spaces. This will also give the reader a better intuitive understanding of what can be modeled ACTI, and how.
The 3C model is centered around the notions of concept, content, and context, and appropriately separating them. Since these terms are not formally defined, it is difficult to talk about whether or not they are perfectly captured in ACTI. Nonetheless, these notions all have clear representatives in ACTI:

\[
\begin{align*}
\text{Concept} & = \text{Abstract Instance} \quad \text{(C.1)} \\
\text{Generic Concept} & = \text{Abstract Template} \quad \text{(C.2)} \\
\text{Content} & = \text{Concrete Instance} \quad \text{(C.3)} \\
\text{Generic Content} & = \text{Concrete Template} \quad \text{(C.4)} \\
\text{Context of Concept} & = CTXT \text{ of Abstract Instance} \quad \text{(C.5)} \\
\text{Context of Content} & = CTXT \text{ of Concrete Instance} \quad \text{(C.6)}
\end{align*}
\]

Clearly, ACTI is concerned with strict separation of concept from content, context from concept, and context from content.

**C.3 Interpreting OBJ within ACTI**

Interpreting OBJ within the mathematical spaces of ACTI is more challenging. This is particularly true because OBJ was designed as an executable specification language, and thus has no real concept of an “implementation.” OBJ objects are the natural analog of implementations in that language, though, an analogy supported by the language’s denotational semantics [21, p. 178].

The core OBJ constructs and their ACTI representatives are:

\[
\text{Object} = \text{Concrete Instance} \quad \text{(C.7)}
\]
Parameterized Object = Concrete Template \hfill (C.8)
Theory \hfill = \hfill Abstract Instance \hfill (C.9)
Parameterized Theory = Abstract Template \hfill (C.10)
Axioms = Collection of *Operation Models* \hfill (C.11)
Views = Interpretation Mappings \hfill (C.12)
Module Composition = Derivation Operators \hfill (C.13)

The only case in which these identities are only approximate is that of views. Ostensibly, OBJ views are type-to-type and operation-to-operation name mappings [21, p. 191]. Unfortunately, because OBJ is based on an algebraic semantic model and all operations are functions in the mathematical sense, views may also map a *composition* of operations to an operation. Two descendants of OBJ, LIL and LILEANNA, carry this a step further, allowing arbitrary code (implementation) fragments to be introduced as part of a view. Because implementations should be encapsulated, and introducing code fragments as part of an abstract interpretation map opens up conflicting goals, this feature of views is (purposefully) not captured by ACTI interpretation mappings.

**C.4 Interpreting RESOLVE within ACTI**

RESOLVE is a fairly natural fit for ACTI, since it’s denotational semantics played a prominent role in the initial formulation of the model. As a result, ACTI is more
general than RESOLVE, but still retains structures that can clearly model the basic RESOLVE constructs:

- Concept = Abstract Template  \hspace{1cm} (C.14)
- Realization = Concrete Template  \hspace{1cm} (C.15)
- Realization Parameters = Parameters to Concrete Template  \hspace{1cm} (C.16)
- Type Correspondence = $R$ in Type Interpretation Mapping  \hspace{1cm} (C.17)
- Type Convention = $C$ in Type Interpretation Mapping  \hspace{1cm} (C.18)
- Module Correspondence = $MCORR$ in Interpretation Mapping  \hspace{1cm} (C.19)
- Module Convention = $MCOV$ in Interpretation Mapping  \hspace{1cm} (C.20)
- Math Definitions = Math Environment in an Instance  \hspace{1cm} (C.21)
- State Variables = $MVE$ in a Math Environment  \hspace{1cm} (C.22)

### C.5 Interpreting Eiffel within ACTI

Eiffel, the only OO language among those presented in Appendix A, identifies the ideas of module and type. This gives the language a different flavor, but ACTI models the two concepts differently. However, it is easy to interpret an Eiffel class as an ACTI concrete instance with a single type defined in its Type Environment. Given this interpretation, the other Eiffel constructs can be interpreted as follows:

- Class = Concrete Instance plus Type  \hspace{1cm} (C.23)
- Generic Class = Concrete Template  \hspace{1cm} (C.24)
- Class Attribute = Entry in Variable Environment  \hspace{1cm} (C.25)
Class Feature = Operation Meaning \hspace{1cm} \text{(C.26)}

Feature Assertions = Operation Model \hspace{1cm} \text{(C.27)}

Class Invariant = EI in Concrete Instance \hspace{1cm} \text{(C.28)}

Inheritance = Combined Derivation Operator and Interpretation Mapping \hspace{1cm} \text{(C.29)}

\text{(C.30)}

Eiffel polymorphism can be interpreted in terms of interpretation mappings, which explain how to translate between module and type abstractions with is-a relations. Eiffel's dynamic binding can also be modeled explicitly in ACTI, for example by associating "tag" values with concrete instances and recording these values in each object belonging to the type defining in that concrete instance. The model of computation built on top of these math spaces can then use these tag values in selecting the appropriate Operation Meaning to apply to a given object.

\section{C.6 Interpreting Standard ML within ACTI}

Just like RESOLVE, Standard ML is also a natural fit for ACTI because of the role its denotational semantics played in the initial formulation of the model. The ACTI structures that clearly model the basic Standard ML constructs are:

\begin{align*}
\text{Structures} &= \text{Concrete Instance} \hspace{1cm} \text{(C.31)} \\
\text{Signatures} &= \text{Abstract Instance} \hspace{1cm} \text{(C.32)} \\
\text{Functors} &= \text{Concrete Template} \hspace{1cm} \text{(C.33)} \\
\text{Types} &= \text{Types} \hspace{1cm} \text{(C.34)}
\end{align*}
Function Values = Operation Meanings \hspace{1cm} (C.35)

Values = entries in a Variable Environment \hspace{1cm} (C.36)

Interestingly, Standard ML functions can be modeled within either the Operation Environment or the Variable Environment of a subsystem, since they are just values of a different type in that language. In the analogies above, they have been identified with ACTI operation meanings, however, since operations have attached behavioral descriptions, which values do not.
Bibliography


[65] Bruce Weide. The Ohio State University, personal communication, 1993.