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.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700  800/521-0600
CONCEPTUAL PROGRAM EDITORS: DESIGN AND FORMAL SPECIFICATION

DISSERTATION

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

By

Paolo Bucci, M.S.

* * * * *

The Ohio State University
1997

Dissertation Committee:
Dr. Timothy J. Long, Co-adviser
Dr. Bruce W. Weide, Co-adviser
Dr. Neelam Soundarajan

Approved by

Dr. Timothy J. Long

Co-advisers
Department of Computer and Information Science
ABSTRACT

When a programmer uses a program editor, his/her view of what programs are and how they are built is affected by the editor itself. This is a special case of a more general phenomenon; namely, that all artifacts have an impact on the way users think of them and of the activities they perform with them. Research in human-machine interaction has explored the ramifications of this insight with respect to how it affects the design of tools and their user interface.

In this dissertation we investigate the implications of this work for the design of program editors. We note that existing program editors convey a low-level, often inconsistent and incomplete conceptual model of programs and program construction. This hides the high-level conceptual view of programs, imposes excessive responsibilities on the programmer, and tends to make the programming task harder and more error prone.

We propose a new class of program editors that we call "conceptual editors". These editors are based on a specific, precise, high-level conceptual model of programs and program construction, and the editor and its user interface can be designed and built to convey the model in a consistent fashion. A conceptual editor, by design, supports, promotes, and even enforces an "appropriate" conceptual view of software through this model.
To show the feasibility of this approach to program editors, we formally specify an 
editor for a small, but significant (in terms of key concepts) subset of a programming 
language. By example, this demonstrates a precise description of a conceptual model and 
of a corresponding editor's capabilities. We also discuss how this design could be 
extended and generalized to an editor for a full-fledged programming language.
To my parents
ACKNOWLEDGMENTS

First, I would like to thank my advisers, Tim Long and Bruce Weide. They both have followed my work closely, and their guidance has been invaluable. Their support and confidence in me has always exceeded my expectations. I am very lucky to have been able to work with them during the past four years. I also thank Neelam Soundarajan for many valuable comments and constructive suggestions.

I am grateful to past and current members of the Reusable Software Research Group at The Ohio State University, and especially Bill Ogden, for numerous discussions and helpful insights.

Many friends have shared and shaped my life during the past nine years. Most of them have come and gone while I struggled to find my way out of this seemingly inescapable maze. They are the reason I can now look back on this long experience and believe it was worth the time and suffering. In more or less chronological order, I would like to thank Hal and Eva McMillan, Wayne Heym, Ming-Jye Sheu, Sanjay Gupta, Manas Mandal, David Gibson, Bharat Kumar, Sergey Zhupanov, Lena Grayfer, Zafar Ali, and, last but not least, Bhavana Nagendra.

In spite of the distance that separates us and the lack of effort on my part, some old friends have simply refused to give up on me and our friendship. They are Fabio Besta, Paolo Rollino, and Filippo and Andrea Tampieri. I thank them from the bottom of my heart because they showed with their actions how much they care about me.
For the past five years one thing more than anything has helped me survive the stress and pressure of graduate school and that is my almost daily aerobic workouts with the Buck-I-Robics program at The Ohio State University. I wish to thank the program coordinator, Liz Davis, and all past and current Buck-I-Robics instructors for helping me become fit and healthy both physically and mentally.

Finally, and most importantly, I wish to thank my parents, Giovanni and Adele Bucci, my brother, Mario, and my sister, Valeria, for their unconditional love and unwavering support.
VITA

November 30, 1962 .................. Born – Ferrara, Italy.

1986 ....................... Laurea in Scienze dell’Informazione, Università degli Studi, Milan, Italy.

1986 ....................... Research Staff Member, CISE, Italy.


1988 ....................... Consultant, Unicad, Spa, Italy.

1988 - 1989 ....................... University Fellow, The Ohio State University.

1989 ....................... M.S. Computer and Information Science, The Ohio State University.

1989 - present ....................... Graduate Teaching and Research Associate, The Ohio State University.

PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in:

- Software Engineering
  - Prof. Bruce W. Weide
- Theory of Computation
  - Prof. Timothy J. Long
- Computer Graphics
  - Prof. Richard E. Parent
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDICATION</td>
<td>iv</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>v</td>
</tr>
<tr>
<td>VITA</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xii</td>
</tr>
<tr>
<td>CHAPTERS:</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 The Problem</td>
<td>1</td>
</tr>
<tr>
<td>1.2 The Thesis</td>
<td>2</td>
</tr>
<tr>
<td>1.3 The Proposed Approach</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Formal Specification of a Type Declaration Conceptual Editor</td>
<td>8</td>
</tr>
<tr>
<td>1.5 Contributions</td>
<td>10</td>
</tr>
<tr>
<td>1.6 Organization</td>
<td>11</td>
</tr>
<tr>
<td>2. CONCEPTUAL EDITORS</td>
<td>13</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Conceptual Models of Programming</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1 Component vs. File/Class</td>
<td>17</td>
</tr>
<tr>
<td>2.2.2 Conceptual Relationships vs. Inheritance</td>
<td>20</td>
</tr>
<tr>
<td>2.2.3 Operation Abstract Description vs. Function Prototype</td>
<td>22</td>
</tr>
<tr>
<td>2.2.4 Abstract Parameter Mode vs. Parameter Passing Mechanism</td>
<td>22</td>
</tr>
<tr>
<td>2.2.5 Chain_Position_Template vs. Pointers</td>
<td>23</td>
</tr>
<tr>
<td>2.3 Design of Conceptual Editors</td>
<td>24</td>
</tr>
<tr>
<td>2.3.1 Meaningful Programs</td>
<td>24</td>
</tr>
<tr>
<td>2.3.2 Meaningful Entities</td>
<td>28</td>
</tr>
<tr>
<td>2.3.3 The Context</td>
<td>30</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>4.2</td>
<td>Limitations and Problems with Existing Approaches</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Text Editors</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Syntax-Directed and Structure Editors</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Visual Environments</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Smalltalk</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Editor Generators</td>
</tr>
<tr>
<td>4.3</td>
<td>Intentional Programming</td>
</tr>
<tr>
<td>4.4</td>
<td>Productivity Tools as Editors</td>
</tr>
<tr>
<td>4.5</td>
<td>Chapter Summary</td>
</tr>
<tr>
<td>5.</td>
<td>CONCLUSIONS</td>
</tr>
<tr>
<td>5.1</td>
<td>Summary</td>
</tr>
<tr>
<td>5.2</td>
<td>Conclusions</td>
</tr>
<tr>
<td>5.3</td>
<td>Contributions</td>
</tr>
<tr>
<td>5.4</td>
<td>Future Research</td>
</tr>
<tr>
<td>A.</td>
<td>NESTED LIST</td>
</tr>
<tr>
<td>A.1</td>
<td>TREE_ENHANCED_MACHINERY</td>
</tr>
<tr>
<td>A.2</td>
<td>Nested_List_Template</td>
</tr>
<tr>
<td>A.3</td>
<td>TREE_POSITION_MACHINERY</td>
</tr>
<tr>
<td>A.4</td>
<td>Nested_List_With_Position_Template</td>
</tr>
<tr>
<td>B.</td>
<td>NESTED MAP SET</td>
</tr>
<tr>
<td>B.1</td>
<td>NESTED_MAP_ENHANCED_MACHINERY</td>
</tr>
<tr>
<td>B.2</td>
<td>Nested_Map_Set_Template</td>
</tr>
<tr>
<td>B.3</td>
<td>NESTED_MAP_SETPOSITION_MACHINERY</td>
</tr>
<tr>
<td>B.4</td>
<td>Nested_Map_Set_With_Position_Template</td>
</tr>
<tr>
<td>B.5</td>
<td>NESTED_MAP_PATH_MACHINERY</td>
</tr>
<tr>
<td>B.6</td>
<td>NESTED_MAP_EXTRACT_PATH_MACHINERY</td>
</tr>
<tr>
<td>B.7</td>
<td>Nested_Map_Set_Path_Template</td>
</tr>
<tr>
<td>C.</td>
<td>AN IMPLEMENTATION FOR THE TYPE DECLARATION EDITOR</td>
</tr>
<tr>
<td>C.1</td>
<td>ID_Template</td>
</tr>
<tr>
<td>C.2</td>
<td>Type_Declaration_Editor - Realization Header</td>
</tr>
<tr>
<td>C.3</td>
<td>Type_Declaration_Editor - Realization Body</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>213</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>35</td>
</tr>
<tr>
<td>7</td>
<td>39</td>
</tr>
<tr>
<td>8</td>
<td>52</td>
</tr>
<tr>
<td>9</td>
<td>54</td>
</tr>
<tr>
<td>10</td>
<td>56</td>
</tr>
<tr>
<td>11</td>
<td>57</td>
</tr>
<tr>
<td>12</td>
<td>58</td>
</tr>
<tr>
<td>13</td>
<td>60</td>
</tr>
<tr>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td>15</td>
<td>64</td>
</tr>
<tr>
<td>16</td>
<td>65</td>
</tr>
</tbody>
</table>

xii
<table>
<thead>
<tr>
<th>Page</th>
<th>Section Title</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>The POSITION_MACHINERY Module</td>
<td>103</td>
</tr>
<tr>
<td>40</td>
<td>The POSITION_MACHINERY Module (continued)</td>
<td>104</td>
</tr>
<tr>
<td>41</td>
<td>The POSITION_MACHINERY Module (continued)</td>
<td>105</td>
</tr>
<tr>
<td>42</td>
<td>The TYPECONTEXT_TEMPLATE Module</td>
<td>108</td>
</tr>
<tr>
<td>43</td>
<td>Example of Type Declaration and the Corresponding Type Context</td>
<td>110</td>
</tr>
<tr>
<td>44</td>
<td>How the Type Context is Built</td>
<td>111</td>
</tr>
<tr>
<td>45</td>
<td>The TYPECONTEXT_MACHINERY Module</td>
<td>113</td>
</tr>
<tr>
<td>46</td>
<td>The TYPECONTEXT_MACHINERY Module (continued)</td>
<td>114</td>
</tr>
<tr>
<td>47</td>
<td>The TYPECONTEXT_MACHINERY Module (continued)</td>
<td>115</td>
</tr>
<tr>
<td>48</td>
<td>The TYPECONTEXT_MACHINERY Module (continued)</td>
<td>116</td>
</tr>
<tr>
<td>49</td>
<td>The Type_Declaration_Editor Concept</td>
<td>121</td>
</tr>
<tr>
<td>50</td>
<td>The Type_Declaration_Editor Concept (continued)</td>
<td>122</td>
</tr>
<tr>
<td>51</td>
<td>The Type_Declaration_Editor Concept (continued)</td>
<td>123</td>
</tr>
<tr>
<td>52</td>
<td>The Type_Declaration_Editor Concept (continued)</td>
<td>127</td>
</tr>
<tr>
<td>53</td>
<td>The Type_Declaration_Editor Concept (continued)</td>
<td>128</td>
</tr>
<tr>
<td>54</td>
<td>The Type_Declaration_Editor Concept (continued)</td>
<td>129</td>
</tr>
<tr>
<td>55</td>
<td>The Type_Declaration_Editor Concept (continued)</td>
<td>131</td>
</tr>
<tr>
<td>56</td>
<td>The Type_Declaration_Editor Concept (continued)</td>
<td>132</td>
</tr>
<tr>
<td>57</td>
<td>The Type_Declaration_Editor Concept (continued)</td>
<td>135</td>
</tr>
<tr>
<td>58</td>
<td>The Type_Declaration_Editor Concept (continued)</td>
<td>136</td>
</tr>
<tr>
<td>59</td>
<td>The Type_Declaration_Editor Concept (continued)</td>
<td>137</td>
</tr>
<tr>
<td>60</td>
<td>The Type_Declaration_Editor Concept (continued)</td>
<td>138</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1 The Problem

A program editor is a tool to build and manipulate program descriptions. Many different approaches to the design of program editors have been explored with the goal of providing programmers with as much support as possible for a challenging and demanding task: building programs. Existing approaches range from general purpose text editors, that allow the user unrestricted manipulation of purely textual representations of programs, to syntax-directed and structure editors, that add syntactic and structural support, to the more recent visual environments, that try to go beyond a purely syntactic view of program descriptions.

Through the way it displays programs and through the kinds of operations it allows for manipulating program representations, an editor conveys and promotes a certain view of programs. This view can have a profound impact on the way programmers think and how they build software. Programs are not strings of characters or any other concrete representation; they are conceptual entities built from other conceptual entities to achieve an intended behavior. Each building block has a meaning, a role, and a purpose that go beyond the concrete syntactic representation the language assigns to the construct.

Unfortunately, existing program editors convey a low-level, often inconsistent and incomplete conceptual model of programs and program construction. This hides the high-level conceptual view of programs, imposes excessive responsibilities on the programmer, and tends to make the programming task harder and more error prone. For example, consider a plain text editor. Text editors are general purpose editing tools, i.e., they are not necessarily designed with the programming task in mind. However, they are still among the most popular editing tools used by programmers. Of course, text editors are used to edit strings of characters regardless of what they mean. Operations are provided that allow users to manipulate this representation in any way that makes sense in the realm of text strings. Clearly, this is lacking as a model of programs: it is clear that the editor does not provide any specific support for the programmer's task; it just supplies a blank canvas on which programmers can construct and manipulate any string of characters. Whether those strings have any meaning as programs, and whether operations such as cutting and pasting an arbitrary piece of text produce any meaningful result, are issues that the programmer is forced to cope with because of the editor's inability to deal with them.

1.2 The Thesis

As part of his research in human-machine interaction, Donald Norman has recognized several important issues in the design of artifacts for human use. Here is a brief review of some of his most important conclusions.

The artifacts that we use have an impact on how we think. The mental model the user forms of the artifact is the key to this statement:
"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." [63, p. 241]

So the artifact influences the mental model that the user forms, and through that model has an impact on how the user thinks about the artifact and the activities involved in using it. Specifically, if the artifact is a program editor, the user's model of the editor and of the interaction with it affects the way the user thinks about and builds programs.

**Conceptual models are fundamental to the design of artifacts.** Conceptual models are usually devised as tools to aid people in understanding a system or to help teach the use of a system:

"A conceptual model is invented to provide an appropriate representation of the target system¹, appropriate in the sense of being accurate, consistent, and complete. Conceptual models are invented by teachers, designers, scientists, and engineers." [63, p. 241]

However, Norman points out that conceptual models should also be the foundation of any user interface design effort:

"In the ideal world, when a system is constructed, the design will be based around a conceptual model. This conceptual model should govern the entire human interface with the system, so that the image of that system seen by the user is consistent, cohesive, and intelligible. I call this image

---

¹. The **target system** is the system that the person is learning or using.
the system image to distinguish it from the conceptual model upon which it is based, and the mental model one hopes the user will form of the system.”

[63, p. 244]

So an appropriate conceptual model of the artifact, and of the activities involved in using it, should be the starting point in the design, not an afterthought aimed at explaining how to use the artifact.

Make important entities explicitly visible. Too often important aspects of a system are hidden among other details, forcing the user to make an additional effort to keep track of them, and possibly causing problems when the user ignores them:

“The important things to watch should be visible and clearly marked. …

The principle of visibility is violated over and over again in everyday things. … Many systems are vastly improved by the act of making visible what was invisible before.” [64, p. 100]

The issue of what aspects should be explicitly presented to the user is directly related to the next issue of how they should be displayed.

Representation is important. How the important aspects of an artifact or system are presented to the user is essential to the user’s proper and successful manipulation of the artifact or system:

“A good representation captures the essential elements of an event, deliberately leaving out the rest. … The critical trick is to get the abstractions right, to represent the important aspects and not the unimportant. … Herein lie both the power and the weakness of representations: Get the relevant aspects right, and the representation provides substantive power to enhance people’s ability to reason and think;
get them wrong, and the representation is misleading, causing people to ignore critical aspects of the event or perhaps form misguided conclusions.” [65, p. 49]

A consequence of this observation is that different representations for different aspects of a system may be necessary in a complex system that involves conceptually distinct activities.

**Constraints are important.** Physical or logical constraints imposed on the user by an artifact should not be perceived as restrictive, but as a helpful tool to limit the space of possible alternatives the user needs to deal with:

“How can design signal the appropriate actions? ... One important set of signals comes through the natural constraints of objects, physical constraints that limit what can be done. Another set of signals comes from the affordances of objects, which convey messages about their possible uses, actions, and functions. ... Affordances suggest the range of possibilities, constraints limit the number of alternatives.” [64, p. 82]

Of course, it is the responsibility of the designer not to make the constraints too restrictive, and to make sure that the alternatives prevented by the constraints are undesirable and thus will not be missed.

This dissertation explores the implications of Norman’s research in human-machine interaction for the design of program editors. It defends the following thesis:

(i) Norman’s conclusions suggest that program editors should be designed with different goals from existing program editors.

(ii) Conceptual program editors—editors designed with these new goals in mind—are substantially different from existing approaches.
(iii) It is possible and desirable to formally model the conceptual entities that make up programs and to formally specify conceptual editors for them.

1.3 The Proposed Approach

A fundamental flaw of existing program editors is the lack of a well thought out conceptual model of software. Editor designers apparently choose a program representation (e.g., text or syntax tree) and then the allowed operations are chosen based on this representation. This is opposed to allowed operations being based on a meaning or conceptual model of represented software. We propose a new class of program editors that we call "conceptual editors". These editors are designed with a specific, precise, high-level conceptual model of programs and program construction in mind. Specifically, a precise description of the model is constructed first, and then the editor and its user interface are designed and built so that the model is conveyed in a consistent fashion and so that users (programmers) can rely on it.

The design of the conceptual model requires the selection and description of appropriate conceptual entities. These are not purely syntactic entities: they have a precise meaning and a role and these have to be made explicit in the descriptions of the entities. Of course, there can be many models. The choice of a model depends on the programming paradigm. For instance, if we choose an object-oriented paradigm, then we might describe a program as a collection of classes and objects, where objects are instances of classes, and classes have methods (operations) and attributes (data members, instance variables). On the other hand, if the chosen paradigm is based on abstract data types (ADTs), we might say that a program is a collection of ADTs, where an ADT is a type and some
operations on variables of that type. Even for a given choice, there may be more than one model—or more than one way to look at things—that applies to the same entity, e.g., data flow model vs. control flow model of algorithms.

Although many of the entities in a conceptual model can be mapped directly to syntactic structures in the language (e.g., procedures, control structures, variables, types, etc.), some of them may not have an explicit description in a program. They may be entities that conceptually make sense and that are fundamental to the program construction process, but that are only implicitly described by traditional program representations. For example, the idea of context (i.e., the set of declared entities available at a specific place in the program, together with all the information required to use them correctly) is one that can be very useful if made explicitly part of the model. If the context is only implicit, then the programmer must mentally build it by examining the whole program and applying the language's scope rules. As Norman points out: “Many systems are vastly improved by the act of making visible what was invisible before.” [64, 100].

In addition to the conceptual entities, it is necessary to describe allowed operations on the entities. Operations are not chosen based on the actual displayed representation (which has not been chosen yet), but based only on the conceptual view being defined. This is critical because one of the fundamental requirements for a conceptual editor is that the operations make sense in the conceptual world of programs. Applying an editor operation to a conceptually valid program must preserve conceptual validity.

The description of the model and of the operations needs to be complete and precise—formal—so that the editor designer has enough information to design an appropriate system image (user interface), and so that it may be possible to evaluate empirically how
successful the editor is at communicating the intended conceptual model. Since one of a conceptual editor's major goals is to convey a specific model of software, it is important that there be an appropriate record of the intended model for comparison.

1.4 Formal Specification of a Type Declaration Conceptual Editor

To show the feasibility of the proposed approach to program editors and to provide an example of a precise description of a conceptual model and of an editor's capabilities, we have, as part of this dissertation, designed and formally specified an editor for a small subset of a modern programming language. Specifically, the editor allows the construction and manipulation of type declarations. These type declarations involve type expressions that are similar to those of traditional languages, such as Pascal, but have been extended to provide a sufficiently rich domain for exploring important issues. In particular:

(i) Type declarations take arguments, i.e., they can be used to declare new type constructors and not just new instances of types.

(ii) Type declarations can be nested, i.e., it is possible to declare local type declarations inside other type declarations, and standard scope rules apply to define the visibility of the declared type constructors.

This "domain" allows us to explore some of the more challenging issues in modeling and specifying a program editor, such as modeling context and specifying context-dependent operations such as conceptual cut and paste, while simplifying the domain so that main points of interest are not obscured by mundane detail.

The conceptual model for our domain consists of four main conceptual entities: type declarations, type expressions, position, and type context. See Figure 1 for examples of these entities. A type declaration consists of a header (with the identifier of the new type
Figure 1: Type Declarations, Type Expressions, and Type Context
constructor and identifiers for the type arguments), a body (the type expression describing
the new type constructor), and local scope (where local type declarations can be nested). A
type expression is essentially a type constructor and a sequence of type expressions (the
arguments to the type constructor). Finally, the type context is a function mapping the type
constructor identifiers to their arities (the numbers of type arguments that they expect).
Each of these entities and operations on them is specified using the RESOLVE language
[94].

The editor has a small yet complete set of operations. It is possible essentially to create
new entities, modify them, and delete them. To be able to express the target of these
operations, the editor introduces another entity called the position. A position is a path
through a type declaration or a type expression hierarchy that determines the current area
of interest in the edited structure. In so doing, the position indicates the argument(s) to a
chosen operation. We argue that, although limited in the scope of the entities that can be
edited and in the functionality provided, this editor addresses most of the interesting and
challenging specification issues for a full-fledged editor for a complete, modern
programming language.

1.5 Contributions

The main focus of this research is to investigate the limitations of existing program
editors and how they fall short in supporting the programming task, and to provide a
viable alternative. The main contributions of this work are:
• Definition of and requirements for a new class of program editors called conceptual editors. A conceptual editor supports, promotes, and even enforces a high-level, consistent conceptual model of programs and program construction. In this way, it tries to bridge the gap between the programmer's mental model of software and the actual objects the programmer is allowed to edit.

• Formal specification of a conceptual editor for type declarations. This is a precise description of the conceptual model of programs and program construction that the editor needs to promote. It can be used to implement different interfaces and evaluate them with regard to the goal of conveying an appropriate, high-level model of software.

• Design and specification of several new general-purpose reusable components. The type declaration editor is built on several new components that are designed to be general enough to be useful in other applications. In particular, the concepts for Nested_List, Nested_List_With_Position, Nested_Map_Set, and Nested_Map_Set_With_Position are new general-purpose components that could be used, for instance, in other language processing applications. The model of

1.6 Organization

In Chapter 2 we define a new, extended role for program editors: from simply a tool for building and manipulating program descriptions, to one helping programmers acquire a more appropriate view of software and the software construction process. We introduce the idea of conceptual editors and their requirements as a proposed response to the extended role of program editors. We also discuss some of the technical issues involved in the design of such an editor.
In Chapter 3, we provide a formal description of a conceptual editor for a small but representative subset of the RESOLVE language. Intuitive descriptions and formal models of the conceptual entities involved are provided. A small, yet complete, set of operations for the editor is formally specified. The chapter concludes by delineating how the concepts and models in the chapter could be generalized and extended to build formal descriptions of conceptual editors for the complete RESOLVE language.

Chapter 4 reviews past and current research efforts in the design of program editors, pointing out where these approaches fall short of the goals of a conceptual editor. Chapter 5 concludes the body of the dissertation by summarizing the work, discussing the contributions, and presenting possible future research directions.

The appendices provide a catalog of the general-purpose components designed as part of this work. Appendix A presents the Nested_List_Template and Nested_List_With_Position_Template concepts together with the mathematical machinery needed for their specifications. Appendix B introduces the Nested_Map_Set_Template, Nested_Map_Set_With_Position_Template, and Nested_Map_Set_Path_Template concepts and the related mathematical machinery. Finally, Appendix C provides an implementation for the Type_Declaration_Editor, described in Chapter 3, using the RESOLVE programming language and the concepts previously introduced.
CHAPTER 2

CONCEPTUAL EDITORS

"The power of the unaided mind is highly overrated. Without external aids, memory, thought, and reasoning are all constrained." — Donald Norman

[65, p. 43]

Software design and construction is a challenging activity and existing program editors provide very limited support for this task. This chapter introduces the idea of conceptual editors as tools designed to aid and guide programmers in the program construction process. It describes a new, extended role for program editors and provides an informal definition and a set of requirements/goals that conceptual editors should achieve. It also explores some of the issues involved in the implementation of some high-level conceptual operations such as cut and paste.

2.1 Introduction

Most current approaches to program editing are text- and/or tree-based. The limitations of such approaches arise from the lack of a high-level conceptual model of program structure, and from the lack of support for the high-level cognitive aspects of the
programmer's task. In other words, these editors are designed based on a traditional view of programs as text manuscripts or syntax trees, and on the ability to manipulate and edit the chosen representation(s) accordingly. They are meant only as tools to build arbitrary program descriptions and they provide little or no guidance to the user as to what may be appropriate ways of building programs or of thinking about programs.

Here we take a different view of programs, and particularly of the role of the program editor. Programs are not strings of characters or any other concrete representation; they are conceptual entities built from other conceptual entities to achieve an intended behavior. Each entity has a meaning, a role, and a purpose that go beyond the concrete syntactic representation the formal programming language assigns to the construct. An experienced programmer possesses such a high-level mental view of what programs are. A novice programmer has to struggle to acquire it. The important issue is that whatever the programmer's mental model of software may be, it has a direct impact on how the programmer designs and builds software.

Meyrowitz and van Dam [56, pp. 323–324], in describing the user view of an interactive editor, write:

"The user of an interactive editor is presented with a conceptual model of the system, and with a user interface. ... Each individual forms a personal user model of an editing system. The user model is a personalized, high-level understanding of the conceptual model provided, of the manipulable entities and their interrelationships, of the set of operations allowed on the entities, and of the interaction language used to invoke these operations."

14
In other words, through the way they display programs and through the kinds of operations they allow for manipulating program representations, editors convey and promote a certain view of programs. This view can have a profound impact on the way programmers think and how they build software.

In this respect, program editors have the potential and, we believe, the responsibility to help programmers view programs at the appropriate level and to form an effective mental model of software. For this to be feasible, a program editor must be designed with this goal in mind. The program editor then is not just a tool to manipulate program representations. It becomes a tool to guide and assist the programmer not just in the difficult task of writing software, but in trying to acquire a proper view of software. We call such an editor a "conceptual editor". An appropriate conceptual model of software and software construction is the necessary starting point in the design of a conceptual editor. Furthermore, a conceptual editor must provide a consistent, high-level view of programs, and a system image that enforces and reinforces at all times the chosen conceptual model.

In this chapter, we describe and characterize a conceptual editor for a "traditional" imperative programming paradigm. The fundamental ideas we assume in such a paradigm are: control structures, procedural abstraction, static typing, data abstraction and encapsulation. In addition, we also assume the presence of templates (parameterized modules/types) since the idea of generic components is at the heart of our conceptual model of software. Languages such as Ada and C++ provide enough features to support such a programming paradigm. The rationale behind the choice of this paradigm is that traditional programming techniques (as opposed to visual programming or intentional programming) are still the most widely used for building "industrial strength" software.
and, we believe, will remain so for a considerable time. So there is substantial value in
designing a new editor that promotes an appropriate (language-independent) model for
this kind of programming while providing qualitatively and quantitatively superior
support for the programmer's task.

In the next section we describe the idea of a conceptual model of software
construction by providing several examples and non-examples\(^1\) of conceptual building
blocks. The following section discusses several issues and requirements that must be
considered in the design of a conceptual editor. Finally, we address the issue of conceptual
cut and paste as an example of the technical difficulties and pitfalls to be faced in the
design of a conceptual editor.

2.2 Conceptual Models of Programming

In this section we describe several “pieces” of a conceptual model. The idea is not to
provide a complete conceptual model of software, but enough examples of what we mean
by “conceptual” to show that not every language construct can be considered to be
conceptual. We also describe the language mechanisms (for C++) that might be used to
provide support for the conceptual pieces, but argue that these mechanisms are low-level
and can be used for other conceptually different purposes, some of which may be
undesirable.

An important conclusion is that our conceptual model is language-independent. This
does not mean that the editor will not use what amounts to a graphical/iconic/textual
language to represent and build programs, but it does mean that in the design of the

\(^1\) A non-example is an example of something that does not satisfy a certain property together with an
explanation of why it does not satisfy the property. In this instance we will discuss examples of
language-based constructs that we do not consider to be conceptual, or to have one clear conceptual
interpretation, and why we believe so.
conceptual model, we do not want to be influenced by language mechanisms or by the hardware or operating system—things that usually have a big impact on the design of programming languages.

The conceptual model is however affected by what we call the programming paradigm. For instance, if we choose an object-oriented paradigm, then we might describe a program as a collection of classes and objects, where objects are instances of classes, and classes have methods (operations) and attributes (data fields). On the other hand, if the chosen paradigm is based on abstract data types (ADTs), we might say that a program is a collection of ADTs, where an ADT is a type and some operations on variables of that type. Even for a given choice, there may be more than one model (or more than one way to look at things) that applies to the same entity, e.g., data flow model vs. control flow model of algorithms.

2.2.1 Component vs. File/Class

Component-based software technology is one of the keys to successfully build large software systems [106]. This section points out the difference between “components” and “files” and how components are conceptually related to each other as opposed to being physically related by direct inclusion. It emphasizes that the #include mechanism is just a syntactic construct to allow the programmer to break down the code in different files. It does not convey any direct conceptual meaning (although we might use #include to express some kind of relationship between components in separate files).

ACTI (Abstract and Concrete Templates and Instances) [18] is a theoretical model of the structure and meaning of software subsystems. It can be used as a conceptual model of software structure. A software system is constructed from subsystems of four kinds:
abstract instances and templates and concrete instances and templates. An abstract component contains the abstract description (specification) for some concrete component (implementation). Templates are generic (parameterized) components. Each component in this model contains all the information (or references to information) necessary to give meaning to the component. All dependencies between components are explicitly recorded.

Through disciplined use of classes, inheritance, templates, and #include mechanisms it is possible to map the ACTI model into C++ programs. Unfortunately, the language mechanisms were not designed with the ACTI model in mind, so that they allow too many dangerous and conceptually confusing uses. In addition, they only allow us to express explicitly a few of the significant relationships and dependencies between components. The others must be expressed in structured comments, which of course are only a matter of discipline and are ignored by any C++-based tool.

In Figure 2 we show an example of an abstract template in RESOLVE/C++ [105]. It describes an extension adding a Reverse operation to a component defining a type Queue_Type. Note how the extends relationship between the two components is explicitly recorded in the code. Also note the use of descriptive keywords such as abstract_template and abstract_instance, and the presence of formal specifications to provide an abstract description of the operation's behavior. All these factors contribute to the high-level conceptual view of components, but they appear in RESOLVE/C++ as comments or as preprocessor-consumed keywords and are ignored by the C++ compiler.
```cpp
#include "AT/Queue/Type.h"

abstract_template <
    abstract_instance class Item
>

class Queue_Reverse :
    extends
    abstract_instance Queue_Type <Item>
{
public:

    procedure Reverse () is_abstract;
    // /*!
    ensures
    self = reverse (#self)
    */
};
```

Figure 2: An Abstract Template in RESOLVE/C++
2.2.2 Conceptual Relationships vs. Inheritance

A conceptual model of component-based software construction must rely on conceptual relationships between components (e.g., extends/behavioral substitutability) [24]. C++ provides the inheritance mechanism that can be used to express conceptual relationships but which has at least three major drawbacks: (1) many (conceptually) very different relationships can be expressed with the same syntactic notation [55, 98], so the language does not provide any indication to the programmer as to which relationship a specific use of inheritance encodes; (2) the relationships expressed through inheritance in C++ are purely syntactic: C++ lacks constructs to explicitly express behavioral relationships; (3) in C++ it is possible to express “bad” relationships using inheritance, e.g., unrestricted code inheritance is universally considered dangerous practice, but C++ does not provide any constraints on the use of the mechanism.

In Figure 3 we can compare a concrete template component in RESOLVE/C++ (top) with the corresponding code in plain C++ (bottom). The component in the example refers to an implementation of the abstract template in Figure 2. The two code fragments are equivalent (the C++ preprocessor transforms the RESOLVE/C++ code into the C++ code). Yet the C++ use of inheritance does not convey any information as to what the real relationship between Queue_Reverse_1 and Queue_Base and between Queue_Reverse_1 and Queue_Reverse might be. The RESOLVE/C++ code, instead, states these relationships explicitly. Of course, there is nothing in the C++ compiler used to compile RESOLVE/C++ code that enforces these relationships.
/// Global Context

#include "AT/Queue/Reverse.h"
/*! -include "AT/Queue/Kernel.h"
 */

class Queue_Reverse_1 :
  extends
    Queue_Base,
    abstract Instance Queue_Reverse <Item>
{
  public:
      procedure Reverse ();
};

#include "AT/Queue/Reverse.h"
template <
  class Item,
  class Queue_Base
>
class Queue_Reverse_1 :
  virtual public Queue_Base,
  virtual public Queue_Reverse <Item>
{
  public:
      virtual void Reverse ();
};

Figure 3: RESOLVE/C++ vs. C++
2.2.3  Operation Abstract Description vs. Function Prototype

Operations (procedures and functions) are a very important building block for components. C++ recognizes the need for separation between interface and implementation of operations, but the function prototype (operation interface) in C++ clearly falls short of a conceptual operation interface. No information about the behavior of the operation is explicitly required, so the only pieces of information guaranteed to be present are the returned type and the parameter types. On the other hand, a conceptual model of an abstract description for an operation should include in addition to the operation name, parameter profile, and returned type (for functions), a complete specification of the behavior of the operation. Only with this information always present, can a programmer reason about the behavior of an operation call without the need to break the abstraction (and conceptualization) and having to look at the operation’s implementation. See Figure 2 for an example of the abstract description of a RESOLVE/C++ operation.

2.2.4  Abstract Parameter Mode vs. Parameter Passing Mechanism

C++ includes two parameter passing mechanisms: “by value” and “by reference”. The advantage of passing an argument by value should be the protection afforded by the language. An argument passed by value is guaranteed not to be changed by the operation. However, if we view the argument as an abstract value and if the type of the argument happens to be implemented as a pointer to some representation, the only thing the language can guarantee and enforce is that the operation will not change the value of the pointer. Nothing can be said or done to prevent the operation from changing the abstract value of the argument.
In RESOLVE we introduce the idea of abstract parameter modes [19]. They are specification notations and are not related to the language passing mechanism. They just simplify the specification of operations in RESOLVE.

- An alters-mode operation parameter might be changed by executing the operation. Its final value is defined by the ensures clause.
- A produces-mode parameter gets a new value that is defined by the ensures clause. The new value does not depend on the parameter’s old value.
- A consumes-mode parameter gets a new value that is an initial value for its type, but its old value generally is relevant to the operation’s effect.
- A preserves-mode parameter suffers no net change in the value between the beginning of the operation and its return, although its value might be changed temporarily while the operation is executing.

Although there is not much that can be done statically to enforce the abstract modes, by decoupling them from the language parameter passing mechanism and by making them part of the operation’s specification, our conceptual model avoids the false sense of security that might arise from the use of “by value” parameter passing, and makes it clear that it is not the language’s responsibility to protect some arguments from being changed.

2.2.5 Chain_Position_Template vs. Pointers

We clearly need to be able to build dynamically growing and shrinking structures. However, C++’s answer is the traditional pointer construct. It is really difficult (if not impossible) to reason at a high, conceptual level about code that deals explicitly with pointers. On the other hand, in our conceptual model of software, we provide a variety of components that define the types of dynamically resizeable objects. For those situations
where the existing components are not adequate, we provide abstractions such as the
Chain_Position_Template [74, 40, 39], which abstracts the idea of pointers (at least for
acyclic structures), providing the programmer with a safe, conceptually clean, formally
specified, high-level substitute for pointers.

2.3 Design of Conceptual Editors

Having established that language mechanisms and constructs are not tantamount to
"conceptual" pieces of a model of software, in this section we address the following
questions:
(i) Given a conceptual model of software and software construction, how could an editor
be designed that supports, promotes, and, if necessary, enforces the chosen conceptual
view?
(ii) What are the important issues in the design of a conceptual editor?

We first introduce the notions of "meaningful programs" and "meaningful entities",
and discuss the importance of the context and of making it visible through the editor’s
interface. Then we emphasize the need for appropriate representations and operations to
convey the chosen conceptual model of programs.

2.3.1 Meaningful Programs

Think of an editor as a tool to transform program descriptions, and the activity of
editing a program as the process of transforming a program description into a new
(different) description by applying one or more of the operations allowed by the editor.
From this point of view, the programmer’s ultimate goal is to transform some starting
description into one that represents a valid program with the intended behavior, by repeatedly applying the operations provided by the editor. We are interested in the intermediate descriptions in this process and in their "meanings".

For most existing editors, it is very hard, if not impossible, to assign a meaning to an intermediate description; just think of the kind of intermediate descriptions it is possible to create by using a text editor. The programmer is responsible for "cleaning-up" the final product, and this may involve a lot more than just thinking about producing a program with the required behavior. On the other hand, imagine an editor that allows (or actually forces) a programmer to move from meaningful description to meaningful description until reaching the desired result. In this way, the programmer can just concentrate on editing a program until it has the appropriate meaning (i.e., the intended behavior) without having to worry about other issues or details, since at all times the program is meaningful. This is how we want a conceptual editor to behave.

There can be many ways to define a meaningful program [70]. In the case of a traditional, compiled programming language, we view a program as meaningful if it can be compiled. Of course, the program may still have semantic errors that cause it not to terminate or to crash, but even these programs are meaningful—they have semantics.

To better understand intuitively how existing editors differ from a conceptual editor, see Figure 4. On the right, the rectangle represents the space of possible program meanings (which according to our definition is the set of all possible behaviors of a compilable program). On the left, the three ovals represent the spaces of program descriptions that can be generated by three different kinds of editors: a plain text editor, a syntax-directed editor (that only allows programs conforming to the context-free syntax of the language), and a conceptual editor (that only allows meaningful programs). Within
each of these spaces, the grayed area represents those descriptions that can be mapped by the programming language semantics to a program meaning. The paths inside the spaces of constructible program descriptions provide examples of editing sessions for each kind of editor: each line segment represents one editor operation transforming a program description into a new one which may or may not be in the set of meaningful programs.

Figure 4: Spaces of Constructible Program Descriptions
Text editors allow a very large space of program descriptions, while only a restricted subset can be considered to be meaningful. Syntax-directed editors reduce somewhat the space of descriptions that can be generated, but they still allow many that are not meaningful. In addition, editors such as text and syntax-directed editors cannot protect the programmer from transforming a meaningful program into one that lies outside the set of meaningful programs. A conceptual editor, instead, restricts its space of constructible program descriptions to only those that are meaningful, so that it is simply not possible to be outside the space of meaningful programs.

It is conceivable that we could relax our notion of meaningful programs. For example, the visual language/environment Prograph [72] chooses to let the user execute incomplete programs and interrupts execution whenever some missing method or attribute needs to be accessed, prompting the user to fix the problem before resuming execution. So it would appear that if we extend the program model to also include the run-time environment, we might be able to come up with some defined behavior even for such flawed programs. We avoid such a definition, however, since we believe in programming as an engineering activity, where program design and construction follow sound engineering principles and are specification-driven, and we do not wish to encourage or promote this kind of incremental, trial-and-error approach to program construction that interpreters have traditionally embraced.

Actually, thanks to the presence of formal specifications and the notion of behavior of program components, we can extend our notion of a meaningful program to make it even more powerful. For instance, consider the template mechanism in C++. A template class is a parameterized class. When the parameter is another class, the C++ compiler can only check that the class provided as the actual argument at instantiation time has operations
with the appropriate names and parameter profiles—and this only for operations that are actually used in the bodies of the template class operations. In our conceptual model, every concrete software component implements a behavior specified in an abstract component. It is possible to impose restrictions on the actual classes used to instantiate a template (e.g., in Figure 3, the template parameter Queue_Base is required to be an implementation of some abstract instance of Queue_Kernel). This would allow a conceptual editor to check that only components satisfying the restrictions are plugged in as actual arguments to the template. This would guarantee that the actual arguments to the template export the appropriate interfaces (including type(s), operations, and their behaviors—not just function prototypes).

2.3.2 Meaningful Entities

Since conceptual editors must only allow the construction of meaningful program descriptions, it is necessary for all operations to preserve the validity of the description. This may seem too restrictive a condition, especially for programmers used to the flexibility and freedom afforded by traditional editing environments. Our thesis, however, is that an editor providing the appropriate conceptual operations would allow the programmer to think in conceptual terms at all times, and in this way the programmer would feel no need to break the validity of the representation and would not miss low-level undisciplined editing operations.

All operations must act on “conceptually meaningful entities”. Meaningful conceptual entities can be described in different ways, and the description may depend on what the conceptual entities are, which, of course, depends on the chosen conceptual model. In general, when we talk about conceptual entities we are referring to the conceptual building
blocks with which we build programs. In our model, conceptual entities range from traditional programming language notions such as variables, statements, expressions, types, and operations, to higher-level entities such as abstract and concrete templates and instances. These are the building blocks of our programs. The fundamental issue here is that in traditional languages, these entities are meaningless in isolation. A variable or a statement can hardly convey any meaning when considered outside the context of the code where they appear.

So the first step in making an entity meaningful is to include its context, i.e., the set of entities that are used in defining the semantics of the chosen entity. For a variable this will include the variable’s type; for a type it will include the other types used in the definition; for an abstract instance component it will include the other components the instance refers to.

In addition to the context, we also need abstraction mechanisms that allow us to assign a meaning to entities without the need to consider the whole concrete description of the entity. We have described in the previous section the example of an operation’s abstract specification as opposed to a C++ function prototype. In a language like C++, the only way to determine the meaning of a function is to look at the function’s code, because the meaning of an operation is “hierarchically constructed” (in the sense of [18]) from the meanings of called operations. However, when an abstract description is provided together with the operation’s header, the meaning (or behavior) of the operation is stated in explicit terms, and there is no need to construct the operation’s meaning from its implementation. Similarly, the use of mathematical modeling to describe programming types gives us an
abstract view of the type without the need for any details about how the type may actually be represented. These features of the conceptual model allow the semantics of individual program units to be defined independently [18, 24].

So we can define a meaningful conceptual entity as the entity together with its context and any mathematical description that may be associated with the entity. A conceptual editor must allow the construction of such entities and the manipulation of these entities as conceptual units.

Also, since each conceptual entity has a different meaning and a different purpose, no attempt should be made to make the model uniform, i.e., to try to describe all entities with the same general model. As a matter of fact, uniformity is a property imposed by those who use a single model such as text or syntax trees to describe the whole language and the realm of programs. Even if it is supposedly easier to learn a single, uniform model that explains everything, such a choice in the case of programs has a major drawback. Since the conceptual building blocks in programming are (conceptually) so different, using a uniform model tends to obscure significant differences, cause confusion (especially in novice programmers who don’t have yet the higher-level, big picture), and ultimately make the programming task more difficult and error prone.

2.3.3 The Context

A declaration is a conceptual building block that allows a programmer to provide a precise description of a new conceptual entity. In declaring a new entity, we may use other entities that are either built-in to the language or that have been previously declared. At a given place in the program, let’s call context the set of all the available entities that are either built-in or user-declared, i.e., those entities that can be used at that point to build a
new piece of the program. Usually, with traditional editors, it is the programmer's responsibility to determine the context at each point in the program. This has to be done by viewing the whole program description and by using the scope rules of the language. This is unfortunate for several reasons. First, since the editor provides only one view of the program, with no explicit view of the context, it is not a trivial task for the programmer to find out what's available and what's not. Second, lack of knowledge of the current context is the cause of many errors, some of which not even the compiler can catch. Finally, the context is necessary to give meaning to an entity, so clearly it is a crucial piece of information to a programmer.

Since a program is essentially made up of declarations of conceptual entities, and uses of these entities, it is also possible to take the context view of a program. Imagine that in addition to displaying a program's description, we show the available entities grouped by kind. So, for example, we might have the set of available variables, the set of available types, the set of available procedures, etc. For each kind of conceptual entity, the editor can display which such entities are available at each point in a program. The context is made explicit instead of being encoded only in the program description itself\(^2\). Then, we could think of adding a new procedure or variable to the program by directly editing the context view, instead of adding it by editing the more traditional program view.

---

2. It is important to note that when a declaration appears in the program, its complete description is usually displayed. In the context view, we might only display the information that is necessary to use the entity, since that is the main purpose of the context.
In Figure 5 we provide an example of the two views. We have on the left a simple

```
PROGRAM

program Example
  var x: integer

procedure A
  var y: integer

  procedure B
  begin...
  end

begin...
end
```

```
CONTEXT

VARcABLES
  x  y

PROCEDURES
  A  B
```

Figure 5: Program View and Context View

program view with a couple of nested procedures, while on the right we have the context view at a specific place in the nested structure. The nested structure view could be used to navigate through the program, and the context view would be useful both as an explicit
display of the context available and possibly as an alternative editing window.

Note that the traditional textual view of the program is chosen only to show the distinction between program description and context view. As we discuss in the next section, a conceptual editor must use representations that convey the conceptual model behind the edited structures. Text alone is clearly not an adequate representation. Note also that the context in the figure does not show any of the available built-in entities.

2.3.4 The Representation(s)

Program editors allow the programmer to create and manipulate program descriptions. The choice of what representation(s) to use to describe programs and the entities involved in program construction is as fundamental to the design of a conceptual editor as the conceptual model itself. Program representations have a substantial impact on how programmers view programs and how they expect to interact with the editor to manipulate program descriptions.

For instance, consider an editor using a purely textual representation. Whether the editor has text editing features or it is a pure structure editor where the only operations allowed are guaranteed to maintain the integrity of the underlying syntax tree, the user is bound to view the program as a string of characters. As a result, the programmer is forced to constantly map from the low-level model of programs used by the editor to the (hopefully) high-level model in his mind. In addition, if the editor only allows structural operations on the abstract syntax tree, the programmer is also very likely to resent the apparently arbitrary restrictions imposed by the editor on the manipulation of what appears to be plain text. This, we believe, is one of the major causes for the general lack of success of purely structural editors and why most current approaches to language-based
program editor design seem to choose a hybrid solution where both text and structure editing are allowed (with the result that the only model of programs effectively promoted by these editors is a low-level, textual one).

Norman [65, p. 49] addresses the issue of representation as follows:

“A good representation captures the essential elements of an event, deliberately leaving out the rest. ... The critical trick is to get the abstractions right, to represent the important aspects and not the unimportant. ... Herein lie both the power and the weakness of representations: Get the relevant aspects right, and the representation provides substantive power to enhance people's ability to reason and think; get them wrong, and the representation is misleading, causing people to ignore critical aspects of the event or perhaps form misguided conclusions.”

This passage makes several interesting points when applied to program representations. First of all, the representation is the key to a successful conceptual editor. If it captures the nature of the conceptual model, it can promote it, and in this way provide support for the programmer's task. On the other hand, the wrong representation may doom an editor to failure. For instance, consider the two representations in Figure 6. The first one is the RESOLVE/C++ code from Figure 3 describing an implementation of an extension to a Queue_Kernel component. The second one is a component coupling diagram (CCD) [105], that is meant to capture only the conceptual relationships between the Queue_Revers e_1 component and those it depends on. Both descriptions are precise and unambiguous, but they differ in the information they convey. For certain tasks the
Figure 6: RESOLVE/C++ Code vs. Component Coupling Diagram
textual representation may be adequate. But for tasks that explicitly deal with component relationships, the CCD representation is more direct and compact, and conveys only the necessary information.

This brings up another important point in the quote above, namely, that the representation(s) must clearly display those aspects of the program that can help the programmer while hiding those that can be confusing and/or irrelevant. For example, as we pointed out in Section 2.3.3, the context is a useful and necessary aspect of programs that should be explicitly displayed, while the language's concrete syntax is only needed by the machine to interpret the program and tends to hinder the programmer with irrelevant details. So a conceptual editor may use multiple representations to guarantee that all essential views of the conceptual model are available. At the same time it must free the programmer from the need to pay attention to syntactic details by taking care of them automatically.

Finally, Norman points out that a misleading representation can cause the user to reach the wrong conclusions. This is a common problem with all modeling situations where we replace the "real world" with some model, and then we proceed to reason and draw conclusions about the real world by observing and manipulating the model. The model, being an abstraction of the real world, only depicts some of the aspects of the real world and ignores the others. However, the danger is that the model may actually have properties that are not properties of the real world we are trying to represent. This clearly can cause us to draw the wrong conclusions.

In choosing an appropriate representation for programs we need to make sure that any notation we introduce (textual or graphical) is precise and unambiguous, so that anything that can be expressed in the notation has a well-defined, unique meaning.
Yet this is not enough. We have already discussed how a purely textual representation is a poor choice for structure editors since programmers are misled into thinking of programs as text and frustrated by the restricted manipulations allowed by the editor. This is not a problem of text only. Any representation that affords operations that do not make sense conceptually will suffer the same drawbacks. For instance, flow chart diagrams are made up of boxes and connecting arrows. But only some restricted combinations of boxes and arrows are meaningful as code fragments. The problem is that the notation itself does not in any way convey information as to which diagrams make sense and which do not (in the same way that nothing in a piece of text can tell us whether it is a valid program fragment or not). So a programmer interacting with an editor using flow charts as the representation for actual code, is likely to be misled into believing that certain (invalid) combinations of boxes and arrows should be allowed. This will result in the programmer being either frustrated by the editor when such combinations are prevented by the editor, or left to deal with the problem of building a meaningful flow chart without help from the editor, if the editor does not prevent invalid combinations.\(^3\)

It is worth noting that in spite of all the drawbacks of purely textual representations, it is not clear that visual (graphical, iconic, etc.) representations are the ultimate answer. Petre [71] discusses some interesting results of research into the relative effectiveness of textual and visual representations for software. Among the problems with existing visual notations, Petre points out that although graphical representations appear to offer the potential for providing a more direct mapping between the descriptions and the conceptual objects being manipulated by providing representations close to the conceptual domain,

\(^3\) Note that the CCD notation introduced in this section may suffer from the same kind of shortcomings we are attributing to flow chart diagrams.
meeting this potential is a challenge. Furthermore, she asserts that “graphical representations can take longer to read and understand, and they are often misunderstood by novices, who cannot 'see' the available cues.”, and that many of the current graphical programming languages and environments are no more than ‘textual’ languages where the familiar ASCII set has been replaced with an alternative fixed vocabulary of symbols.

The conclusions are that (1) a representation is not necessarily good just because it is graphical; (2) symbols and notations in a graphical representation must be chosen so that they convey the desired conceptual model; (3) the best representation is probably going to be a combination of text and visual elements: one that exploits the advantage of both notations and avoids as much as possible their shortcomings; (4) finally, multiple representations are the key to making all essential aspects of software clearly available to the programmer.

2.3.5 The Operations

Each editor’s operation must be based on the conceptual model itself. In other words, it needs to make sense in the realm of the conceptual model, and it must not be chosen based on the given representation but on what the representation stands for.

For instance, consider the CCD notation introduced in the previous section. A CCD is made up of boxes and arrows and it depicts conceptual relationships among components. So the operations allowed on a CCD cannot be operations to add a box or to connect two boxes with an arrow. An example of a conceptual operation is an operation allowing the user to create or add a new extension to a given component (see Figure 7).
Figure 7: Adding an Extension
In addition, each editor's operation must satisfy the condition that it transforms a meaningful program into a meaningful program, i.e., the space of meaningful programs is closed under the available operations.

This requirement has some interesting implications for the interaction between the programmer and the editor. Conceptually each operation needs to be atomic (to be able to maintain the validity of the edited description). But it may require a sequence of smaller steps that is imposed by the editor on the programmer and which has to be completed before the operation is complete (and before the programmer can go on to the next operation).

Let's look at some examples. Fundamental operations in a conceptual editor allow the programmer to add a new entity, remove an existing entity, and change an existing entity (in some controlled fashion). In the following examples, for simplicity, we will consider only traditional programming language entities.

2.3.5.1 Add

Adding a new entity may involve declaring a new variable, or a new type, or a new procedure, or adding a statement to a procedure body. For each new entity, enough information needs to be specified at creation time so that the entity is a meaningful conceptual entity. This information depends on the kind of entity. For example, a new variable needs to be assigned a name (unique identifier) and a type; a new type needs a name and a type expression defining the type; a new procedure needs a name, a parameter profile, and a body. Some of these pieces of information may be incomplete at creation time (e.g., the body of a procedure, or the type expression in a type declaration), but the programmer is required to supply enough information to ensure the new entity has a
meaning before it is used. The reason for this is that an entity must (conceptually) make sense before it can be used. And trying to modify the meaning of an entity, once it is heavily used elsewhere, is a tricky operation that should be avoided when possible by careful design before program construction.

Once enough information has been specified by the user, the new entity appears in the program and in the appropriate context. Other than the fact that the user is forced to input the necessary information at creation time, this kind of operation is straightforward and reasonably simple.

2.3.5.2 Remove

Removing an existing entity poses interesting issues. If there is no other entity depending on the one to be removed, i.e., if the entity to be removed does not appear in the context used to define some other entity, then the entity can be removed without condition. Remove becomes more complicated when there are other entities depending on the one to be removed. In this case, we need to consider the meaning of the remaining entities once it has been removed. According to our notion of a meaningful conceptual entity, the remaining entities generally have no meaning. For example, suppose we wanted to delete a type declaration while there are still variables declared of that type. Then we could not decide if certain operations on such variables still make sense. For these reasons, we may simply prevent the user from deleting the entity until all dependencies have been removed.
The editor may provide support for facilitating the removal of such an entity. If users really want to get rid of an entity, then they'll be willing to get rid of all the dependencies on it. The editor might help by displaying all the existing dependencies, by offering to remove them all automatically (when that is feasible and makes sense) or to remove them selectively by asking the user to make a reasonable choice for each dependency.

Note that the remove operation is considered atomic since it must transform a meaningful program into another meaningful program. This means that the user has to go through the whole process of removing all dependencies before performing any other operation. Of course, the editor might allow a user to abort the operation at any time to restore the situation before the remove operation was invoked.

2.3.5.3 Change

The ability to modify an existing entity is probably the most important and useful of the basic operations. It is also necessary to restrict it in such a way that it only allows modifications that do not compromise the validity of the edited description. Just as in the case of the remove operation, if no other entity depends on the entity to be modified, then the chosen entity can be modified according to its own model. For example, if we wanted to add a formal parameter to a procedure declaration, and the procedure was not invoked anywhere, then we would be allowed to modify the procedure header by adding the new parameter. However, if the procedure was indeed used somewhere in the program, adding a new formal parameter would require adding an appropriate actual parameter at each call.

As another example, consider changing the type of a formal parameter. Again, if nothing depends on it, there is no problem. Assuming that the parameter is actually used somewhere (e.g., there is a call to the procedure to which it is a formal or code in the
procedure body that uses it), changing its type clearly has an impact on those entities. And in this case, even though the editor would have no trouble pointing out to the user all the trouble spots, it might be very tricky for the editor to take automatically, or even to suggest an appropriate course of action to resolve each dependency.

At this point, we can recognize a pattern in the required behavior for any operation in the conceptual editor. Whenever an operation is invoked, if it only creates new dependencies but it does not affect any existing ones, the operation can be executed without additional constraints. If, however, performing the operation would affect existing dependencies, the editor must force the user to handle all conflicts before the operation completes.

It is clear by the relative complexity of these operations, that more complex operations such as general (conceptual) cut and paste will impose even more interaction constraints on the programmer. The important conclusion we reach from this observation is that whenever a programmer performs one of these general operations, say on a textual representation using a text editor, the implications for the meaning of the program are substantial and far reaching. Traditional editors not only allow such operations, they also fail to aid the programmer in cleaning up the resulting mess and in trying to avoid the potential pitfalls arising from the unconstrained use of such operations. So although it may seem that the restrictions and constraints imposed on the user by a conceptual editor are too many and too strict, they must be viewed in light of an understanding of what the unconstrained operations really do. These constraints are there in a conceptual editor simply because unconstrained operations conceptually do not make sense.
In the next section we address in some detail the technical issues involved in the design of conceptual cut and paste operations. As mentioned earlier, these operations are substantially more complex than those described in this section, and both their behavior and the editor-user interaction need to be carefully designed.

2.4 Conceptual Cut and Paste

Most editing environments provide a set of related operations to manipulate (e.g., delete, duplicate, move) possibly large chunks of the edited structures. Text editors, syntax directed/structure program editors, visual program editors, word-processors and desktop publishing tools, spreadsheets, draw/paint programs, math equation editors, 3D geometric modelers and CAD software, sound editors and movie editors all support a similar model for these operations.

This model consists of three related operations, copy, cut, and paste, a separate storage area, usually called the clipboard, and a selection that specifies the place or structure in the edited object where the operation must take place. Copy does not affect the currently edited structure. It just makes a copy of the selected entities and stores it in the clipboard. Cut instead removes the selected entities from the edited structure and stores them in the clipboard. Paste usually replaces the selected structures with the contents of the clipboard (and leaves the clipboard unchanged, so a copy is made). We will refer to this general setup as “Cut & Paste” (C/P).

The main purposes of C/P are to save input time on structures that are similar to existing ones by duplicating and then editing them, and to move or remove large chunks of the edited structure. At first these operations appear to be extremely powerful and practically indispensable. But they introduce substantial problems.\(^4\)
The first problem is that in most editors there is a mismatch between the editable
descriptions and the meaningful descriptions; that is, between the descriptions that can be
built with the editor and those that are meaningful in the domain the user is trying to
model. For instance, consider a text editor used to edit programs. The only representation
is a plain string of characters and the editor has no idea that the edited strings are supposed
to represent a program. Thus the editor allows the creation of text strings that clearly are
not programs (and may not even resemble programs). Another example of this mismatch
is a general-purpose drawing application used to build diagrams in some chosen notation,
e.g., architectural layouts. Again the drawing application is not aware of any meaning
associated with symbols such as chairs and tables, and would cheerfully allow the user to
set a chair inside a wall or to draw meaningless symbols in the diagrams.

In these cases, C/P is bound to corrupt any description, because it operates on a lower
level representation of the intended structures. In the text editor example, general C/P will
preserve the integrity of the edited string, but it will most likely mangle the actual program
(both syntactically and semantically). Similarly, the drawing application’s C/P will still
produce acceptable pictures even though they are completely meaningless as architectural
drawings.

So we can ask the question: Is the only problem with C/P due to this description
mismatch or is there a more fundamental problem with C/P? In other words, if we built an
editor (conceptual editor) in which only meaningful structures could be edited (i.e., there
would be no mismatch), could we specify and expect a reasonable behavior for C/P?

4. The operations that cause problems are cut and paste. Since it does not modify the edited structure, copy
is a relatively safe and conceptually clean operation if the selection being copied makes sense
conceptually.
Unfortunately, even in this case we face a big problem. In structures such as programs there are a great number of dependencies and connections between different parts in the structure. Every time a piece is removed some dependencies may be broken, and every time a piece is pasted new dependencies may be introduced and conflicts can arise between existing and pasted entities. In other words, the problem is the large number of loose ends that may have to be taken care of whenever C/P is performed.

This problem with C/P is intrinsic to the nature of the edited structures and to the C/P operations themselves. We claim that it is a bad idea to have any operation that breaks/creates many connections, unless there is an obvious appropriate behavior that the editor can implement. The complexity of the connections and dependencies that may have to be broken or created is a measure of the conceptual complexity of an operation. It is pretty clear that we should avoid operations that, due to exceeding conceptual complexity, are likely to have impact beyond the user's understanding.

There are basically two approaches with which we can choose to handle this problem:

(i) Substantially restrict the applicability and scope of C/P.

(ii) Come up with a different set of operations that can help speed up the entry of structures similar to existing ones and move structures around, while not dealing with a large number of dependencies.

A combination of the two approaches may be an acceptable solution to the problem.

For instance, we could provide what we call syntactic C/P, i.e., C/P where all dependencies on the context get thrown out, and only the selected syntactic structures are copied and pasted; and, in addition, operations to duplicate entities within the same
context (so that all dependencies required by the copied structure are satisfied), to move a declaration up or down the hierarchy (as long as no dependencies are broken and no conflicts arise), etc.

Note that as we have discussed in Section 2.3.5.2, it is fairly straightforward to define a "reasonable" behavior for the cut operation. Since cutting a structure that other entities depend on would invalidate them, the editor should force and assist the user in the process of removing all dependencies before the cut operation can be performed.

2.5 Chapter Summary

Program editors affect the way programmers view programs and program construction. Thus, they have the potential and the responsibility to promote, support, and even enforce a high-level, consistent conceptual model of software. Traditionally, program editors are viewed simply as tools to build and manipulate program descriptions, and therefore they do not address this broader and more fundamental role.

We introduce the idea of a conceptual editor as a program editor whose main objective is to convey an appropriate high-level model of software. Some of the requirements that a conceptual editor must satisfy are that:

- It is designed around an appropriate conceptual model of software.
- It provides one or more representations that reflect the underlying model, and explicitly displays the conceptual entities involved in the construction of software (even those that are implicit in traditional programs, e.g., the context).
- It allows only operations that are consistent with the conceptual view, i.e., they manipulate conceptual entities and they preserve the conceptual validity of the edited program description.
• The basic, primitive operations provided by the editor should be conceptually simple, and localized in their effect.

If the editor is to be a conceptual editor, no compromise of the conceptual view can be accepted. Such editors provide the ability to interact with the user at a level closer to the user's conceptual view. They also help avoid many errors caused by editors that promote a low-level model and that allow operations without meaning in the conceptual world. Programmers, when first using a conceptual editor, may perceive it to be constraining and restrictive and experience a level of discomfort. However, eventually the advantages of such editors should more than compensate for these feelings.
CHAPTER 3

FORMAL SPECIFICATION OF A TYPE DECLARATION EDITOR

This chapter presents a formal model of a conceptual editor for a small but significant (in terms of key concepts) subset of the RESOLVE language. The models and the specifications are expressed in the RESOLVE language. A detailed description of the RESOLVE mathematical specification language can be found in [38].

3.1 Introduction

In Chapter 2 we described the notion of conceptual editors, their role, and the requirements that they must satisfy. In this chapter we want to provide an example of what the design of a conceptual editor may entail. We will provide formal models for the necessary conceptual entities, and use them to formally specify a conceptual editor for a small subset of the RESOLVE language, a type declaration editor.

This is useful and significant for several reasons:

• The formal model conveys in an unambiguous fashion the intended conceptual model of the structure of type declarations and type expressions. Furthermore, the model is designed so that it conveys only those properties of each entity that are essential.
• A precise, unambiguous description of the conceptual entities and of the editor's functionality is useful as it separates the role of the designer, from that of the implementors of the interface and the functionality. This is particularly relevant because of the requirement that the conceptual editors' operations be chosen based on the conceptual model and not on the used representation.

• As a proof of concept. It is possible to formally specify a fairly complicated tool, and, in principle, we show that it would be possible to formally specify a conceptual editor for a full-fledged specification and programming language like RESOLVE.

• Finally, a precise description of the editor model and functionality would allow the implementation and separate evaluation of multiple views/interfaces.

3.2 Why a Type Declaration Editor?

RESOLVE is both a specification and a programming language. It contains a considerable number of constructs representing many distinct conceptual entities, that programmers may need in order to specify and implement software components. One construct in particular accounts for a sizable portion of any RESOLVE component, namely expressions. An expression is made up of operators and operands, and can be evaluated to produce a value. It reflects the general structure of many other programming constructs, and presents many of the issues involved in modeling and specifying conceptual program editors. For instance, expressions have a nested, recursive structure, they allow the declaration of new identifiers and introduce the idea of nested scopes, they require type checking, etc. In other words, they provide a sufficiently rich subset of the RESOLVE language in which to explore significant issues and problems in the design and specification of general conceptual editors.
Type expressions are expressions that define new math or program types, and they form a proper subset of general expressions. Type expressions are structurally similar to general expressions, but they do not involve some of the interesting issues described above. However, by considering type declarations in addition to type expressions, we can essentially explore all of the important issues of general expressions without unnecessary complexity.

In this chapter, we start by describing type expressions: the intuitive model, the formal model, and other necessary mathematical machinery. After that we provide a similar treatment for type declarations, followed by the machinery necessary to model the concept of position in a type declaration and type expression hierarchies and the concept of type context. Then, we use the formal models for type expression, type declaration, position, and type context to specify a conceptual editor for type declarations. In the last section we discuss how the simple type declaration editor could be generalized to model and formally specify the remaining concepts in the complete RESOLVE language or other modern programming language and to include more powerful editing operations.

The diagram in Figure 8 summarizes the math components (the octagonal boxes) involved in the specification of the editor, and their relationships. An arrow connecting two boxes in the diagram expresses the fact that the source module (the one at the tail of the arrow) uses the definitions in the target module (the one at the head of the arrow), and possibly the definitions in the modules used by the target component, and so on transitively. The rounded rectangle refers to the only program component in the diagram, the abstract description (specification) of the type declaration editor. Of course, this
Figure 8: Module Organization
component is related to all the math modules in the diagram. The dashed-line boxes group together conceptually related modules, and the numbers in the diagram refer to the sections in which the corresponding modules are described.

3.3 Type Expressions

The first conceptual entity we consider is type expressions. We describe an intuitive model of type expressions in RESOLVE. Then we construct a mathematical model of type expressions that precisely captures the essential structural properties of the intuitive model. This is done by introducing the mathematical type TREE, and then restricting this type to enforce the additional constraints that type expressions need to satisfy. Finally we define several useful math operations for type expressions.

3.3.1 Intuitive Model

A type expression is the entity generally used in RESOLVE programs to specify a type in places where a type is expected, e.g., in a type declaration or variable declaration. Some examples of RESOLVE type expressions are presented in Figure 9. A type expression is either an atomic type such as integer or boolean, or the result of applying a type constructor to a sequence of other type expressions, e.g. set of integer. A type constructor can be thought of as a parameterized type, where the number of arguments required is called the arity of the type constructor. We can also view an atomic type as a type constructor of arity 0. In general, a type constructor has a fixed arity. For instance set has arity 1, and function has arity 2. One exception to this rule is the tuple type constructor. It represents the mathematical Cartesian product of its argument types, and has variable arity.
boolean

set of integer

function from DOMAIN to RANGE

tuple ( 
    name: string of character 
    age: integer 
    weight: function from DAY to integer 
)

Figure 9: Examples of RESOLVE Type Expressions

This intuitive model of type expressions has three essential structural characteristics that we want to capture in the mathematical model of type expressions:

(i) A type expression is a recursive (nested, hierarchical) structure, where each level is made up of a label (the type constructor) and a sequence of substructures of the same kind;

(ii) The order of the substructures at each level is important;

(iii) The number of substructures at any level is not arbitrary, but it is determined by the arity of the corresponding label.
3.3.2 TREE THEORY

The reader is probably familiar with the notion of a tree as an abstraction and as a common data structure used in many applications [42]. This structure satisfies properties (i) and (ii), but not (iii). In this section, we present a mathematical theory that defines the mathematical type TREE, which precisely captures the usual notion of a tree, and satisfies the first two of the three required properties. This theory is formally described in the TREE_THEORY_TEMPLATE module (Figure 10).

This module (like all RESOLVE modules) is divided into two parts: the context consisting of the collection of all the external entities upon which this module depends, and the interface consisting of the collection of the new entities exported by this module. In this case, there is only one dependency with the external world, and that is the generic parameter LABEL that determines the type of the labels in the tree. The interface is where all the action takes place. This module defines a new math type TREE, and a couple of operations that can be used to generate TREE values, EMPTY_TREE\(^1\) and COMPOSE. The meaning and the properties of the new type and operations is implicitly defined through three axioms. Axiom 1 states that the EMPTY_TREE cannot result from the composition of a label with a string of trees; Axiom 2 states that COMPOSE is one-to-one; and Axiom induction says that the set of all TREES is inductive, i.e., any of its subsets that contains the EMPTY_TREE and that is closed under COMPOSE has to be the set of all TREES itself.

---

1. Note that in RESOLVE constants are expressed as 0-ary functions. So the intended meaning of the 0-ary math operation EMPTY_TREE is the standard empty tree constant value.
mathematics TREE_THEORY_TEMPLATE

context

parametric context

math type LABEL

interface

math type TREE

math operation EMPTY TREE: TREE

math operation COMPOSE (root: LABEL, children: string of TREE): TREE

axiom 1 is
   for all root: LABEL, children: string of TREE
      (COMPOSE (root, children) /= EMPTY_TREE)

axiom 2 is
   for all root1, root2: LABEL, children1, children2: string of TREE
      (if COMPOSE (root1, children1) = COMPOSE (root2, children2)
        then root1 = root2 and children1 = children2)

axiom induction is
   for all s: set of TREE
      (if EMPTY_TREE is in s and
         for all root: LABEL, children: string of TREE
            (if for all t: TREE
               where (t is in elements (children))
                  (t is in s)
               then COMPOSE (root, children) is in s)
         then s = universal_set)

end TREE_THEORY_TEMPLATE

Figure 10: The TREE_THEORY_TEMPLATE Module
math operation SIZE ( 
    t: TREE
): integer
implicit definition
if t = EMPTY_TREE
then SIZE (t) = 0
else there exists label: LABEL, children: string of TREE
    (t = COMPOSE (label, children) and
        SIZE (t) = 1 + sum i: integer
            where (1 <= i <= |children|)
            (SIZE (children[i]))

math operation HEIGHT ( 
    t: TREE
): integer
implicit definition
if t = EMPTY_TREE
then HEIGHT (t) = 0
else there exists label: LABEL, children: string of TREE
    (t = COMPOSE (label, children) and
        HEIGHT (t) = 1 + max x: TREE
            where (x is in elements (children))
            (HEIGHT (x))

Figure 11: SIZE and HEIGHT Operations for TREES

From these axioms it is possible to prove the Recursion Theorem for TREE_THEORY, which asserts that we can make inductive definitions over TREES (see [21] for a formal statement of this theorem and for the proof). So, for instance, we can define inductively math operations SIZE and HEIGHT (Figure 11) that yield, respectively, the number of labels in a given tree and the number of levels in the tree. The Recursion Theorem guarantees that these are well-defined operations (functions).

It is important to note that TREE_THEORY makes use of STRING_THEORY (which, for convenience, is built-in to the RESOLVE language), and that it implicitly defines another new math type string of TREE.
Unless the reader is already familiar with mathematical structures such as TREE\_THEORY, the connection between the formal presentation in Figure 10 and the intuitive, mental model of a tree may not be straightforward. This will become even more significant later when we introduce other mathematical theories where the reader's experience and background are unlikely to provide an appropriate mental model to use in understanding the new structures. For this reason we will make use of pictures intended to convey an appropriate image of the mathematical structure under consideration. Figure 12 shows a picture of a TREE. Here, we can see the recursive, nested structure of TREES. The explicit representation of strings emphasizes that in trees, the order of the siblings matters.
In addition, this picture provides a visual description of both types involved, TREE and string of TREE, and of their relationship. We will sometime use the terms traditionally employed when talking about trees. We will refer to the labels in a tree as the nodes of the tree, to the label at the top of a tree as the root of the tree, to the subtrees of a given node as the children of the node, and to the nodes with no children as the leaves of the tree.

3.3.3 TYPE_EXPRESSION

It is easy to see that the TREES defined in the previous section satisfy properties (i) and (ii) of type expressions. But clearly there is nothing in the model for TREES that prevents any node from having an arbitrary number of children, that is, TREES do not satisfy property (iii). This means that not all TREES are models of legal type expressions.

In the TYPE_EXPRESSION_TEMPLATE module (Figure 13) we define a new math subtype^, TYPE_EXPRESSION, as a TREE with the constraint that each node has the appropriate number of children. Let's take a closer look at this module.

Since a type expression is going to be a TREE where the label corresponds to a type constructor and the children are the argument type expressions, the module is parameterized by a type TYPE_ID, representing the type of the identifiers of type constructors. In addition, this modules needs a distinguished value, NIL, of type TYPE_ID (again, this is expressed in RESOLVE as a 0-ary function of type TYPE_ID), and a math operation ARITY that, for a value of type TYPE_ID, gives the arity of the corresponding type constructor. The constant NIL is used in the model as a place holder for a type constructor that has not yet been specified. It has arbitrary arity. It is introduced in the

---

2. See [36] for a definition of math subtype and a discussion of how it differs from the notion of math type in RESOLVE.
mathematics TYPE_EXPRESSION_TEMPLATE

context

global context

mathematics TREE_THEORY_TEMPLATE

parametric context

math type TYPE_ID

math operation NIL : TYPE_ID

math operation ARITY (tid: TYPE_ID): integer

restriction

  for all tid: TYPE_ID (ARITY (tid) >= -1) and
  ARITY (NIL) = -1

interface

math facility TREE_THEORY_FACILITY is

  TREE_THEORY_TEMPLATE (TYPE_ID)

math subtype TYPE_EXPRESSION is TREE

  exemplar te

  constraint

    IS_TYPE_EXPRESSION (te)

math subtype TYPE_EXPRESSION_STRING is string of TYPE_EXPRESSION

math operation IS_TYPE_EXPRESSION { t: TREE }

  : boolean

  implicit definition

  if t = EMPTY_TREE

    then IS_TYPE_EXPRESSION (t) = false

  else there exists label: TYPE_ID, children: string of TREE

    (t = COMPOSE (label, children) and

    IS_TYPE_EXPRESSION (t) =

    (if ARITY (label) >= 0

      then ARITY (label) = |children| and

      (for all x: TREE where (x is in elements (children))
      (IS_TYPE_EXPRESSION (x))))

  end TYPE_EXPRESSION_TEMPLATE

Figure 13: The TYPE_EXPRESSION_TEMPLATE Module
model to support the specification of a type expression editor which needs to account for type expressions that have not yet been fully constructed (i.e., they have one or more NIL subexpressions). We create the convention that the value of the ARITY is non-negative for fixed arity type constructors, and it is -1 for NIL and for any other variable-arity type constructor (e.g., tuple). These constraints are expressed formally through a restriction on the acceptable parameters to the math module (at the end of the parametric context section of the module).

The interface part of the module defines the new subtypes TYPE_EXPRESSION and TYPE_EXPRESSION_STRING together with the predicate IS_TYPE_EXPRESSION. In a separate TYPE_EXPRESSION_MACHINERY module (Figures 14-16), we introduce several math operations that will be useful in the following modules and in the specification of the Type_Declaration_Editor component. Here, we describe in some detail the TYPE_EXPRESSION_TEMPLATE module, and only briefly and informally state the meanings of the operations in the TYPE_EXPRESSION_MACHINERY module. When necessary, we will provide a more detailed description where the definitions are actually used.

After instantiating TREE_THEORY_TEMPLATE to define a TREE type with labels given by the type constructor identifier TYPE_ID, we define TYPE_EXPRESSION and TYPE_EXPRESSION_STRING. A TYPE_EXPRESSION is simply a TREE that satisfies the predicate IS_TYPE_EXPRESSION, and TYPE_EXPRESSION_STRING is just a string of TYPE_EXPRESSION. The key definition here is the predicate (boolean-valued math operation) IS_TYPE_EXPRESSION, which decides whether a given TREE satisfies the appropriate constraint (property (iii)) and is therefore a TYPE_EXPRESSION. IS_TYPE_EXPRESSION is defined inductively, and the constraint it imposes is that at
each node in the given TREE, the number of children must match exactly the arity of the label's type constructor (as given by the operation ARITY), unless the arity is -1 (i.e., it is arbitrary), in which case any number of children will do. Of course, based on our intuitive notion of a type expression, if the tree is empty, then it cannot be a TYPE_EXPRESSION.

This definition captures all the properties we required of type expressions, and thus we take it as our (formal) model of type expressions. However, since part of our goal is to be able to design and specify a program component that will be used in building a conceptual editor for type expressions, we need to extend or enhance this model to provide the expressive power to directly refer to specific nodes within the type expression tree. This is done in Section 3.5.1.

Several other math operations related to type expressions will be needed. These are grouped in a separate module called TYPE_EXPRESSION_MACHINERY (Figures 14-16). Note that this module has all the parameters of the TYPE_EXPRESSION_TEMPLATE module plus one more, an instance of the TYPE_EXPRESSION_TEMPLATE module itself. Here are the math operations defined in this module:

- **EXTRACT_LABEL** – Given a TYPE_EXPRESSION, it gives the TYPE_ID of the root's label.
- **EXTRACT_CHILDREN** – Given a TYPE_EXPRESSION, it gives the TYPE_EXPRESSION_STRING made up of the children of the root node.
- **NIL_LEAF** – A 0-ary function (constant) whose value is a NIL leaf TYPE_EXPRESSION (i.e., a type expression with a NIL root label and no children).
mathematics TYPE_EXPRESSION_MACHINERY

context

global context

mathematics TYPE_EXPRESSION_TEMPLATE

parametric context

math type TYPE_ID

math operation NIL : TYPE_ID

math operation ARITY (tid: TYPE_ID) : integer

math facility TYPE_EXPRESSION_FACILITY is

TYPE_EXPRESSION_TEMPLATE (TYPE_ID, NIL, ARITY)

restriction

for all tid: TYPE_ID (ARITY (tid) >= -1) and
ARITY (NIL) = -1

interface

math operation EXTRACT_LABEL ( 
    te: TYPE_EXPRESSION
): TYPE_ID

implicit definition

    te = COMPOSE (EXTRACT_LABEL (te), EXTRACT_CHILDREN (te))

math operation EXTRACT_CHILDREN ( 
    te: TYPE_EXPRESSION
): TYPE_EXPRESSION_STRING

implicit definition

    te = COMPOSE (EXTRACT_LABEL (te), EXTRACT_CHILDREN (te))

math operation NIL_LEAF ( 
): TYPE_EXPRESSION

explicit definition

    COMPOSE (NIL, empty_string)

Figure 14: The TYPE_EXPRESSION_MACHINERY Module
math operation EXTRACT_SUBEXPRESSION ( 
  te: TYPE_EXPRESSION 
  pos: integer 
): TYPE_EXPRESSION 
implicit definition 
  if  pos >= 0 and pos < |EXTRACT_CHILDREN (te)| 
  then 
      there exists a, b: TYPE_EXPRESSION_STRING 
      (|a| = pos and 
      EXTRACT_CHILDREN (te) = 
          a * <EXTRACT_SUBEXPRESSION (te, pos)> * b) 
  else 
      EXTRACT_SUBEXPRESSION (te, pos) = TYPE_EXPRESSION.base_point 

math operation EXTRACT_SUBEXPRESSION ( 
  te: TYPE_EXPRESSION 
  path: string of integer 
): TYPE_EXPRESSION 
implicit definition 
  if  path = empty_string 
  then EXTRACT_SUBEXPRESSION (te, path) = te 
  else 
      EXTRACT_SUBEXPRESSION (te, path) = 
          EXTRACT_SUBEXPRESSION ( 
              EXTRACT_SUBEXPRESSION ( 
                  EXTRACT_SUBEXPRESSION (te, first (path)), 
                  all_but_first (path)) 
          ) 

math operation OCCURS_COUNT ( 
  te: TYPE_EXPRESSION 
  id: TYPE_ID 
): integer 
implicit definition 
  if  te = NIL_LEAF 
  then OCCURS_COUNT (te, id) = 0 
  else 
      if  EXTRACT_LABEL (te) = id 
      then 
          OCCURS_COUNT (te, id) = 
              OCCURS_COUNT (EXTRACT_CHILDREN (te)) + 1 
      else 
          OCCURS_COUNT (te, id) = OCCURS_COUNT (EXTRACT_CHILDREN (te)) 

Figure 15: The TYPE_EXPRESSION_MACHINERY Module (continued)
```
math operation OCCURS_COUNT (  
    tes: TYPE_EXPRESSION_STRING  
    id: TYPE_ID  
  ) : integer  
implicit definition  
if   tes = empty_string  
then OCCURS_COUNT (tes, id) = 0  
else OCCURS_COUNT (tes, id) = OCCURS_COUNT (first (tes), id) +  
    OCCURS_COUNT (all_but_first (tes), id)

math operation DIFFER_ONLY_AT_CURRENT (  
    tel: TYPE_EXPRESSION  
    te2: TYPE_EXPRESSION  
    path: string of integer  
  ) : boolean  
implicit definition  
if     path = empty_string  
then  
  DIFFER_ONLY_AT_CURRENT (tel, te2, path) = true  
else  
  DIFFER_ONLY_AT_CURRENT (tel, te2, path) =  
    (EXTRACT_LABEL (tel) = EXTRACT_LABEL (te2) and  
    all_but_nth (EXTRACT_CHILDREN (te2), first (path)) =  
    all_but_nth (EXTRACT_CHILDREN (tel), first (path))) and  
  DIFFER_ONLY_AT_CURRENT (  
    nth (EXTRACT_CHILDREN (tel), first (path)),  
    nth (EXTRACT_CHILDREN (te2), first (path)),  
    all_but_last (path))

end TYPE_EXPRESSION_MACHINERY
```
• **EXTRACT_SUBEXPRESSION**\(^3\) – Given a TYPE_EXPRESSION and an integer, it gives the child TYPE_EXPRESSION at the position specified by the integer, where position 0 stands for the first (leftmost) child of the root node, position 1 for the second, and so on.\(^4\)

• **EXTRACT_SUBEXPRESSION** – Given a TYPE_EXPRESSION and a string of integer, it gives the child TYPE_EXPRESSION obtained by following the path defined by the given string of integer through the given TYPE_EXPRESSION.

• **OCCURS_COUNT** – Given a TYPE_EXPRESSION and a TYPE_ID, it gives the number of labels in the TYPE_EXPRESSION that match the given TYPE_ID.

• **OCCURS_COUNT** – Given a TYPE_EXPRESSION_STRING and a TYPE_ID, it gives the number of labels in the TYPE_EXPRESSION_STRING that match the given TYPE_ID.

• **DIFFER_ONLY_AT_CURRENT** – A Boolean function that, given two TYPE_EXPRESSIONS and a string of integer, is true if and only if the two TYPE_EXPRESSIONS differ only at the child TYPE_EXPRESSIONS obtained by following the path defined by the given string of integer through the two TYPE_EXPRESSIONS.

This concludes our description of type expressions and the related math model and definitions.

---

3. For convenience, we make use of overloading of math operation names by defining distinct math operations performing similar functions with the same name and distinguishing them by the number and types of their arguments.

4. Note that in RESOLVE all math operations are total functions. Whenever a function would be undefined (e.g., if the position given to **EXTRACT_SUBEXPRESSION** were negative or greater than the number of children), we define it as the distinguished value base_point of the appropriate type (in the example of **EXTRACT_SUBEXPRESSION** this is TYPE_EXPRESSION.base_point). We assume that all math types have such a distinguished value.
3.4 Type Declarations

This section follows a format similar to that of the previous section. We first describe an intuitive model for type declarations. The intuitive model is made general enough to allow us to explore the issues connected with declarations, nested scopes, context, etc. Next we formalize the model by introducing the PARTIAL_FUNCTION TEMPLATE module and the new NESTED_MAP_THEORY module, and then restricting this model to fit the requirements of the intuitive model. We conclude this section by describing several useful math operations for type declarations.

3.4.1 Intuitive Model

A type declaration is the construct used in RESOLVE to specify a new type and giving it a name. The model of such a construct could be as simple as a pair with the name of the new type and a type expression to describe the new type. However, since we are interested in exploring the more general issues of nested scopes and context, we extend our notion of a type declaration to allow for the definition of new parameterized type constructors. The new type declarations have a structure that is very similar to that of procedures in Pascal. They have a header that defines the name of new type constructor and the names of the formal parameters. They have a body (a type expression) that specifies the structure of the new type constructor using other type constructors and the formal parameters. Finally, they have a local scope where other (local) type declarations can introduce new local type constructors.

In Figure 17 we have an example of a type declaration. It declares a new type constructor called Example with two arguments Item1 and Item2. Example has arity 2 (i.e., the number of formal parameters), and Item1 and Item2 have both arity 0^5. The
body is a type expression that uses one built-in type constructor, function, the two formal parameters, Item1 and Item2, and two local type constructors, Point3D and Segment3D. Point3d and Segment3D are declared in the local scope. Both are type constructors of arity 1.

The scope rules for our type declarations are similar to the scope rules for nested Pascal procedures with one difference: the order of the local declarations does not matter. So a locally declared type constructor (e.g., Point3D) is visible anywhere in the body and the local scope of the enclosing type declaration (Example), and nowhere outside the enclosing type declaration.

This intuitive model of type declarations has three essential structural characteristics that we want to capture in the mathematical model of type declarations:

(i) A type declaration is a recursive (nested, hierarchical) structure, where each level is made up of a label (containing the type declaration header and its body) and a set of substructures of the same kind (the local type declarations);

(ii) The order of the substructures at each level does not matter;

(iii) The number of substructures at any level is arbitrary.

The trees described in the previous section could be an adequate model of type declarations. However, they fail to capture property (ii). What we need is a structure that is sometime referred to as "unordered trees", and that we call NESTED_SET.

---

5. For simplicity, we assume that all type declaration formal arguments have arity 0.
Figure 17: Example of Type Declaration
3.4.2 NESTED_SET_THEORY

Intuitively NESTED_SET_THEORY is obtained by replacing the string, grouping the children of a node in a tree, with a set. Compare the TREE in Figure 12 with the NESTED_SET in Figure 18. Structurally they are very similar, but they differ in the key property that the children of a node in a TREE are ordered implicitly by being organized in a string, while the children of a node in a NESTED_SET are unordered elements of a set.

Formally we can define a NESTED_SET_THEORY module (Figure 19) that exports the NESTED_SET type (and also the set of NESTED_SET type), the constant function EMPTY_NESTED_SET and the constructor function COMPOSE. Essentially this theory is the TREE_THEORY module (Figure 10) where we have replaced the string child grouping type with set.

Though NESTED_SET is the math type we wish to use to model type declarations, we are ultimately interested in using these models to specify program components involved in the implementation of a type declaration editor. Unfortunately, there is an intrinsic problem with the design of any component exporting a type modeled by NESTED_SET. Such a program type has to have operations to add and remove children at a given level in the hierarchy of the corresponding data structure. The remove operation needs to able to refer to the specific child that has to be removed. In trees, the position of the child in the string of children of the given tree can be used to refer to any specific child. In an object modeled by NESTED_SET this positional information is not available (by design). So the only way we can refer to a child is by having a separate copy of it. But if we have to keep copies of all the elements in an object modeled by NESTED_SET, this would defeat the

---

6. For convenience, SET_THEORY is assumed to be built-in to the RESOLVE language.
Figure 18: A NESTED_SET Object
mathematics NESTED_SET_THEORY_TEMPLATE

context

parametric context

math type LABEL

interface

math type NESTED_SET

math operation EMPTY_NESTED_SET: NESTED_SET

math operation COMPOSE (    root: LABEL    children: set of NESTED_SET ) : NESTED_SET

axiom 1 is
    for all root: LABEL, children: set of NESTED_SET
    (COMPOSE (root, children) /= EMPTY_NESTED_SET)

axiom 2 is
    for all root1, root2: LABEL, children1, children2: set of NESTED_SET
    (if COMPOSE (root1, children1) = COMPOSE (root2, children2)
    then root1 = root2 and children1 = children2)

axiom induction is
    for all s: set of NESTED_SET
    (if EMPTY_NESTED_SET is in s and
    for all root: LABEL, children: set of NESTED_SET
    (if for all ns: NESTED_SET
    where (ns is in children)
    (ns is in s)
    then COMPOSE (root, children) is in s)
    then s = universal_set)

end NESTED_SET_THEORY_TEMPLATE

Figure 19: The NESTED_SET_THEORY_TEMPLATE Module
purpose of the data structure. Not to mention that these elements may be arbitrary large objects, and keeping two copies of them would impose unnecessarily high memory requirements on any application built using the component. See the Set_Template components in [105] for an example of a component that displays this problem.

To solve this problem, we need a model that satisfies properties (i)-(iii) for type declarations, while at the same time giving us an efficient way to refer to the elements in the structure. We will describe the proposed solution in Section 3.4.4. In the next section we introduce the PARTIAL_FUNCTION TEMPLATE math module that will be necessary to build the type declaration model.

3.4.3 PARTIAL_FUNCTION

The purpose of the PARTIAL_FUNCTION TEMPLATE (Figures 20-21) module is to provide a model for partial functions. It defines a new math subtype PARTIAL_FUNCTION, and six math operations. It is parameterized by two math types: DOMAIN_ITEM represents the domain of the partial function, and RANGE_ITEM is the range of the partial function. The type itself is modeled by a set of ordered pairs with the function property (expressed in the constraint). The math operations defined in this module have the following intuitive meaning:

- EMPTY_PARTIAL_FUNCTION is a 0-ary function defined as the empty set, i.e., the partial function that is undefined at all points in its domain.
- DEFINED_IN – A Boolean function that gives true if and only if the given PARTIAL_FUNCTION is defined at the given domain value.
**mathematics** \textsc{PARTIAL\_FUNCTION\_TEMPLATE}

**context**

**parametric context**

**math type** \textsc{DOMAIN\_ITEM}

**math type** \textsc{RANGE\_ITEM}

**interface**

**math subtype** \textsc{PARTIAL\_FUNCTION} \textit{is finite set of}

\begin{align*}
&d: \textsc{DOMAIN\_ITEM} \\
r: \textsc{RANGE\_ITEM}
\end{align*}

**exemplar** \( m \)

**constraint**

\textit{for all} \( d: \textsc{DOMAIN\_ITEM}, r_1, r_2: \textsc{RANGE\_ITEM} \)
\textit{where} \(( (d, r_1) \text{ is in } m \text{ and } (d, r_2) \text{ is in } m)\)
\((r_1 = r_2)\)

**math operation** \textsc{EMPTY\_PARTIAL\_FUNCTION} : \textsc{PARTIAL\_FUNCTION}

**explicit definition**

\{
\}

**math operation** \textsc{DEFINED\_IN} ( \( m: \textsc{PARTIAL\_FUNCTION} \)
\( d: \textsc{DOMAIN\_ITEM} \)) : boolean

**explicit definition**

\textit{there exists} \( r: \textsc{RANGE\_ITEM} ((d, r) \text{ is in } m)\)

**math operation** \textsc{DIFFER\_ONLY\_AT} ( \( m_1: \textsc{PARTIAL\_FUNCTION} \)
\( m_2: \textsc{PARTIAL\_FUNCTION} \)
\( \text{d\_items: set of } \textsc{DOMAIN\_ITEM} \)) : boolean

**explicit definition**

\textit{for all} \( \text{dr\_pair: } (d: \textsc{DOMAIN\_ITEM}, r: \textsc{RANGE\_ITEM}) \)
\textit{where} \((\text{dr\_pair.d is not in d\_items})\)
\((\text{dr\_pair is in } m_1 \text{ iff } \text{dr\_pair is in } m_2)\)

---

Figure 20: The \textsc{PARTIAL\_FUNCTION\_TEMPLATE} Module
math operation DOMAIN (  
    m : PARTIAL_FUNCTION  
  ) : set of DOMAIN_ITEM  
explicit definition  
{d: DOMAIN_ITEM  
    where (there exists r: RANGE_ITEM ((d, r) is in m)) (d)}

math operation RANGE (  
    m : PARTIAL_FUNCTION  
  ) : set of RANGE_ITEM  
explicit definition  
{r: RANGE_ITEM  
    where (there exists d: DOMAIN_ITEM ((d, r) is in m)) (r)}

math operation APPLY (  
    m : PARTIAL_FUNCTION  
    d : DOMAIN_ITEM  
  ) : RANGE_ITEM  
implicit definition  
if DEFINED_IN (m, d)  
then there exists r: RANGE_ITEM  
    ((d, r) is in m and  
    APPLY (m, d) = r)  
else APPLY (m, d) = RANGE_ITEM.base_point

end PARTIAL_FUNCTION_TEMPLATE

Figure 21: The PARTIAL_FUNCTION_TEMPLATE Module (continued)
• **DIFFER_ONLY_AT** – A Boolean function that is true if and only if the two given
PARTIAL_FUNCTIONS differ only at domain values in the given set of DOMAINS (i.e., for all values in the domain of the two functions, except possibly those in
d_items, the two functions are either both undefined, or they are both defined and
have the same value).

• **DOMAIN** – Gives the set of DOMAINS on which the given PARTIAL_FUNCTION
is defined.

• **RANGE** – Gives the set of RANGE_ITEMS that are values of the given
PARTIAL_FUNCTION for some value in the domain where the function is defined.

• **APPLY** – Gives the result of applying the given PARTIAL_FUNCTION to the given
DOMAIN_ITEM. If the function is not defined at the given DOMAIN_ITEM, the value of
APPLY is RANGE_ITEM.base_point.

### 3.4.4 NESTED_MAP_THEORY

Now let’s return to the original goal of modeling type declarations in such a way that
the required structural properties are satisfied, and so that it is also possible to efficiently
access the elements making up the modeled structure.

As we have seen earlier, a TREE is formally defined as the composition of a label of
some kind with a string of TREES. A NESTED_SET instead is the composition of a label
with a set of NESTED_SET. The basic idea here is to modify the NESTED_SET model by
replacing the set used to group substructures with a partial function from an identifier type
(ID) to a substructure of the kind we are trying to define.
Figures 22-23 show the resulting \texttt{NESTED\_MAP\_THEORY} module. This module is very similar to the \texttt{TREE\_THEORY\_TEMPLATE} and \texttt{NESTED\_SET\_THEORY\_TEMPLATE} modules. It defines two new math types, \texttt{NESTED\_MAP} and \texttt{NESTED\_MAP\_SET}, three math operations \texttt{EMPTY\_NESTED\_MAP}, \texttt{EMPTY\_NESTED\_MAP\_SET}, and \texttt{COMPOSE}, and three axioms. The main difference is that this module has two parameters, \texttt{ID} and \texttt{LABEL}, instead of one like the other two modules.

It is not straightforward to visualize the structures represented by the two types (\texttt{NESTED\_MAP} and \texttt{NESTED\_MAP\_SET}) by only using the given axioms. Let's try to build the right intuitions one step at a time. First, let's consider \texttt{NESTED\_MAP\_SET}. This is a partial function from \texttt{ID} to \texttt{NESTED\_MAP}. Since we don't know yet what a \texttt{NESTED\_MAP} is going to look like, we will draw it as a cloud-shaped symbol. Figure 24 shows what we know so far. Now, from the axioms, we know that other than the \texttt{EMPTY\_NESTED\_MAP}, all other \texttt{NESTED\_MAPs} must be the result of \texttt{COMPOSE\_ing} a \texttt{LABEL} and a \texttt{NESTED\_MAP\_SET}. Figure 25 shows the result of this composition using the picture from Figure 24 to represent the \texttt{NESTED\_MAP\_SET}. But now we can replace the cloud-shaped symbols in this diagram with (smaller) copies of the same structure, since these are supposed to be \texttt{NESTED\_MAPs}. So we can think of a \texttt{NESTED\_MAP} as a structure like the one depicted in Figure 26. By just "ignoring" the label at the \texttt{root} of this diagram we get Figure 27 which is one way to picture a \texttt{NESTED\_MAP\_SET}.

It is useful to think of a \texttt{NESTED\_MAP\_SET} object as \texttt{set of \texttt{NESTED\_SET} object} where each \texttt{NESTED\_SET} in the \texttt{set of \texttt{NESTED\_SET}} has an \texttt{ID} associated to it that can be used to easily refer to \texttt{NESTED\_SET}. In other words, in most cases we want to picture a \texttt{NESTED\_MAP\_SET} as depicted in Figure 18 (without the root label, of course), and think
mathematics NESTED_MAP_THEORY_Template

colorbox

global context

mathematics PARTIAL_FUNCTION_Template

parametric context

math type ID

math type LABEL

interface

math type NESTED_MAP

math facility PARTIAL_FUNCTION_FACILITY is

PARTIAL_FUNCTION_Template (ID, NESTED_MAP)

renaming

PARTIAL_FUNCTION_FACILITY.PARTIAL_FUNCTION as NESTED_MAP_SET
PARTIAL_FUNCTION_FACILITY.EMPTY_PARTIAL_FUNCTION as

EMPTY_NESTED_MAP_SET

math operation EMPTY_NESTED_MAP: NESTED_MAP

math operation COMPOSE ( root: LABEL, children: NESTED_MAP_SET ) : NESTED_MAP

axiom 1 is

for all root: LABEL, children: NESTED_MAP_SET

(COMPOSE (root, children) /= EMPTY_NESTED_MAP)

axiom 2 is

for all root1, root2: LABEL, children1, children2: NESTED_MAP_SET

(if COMPOSE (root1, children1) =

COMPOSE (root2, children2)

then root1 = root2 and children1 = children2)

Figure 22: The NESTED_MAP_THEORY Module
axiom induction is
for all s: set of NESTED_MAP
   (if EMPTY_NESTED_MAP is in s and
    for all root: LABEL, children: NESTED_MAP_SET
       (if for all m: NESTED_MAP
          where (m is in RANGE (children))
             (m is in s)
          then COMPOSE (root, children) is in s)
    then s = universal_set)
end NESTED_MAPS_THEORY_TEMPLATE
Figure 24: Step 1 - A NESTED_MAP_SET Object
Figure 25: Step 2 - A NESTED_MAP Object
Figure 26: Step 3 - The final NESTED_MAP Object
Figure 27: Step 4 - The final NESTED_MAP_SET Object
of the ID associated to each of the elements as the unique name or identifier for the
element. That's why we chose the name NESTED_MAP_SET in spite of the fact that this
type is really a PARTIAL_FUNCTION.

In Appendix B we define some extensions to the model defined by the
NESTED_MAP_THEORY_TEMPLATE, and use them to specify several program components
(RE SOLVE concepts) that can be used in implementing conceptual program editors as
well as any other language-based tools.

For now, we need to point out that the NESTED_MAP type satisfies all the required
properties: it is a recursive structure, with an arbitrary number of substructures of the same
kind, where the order of the substructures does not matter. It is also possible to specify
program components that allow the manipulation of objects modeled by the NESTED_MAP
(or NESTED_MAP_SET) type in an efficient and convenient way, without the need to
duplicate the potentially large LABELS (though the hopefully small IDS will need to be
copied). See Appendix B for some examples of such program components.

Next we use the NESTED_MAP type to model type declarations.

3.4.5 TYPE_DECLARATION

This section describes two math modules: the TYPE_DECLARATION_TEMPLATE
module, which gives the formal definition for type declarations, and the
TYPE_DECLARATION_MACHINERY module, that introduces several useful math
operations for type declarations. Throughout this section, the reader may want to refer to
Figure 28 for examples of some of the mathematical entities involved in the model. The
figure depicts, in graphical form, the mathematical description of the type declaration in
Figure 17.
Figure 28: Example of a TYPEDECLARATION Object
The `TYPE_DECLARATION_TEMPLATE` module (Figures 29-30), like every other RESOLVE math module, has a context section and an interface section. Let's look at the parametric context section. `TYPE_DECLARATION_TEMPLATE` has five parameters. `ID1` is the type of the domain of the `NESTED_MAP` type that will be used to model type declarations. `ID2` is the type representing (built-in and newly declared) type constructors. This will be the type of the labels in the `TYPE_EXPRESSIONS` that will serve as the bodies of the type declarations. The distinguished value `NIL` of type `ID2`, and the `ARITY` math operation, together with the restriction at the end of the parametric context section, are needed by the last parameter, an instance of the `TYPE_EXPRESSION_TEMPLATE`. Of course, since type declarations contain type expressions as their bodies, the `TYPE_DECLARATION_TEMPLATE` needs to refer to the `TYPE_EXPRESSION_TEMPLATE` module.

The interface part of the module defines four new math subtypes, `TYPE_DECLARATION_HEADER`, `TYPE_DECLARATION_LABEL`, `TYPE_DECLARATION`, and `TYPE_DECLARATION_SET`. Let's consider them one at a time.

`TYPE_DECLARATION_HEADER` models the header of a type declaration. It is a 2-tuple with a field `tid` of type `ID2` that represents the unique identifier for the new type constructor (think of it as the name in a Pascal procedure). The second field `pp` is a string of `ID2` that represents the "parameter profile" of the type declaration. In other words, it keeps track of the number of arguments (the arity of the new type constructor), and gives them unique names that can be used in type expressions just like any other type constructors. For simplicity, all the arguments are type constructors of arity 0. The constraint imposed on the subtype ensures that all the `ID2s` in a `TYPE_DECLARATION_HEADER` are distinct.
mathematics TYPE DECLARATION TEMPLATE

context

global context

mathematics TYPE EXPRESSION TEMPLATE
mathematics NESTED MAP THEORY TEMPLATE
mathematics PARTIAL FUNCTION TEMPLATE

parametric context

math type ID1

math type ID2

math operation NIL : ID2

math operation ARITY (tid: ID2): integer

math facility TYPE EXPRESSION FACILITY is
    TYPE EXPRESSION TEMPLATE (ID2, NIL, ARITY)

restriction
    for all tid: ID2 (ARITY (tid) >= -1) and
    ARITY (NIL) = -1

interface

math subtype TYPE DECLARATION HEADER is
    
    tid: ID2
    pp: string of ID2

exemplar tdh

constraint
tdh. tid is not in elements (tdh.pp) and
|elements (tdh.pp)| = |tdh.pp|

math subtype TYPE DECLARATION LABEL is

    header: TYPE DECLARATION HEADER
    body: TYPE EXPRESSION

Figure 29: The TYPE DECLARATION TEMPLATE Module

87
math facility NESTED_MAP_THEORY_FACILITY is
  NESTED_MAP_THEORY_TEMPLATE (ID1, TYPE_DECLARATION_LABEL)

math facility TYPE_DECLARATION_SET_FACILITY is
  PARTIAL_FUNCTION_TEMPLATE (ID1, TYPE_DECLARATION)

renaming
  TYPE_DECLARATION_SET_FACILITY.PARTIAL_FUNCTION as
  TYPE_DECLARATION_SET
  TYPE_DECLARATION_SET_FACILITY.EMPTY_PARTIAL_FUNCTION as
  EMPTY_TYPE_DECLARATION_SET

math operation IS_TYPE_DECLARATION (nm: NESTED_MAP): boolean
  implicit definition
    IS_TYPE_DECLARATION (nm) =
    there exists tdl: TYPE_DECLARATION_LABEL,
      children: PARTIAL_FUNCTION
      (for all x: NESTED_MAP where (x is in RANGE (children))
        (IS_TYPE_DECLARATION (x)) and
        nm = COMPOSE (tdl, children))

math subtype TYPE_DECLARATION is NESTED_MAP
  exemplar td
  constraint
    IS_TYPE_DECLARATION (td)

end TYPE_DECLARATION_TEMPLATE

Figure 30: The TYPE_DECLARATION_TEMPLATE Module (continued)
The second math subtype defined in the module is TYPEDECLARATION_LABEL. This type will be used as the label type for the NESTED_MAP. It is a 2-tuple with a header field of type TYPEDECLARATION_HEADER, and a body field of type TYPE_EXPRESSION. No constraint is necessary here.

Finally, we are ready to define the TYPEDECLARATION math subtype. TYPEDECLARATION is NESTED_MAP with ID1 domain type and TYPEDECLARATION_LABEL label type. The constraint imposed by the IS_TYPEDECLARATION predicate is simply stating that any NESTED_MAP other than the EMPTY_NESTED_MAP (and all NESTED_MAPS containing somewhere an EMPTY_NESTED_MAP) is a valid TYPEDECLARATION.

Now that we have the TYPEDECLARATION subtype, we can define the TYPEDECLARATION_SET math subtype as a PARTIAL_FUNCTION from ID1 to TYPEDECLARATION. Note that to be consistent with the naming convention we chose in Section 3.4.4 (NESTED_MAPS_THEORY), and for the same reason of trying to convey our mental image of a group of TYPEDECLARATIONS as a set, we named a PARTIAL_FUNCTION TYPEDECLARATION_SET.

Given the types (i.e., the formal model), several other math operations related to type declarations will be needed. These are grouped in a separate module called TYPEDECLARATION_MACHINERY (Figures 31-35). Note that this module has all the parameters of the TYPEDECLARATION_TEMPLATE module plus two more, an instance of the TYPE_EXPRESSION_MACHINERY module and an instance of the TYPEDECLARATION_TEMPLATE module itself. Here are the math operations defined in this module:
mathematics TYPE_DECLARATION_MACHINERY

context

global context

mathematics TYPE_EXPRESSION_TEMPLATE
mathematics TYPE_EXPRESSION_MACHINERY
mathematics TYPE_DECLARATION_TEMPLATE

parametric context

math type ID1
math type ID2
math operation NIL : ID2
math operation ARITY (tid: ID2): integer

math facility TYPE_EXPRESSION_FACILITY is
  TYPE_EXPRESSION_TEMPLATE (ID2, NIL, ARITY)

math facility TYPE_EXPRESSION_ENHANCED_FACILITY is
  TYPE_EXPRESSION_MACHINERY (
    ID2, NIL, ARITY, TYPE_EXPRESSION_FACILITY
  )

math facility TYPE_DECLARATION_FACILITY is
  TYPE_DECLARATION_TEMPLATE (
    ID1, ID2, NIL, ARITY, TYPE_EXPRESSION_FACILITY
  )

restriction
  for all tid: ID2 (ARITY (tid) >= -1) and
  ARITY (NIL) = -1

interface

math operation INITIAL_TYPE_DECLARATION (  
  label: TYPE_DECLARATION_HEADER 
): TYPE_DECLARATION

explicit definition
  COMPOSE ((label, NIL_LEAF), EMPTY_TYPE_DECLARATION_SET)

Figure 31: The TYPE_DECLARATION_MACHINERY Module

90
math operation EXTRACT_LABEL (td: TYPE_DECLARATION) : TYPE_DECLARATION_LABEL

implicit definition

there exists children: TYPE_DECLARATION_SET
(td = COMPOSE (EXTRACT_LABEL (td), children))

math operation EXTRACT_CHILDREN (td: TYPE_DECLARATION) : TYPE_DECLARATION_SET

implicit definition

there exists l: TYPE_DECLARATION_LABEL
(td = COMPOSE (l, EXTRACT_CHILDREN (td)))

math operation EXTRACT_CHILDREN (tds: TYPE_DECLARATION_SET id: IDI) : TYPE_DECLARATION_SET

explicit definition

EXTRACT_CHILDREN (APPLY (tds, id))

math operation EXTRACT_HEADER (tds: TYPE_DECLARATION_SET id: IDI) : TYPE_DECLARATION_HEADER

explicit definition

EXTRACT_LABEL (APPLY (tds, id)).header

math operation EXTRACT_HEADER (tds: TYPE_DECLARATION_SET path: string of IDI) : TYPE_DECLARATION_HEADER

explicit definition

EXTRACT_LABEL (APPLY (EXTRACT_CURRENT (tds, all_but_last (path)), last (path))).header

Figure 32: The TYPE_DECLARATION_MACHINERY Module (continued)
math operation \texttt{EXTRACT\_BODY} ( \\
\hspace{1em} \textit{tds}: \texttt{TYPE\_DECLARATION\_SET} \\
\hspace{1em} \textit{path}: \texttt{string of IDI} \\
\hspace{1em} ): \texttt{TYPE\_EXPRESSION} \\
\textbf{explicit definition} \\
\texttt{EXTRACT\_LABEL} ( \\
\hspace{1em} \texttt{APPLY} ( \\
\hspace{2em} \texttt{EXTRACT\_CURRENT} \ (\textit{tds}, \texttt{all\_but\_last} \ (\textit{path})), \\
\hspace{2em} \texttt{last} \ (\textit{path})).\texttt{body} \\
\textbf{math operation \texttt{EXTRACT\_CURRENT} ( \\
\hspace{1em} \textit{tds}: \texttt{TYPE\_DECLARATION\_SET} \\
\hspace{1em} \textit{path}: \texttt{string of IDI} \\
\hspace{1em} ): \texttt{TYPE\_DECLARATION\_SET} \\
\textbf{implicit definition} \\
\hspace{1em} \textbf{if} \textit{path} = \texttt{empty\_string} \textbf{then} \\
\hspace{2em} \texttt{EXTRACT\_CURRENT} \ (\textit{tds}, \textit{path}) = \textit{tds} \\
\hspace{1em} \textbf{else} \\
\hspace{2em} \texttt{EXTRACT\_CURRENT} \ (\textit{tds}, \textit{path}) = \\
\hspace{2em} \texttt{EXTRACT\_CURRENT} \ (\texttt{EXTRACT\_CHILDREN} \ (\textit{tds}, \texttt{first} \ (\textit{path})), \\
\hspace{3em} \texttt{all\_but\_first} \ (\textit{path})) \\
\textbf{math operation \texttt{IS\_DEFINED} ( \\
\hspace{1em} \textit{tid}: \texttt{ID2} \\
\hspace{1em} \textit{tds}: \texttt{TYPE\_DECLARATION\_SET} \\
\hspace{1em} ): \texttt{boolean} \\
\textbf{implicit definition} \\
\hspace{1em} \textbf{if} \textbf{there exists} \ \textit{path}: \texttt{string of IDI}, \ \textit{id}: \texttt{IDI} \\
\hspace{2em} \texttt{(EXTRACT\_LABEL} ( \\
\hspace{3em} \texttt{APPLY} ( \\
\hspace{4em} \texttt{EXTRACT\_CURRENT} \ (\textit{tds}, \textit{path}), \\
\hspace{4em} \textit{id})).\texttt{header.tid} = \textit{tid} \ \textbf{or} \\
\hspace{4em} \textit{tid} \ \textbf{is in elements} ( \\
\hspace{5em} \texttt{EXTRACT\_LABEL} ( \\
\hspace{6em} \texttt{APPLY} ( \\
\hspace{7em} \texttt{EXTRACT\_CURRENT} \ (\textit{tds}, \textit{path}), \\
\hspace{7em} \textit{id})).\texttt{header.pp}) \\
\hspace{2em} \textbf{then} \texttt{IS\_DEFINED} \ (\textit{tid}, \textit{tds}) = \texttt{true} \\
\hspace{2em} \textbf{else} \texttt{IS\_DEFINED} \ (\textit{tid}, \textit{tds}) = \texttt{false} \\

Figure 33: The \texttt{TYPE\_DECLARATION\_MACHINERY} Module (continued)
math operation ARITY (  
  tid: ID2  
  tds: TYPE_DECLARATION_SET  
) : integer  

implicit definition  
if IS_DEFINED (tid, tds)  
then  
  there exists path: string of IDI, id: IDI,  
  tdh: TYPE_DECLARATION_HEADER  
  (tdh = EXTRACT_LABEL (  
    APPLY (  
      EXTRACT_CURRENT (tds, path), id)  
    .header  
    and  
    ((tdh.tid = tid and ARITY (tid, tds) = |tdh.pp|) or  
    (tid is in elements (tdh.pp) and  
    ARITY (tid, tds) = 0)))  
  )  
elsif  
  ARITY (tid, tds) = integer.base_point  

math operation DEPENDENCY_COUNT (  
  tds: TYPE_DECLARATION_SET  
  tid: ID2  
) : integer  

explicit definition  
OCCURS_COUNT (tds, tid) - OCCURS_COUNT (EXTRACT_TD (tds, tid))  

math operation EXTRACT_TD (  
  tds: TYPE_DECLARATION_SET  
  tid: ID2  
) : TYPE_DECLARATION  

implicit definition  
if IS_DEFINED (tid, tds)  
then  
  there exists path: string of IDI, id: IDI,  
  td: TYPE_DECLARATION  
  (td = APPLY (EXTRACT_CURRENT (tds, path), id) and  
  (EXTRACT_LABEL (td).header.tid = tid or  
  tid is in elements (EXTRACT_LABEL (td).header.pp)) and  
  EXTRACT_TD (tds, tid) = td))  
elsif  
  EXTRACT_TD (tds, tid) = TYPE_DECLARATION.base_point  

Figure 34: The TYPE_DECLARATION_MACHINERY Module (continued)
math operation OCCURS_COUNT ( 
    td: TYPE_DECLARATION 
    tid: ID2 
): integer 
explicit definition 
    OCCURS_COUNT (EXTRACT_LABEL (td).body, tid) + 
    OCCURS_COUNT (EXTRACT_CHILDREN (td), tid) 

math operation OCCURS_COUNT ( 
    tds: TYPE_DECLARATION_SET 
    tid: ID2 
): integer 
explicit definition 
    sum id: ID1 
    where (id is in DOMAIN (tds)) 
        (OCCURS_COUNT (APPLY (tds, id)), tid) 
end TYPE_DECLARATION_MACHINERY 

Figure 35: The TYPE_DECLARATION_MACHINERY Module (continued)
• **INITIAL_TYPE_DECLARATION** – It gives the **TYPE_DECLARATION** with root label made up of the given **TYPE_DECLARATION_HEADER** and a **NIL_LEAF** body, with no local type declarations (empty local scope).

• **EXTRACT_LABEL** – Given a **TYPE_DECLARATION**, it gives the **TYPE_DECLARATION_LABEL** of the root’s label.

• **EXTRACT_CHILDREN** – Given a **TYPE_DECLARATION**, it gives the **TYPE_DECLARATION_SET** made up of the children of the root node.

• **EXTRACT_CHILDREN** – Given a **TYPE_DECLARATION_SET** and an **IDI**, it gives the **TYPE_DECLARATION_SET** made up of the children of the root node of the **TYPE_DECLARATION** with the given **IDI**.

• **EXTRACT_HEADER** – Given a **TYPE_DECLARATION_SET** and an **IDI**, it gives the **TYPE_DECLARATION_HEADER** of the root’s label.

• **EXTRACT_HEADER** – Given a **TYPE_DECLARATION_SET** and a string of **IDI**, it gives the **TYPE_DECLARATION_HEADER** of the root’s label of the **TYPE_DECLARATION_SET** obtained by following the path defined by the given string of **IDI** through the given **TYPE_DECLARATION_SET**.

• **EXTRACT_BODY** – Given a **TYPE_DECLARATION_SET** and a string of **IDI**, it gives the **TYPE_EXPRESSION**, which is the body of the **TYPE_DECLARATION** obtained by following the path defined by the given string of **IDI** through the given **TYPE_DECLARATION_SET**.

• **EXTRACT_CURRENT** – Given a **TYPE_DECLARATION_SET** and a string of **IDI**, it gives the **TYPE_DECLARATION_SET** obtained by following the path defined by the given string of **IDI** through the given **TYPE_DECLARATION_SET**.

95
• **IS_DEFINED** – A Boolean function that is true if and only if there exists (possibly nested) in the given TYPE_DECLARATION_SET a TYPE_DECLARATION that defines the given ID2.

• **ARITY** – An integer-valued function that gives the arity of the given ID2 in the given TYPE_DECLARATION_SET, if there exists (possibly nested) in the given TYPE_DECLARATION_SET a TYPE_DECLARATION that defines the given ID2. Otherwise, it gives the value integer.base_point.

• **DEPENDENCY_COUNT** – An integer-valued function that gives the number of times the given ID2 occurs in the given TYPE_DECLARATION_SET but not in the TYPE_DECLARATION that defines the given ID2.

• **EXTRACT_TD** – Given a TYPE_DECLARATION_SET and an ID2, it gives the TYPE_DECLARATION that defines the given ID2, if such declaration exists. Otherwise it gives the value TYPE_DECLARATION.base_point.

• **OCCURS_COUNT** – Given a TYPE_DECLARATION and an ID2, it gives the total number of occurrences of the given ID2 in the body of the given TYPE_DECLARATION and in the bodies of all nested TYPE_DECLARATIONS (recursively down to the leaf TYPE_DECLARATIONS).

• **OCCURS_COUNT** – Given a TYPE_DECLARATION_SET and an ID2, it gives the total number of occurrences of the given ID2 in the bodies of the TYPE_DECLARATIONS in the given TYPE_DECLARATION_SET and in the bodies of all nested TYPE_DECLARATIONS (recursively down to the leaf TYPE_DECLARATIONS).

This concludes our description of type declarations and the related math model and definitions.
3.5 Other Modules

This section describes the other math modules and conceptual entities necessary to formally specify the type declaration editor. These include the idea of specifying a position in the hierarchies of TYPE_DECLARATION_SETS and TYPE_EXPRESSIONS, and a model of the context describing the available type constructors and their arity.

3.5.1 Position

Here we describe the modules defining the idea of position in a TYPE_DECLARATION_SET and in a TYPE_EXPRESSION. These concepts are necessary to be able to specify operations that act on TYPE_DECLARATION_SETs and TYPE_EXPRESSIONs. Module POSITION_TEMPLATE (Figures 36-38) defines the appropriate math subtypes, and module POSITION_MACHINERY (Figures 39-41) defines the necessary math operations.

The POSITION_TEMPLATE module is parameterized by instances of the TYPE_EXPRESSION and TYPE_DECLARATION modules described earlier, and by the types (ID1 and ID2) and operations (NIL and ARITY) these modules expect (Figure 36).

Let's consider the model for position in a TYPE_EXPRESSION. We can uniquely specify a position in a TYPE_EXPRESSION by providing a string of integers (a path) that, starting from the root of the TYPE_EXPRESSION, tells us which child to choose at each successive level down the structure until we get to the intended TYPE_EXPRESSION. So an empty string of integers would refer to the position at the root of the TYPE_EXPRESSION. A string of the form <n>, where an is an integer in the appropriate range, would refer to the n-th child of the root. A string of the form <n1, n2>, where n1
mathematics POSITION_TEMPLATE

context

global context

mathematics TYPE_EXPRESSION_TEMPLATE
mathematics TYPE_EXPRESSION_MACHINERY
mathematics TYPE_DECLARATION_TEMPLATE
mathematics TYPE_DECLARATION_MACHINERY

parametric context

math type ID1
math type ID2
math operation NIL : ID2
math operation ARITY (tid: ID2): integer

math facility TYPE_EXPRESSION_FACILITY is
TYPE_EXPRESSION_TEMPLATE (ID2, NIL, ARITY)

math facility TYPE_EXPRESSION_ENHANCED_FACILITY is
TYPE_EXPRESSION_MACHINERY (ID2, NIL, ARITY,
TYPE_EXPRESSION_FACILITY)

math facility TYPE_DECLARATION_FACILITY is
TYPE_DECLARATION_TEMPLATE (ID1, ID2, NIL, ARITY,
TYPE_EXPRESSION_FACILITY)

math facility TYPE_DECLARATION_ENHANCED_FACILITY is
TYPE_DECLARATION_TEMPLATE (ID1, ID2, NIL, ARITY,
TYPE_EXPRESSION_FACILITY,
TYPE_EXPRESSION_ENHANCED_FACILITY,
TYPE_DECLARATION_FACILITY)

restriction
for all tid: TYPE_ID (ARITY (tid) >= -1) and
ARITY (NIL) = -1

Figure 36: The POSITION_TEMPLATE Module
interface

math subtype TE_POSITION is string of integer
  exemplar tep
  constraint
    for all i: integer
      where (i is in elements {tep})
      (i >= 0)

math operation INITIAL_TE_POSITION : TE_POSITION
  explicit definition
  empty_string

math subtype TDS_POSITION is
  path: string of ID
  body_is_open: boolean
  body_position: TE_SELECTION
  
  exemplar tdsp
  constraint
    if tdsp.body_is_open
    then tdsp.path /= empty_string
    else tdsp.body_position = INITIAL_TE_POSITION

math operation INITIAL_TDS_POSITION : TDS_POSITION
  explicit definition
  (empty_string, false, INITIAL_TE_POSITION)

math subtype TYPE_DECLARATION_SET_WITH_POSITION is
  tds: TYPE_DECLARATION_SET
  pos: TDS_POSITION

  exemplar tdswp
  constraint
  IS_VALID_TDS_POSITION (tdswp.posion, tdswp.tds)

Figure 37: The POSITION_TEMPLATE Module (continued)

99
math operation IS_VALID_TDS_POSITION ( 
    pos: TDS_POSITION  
    tds: TYPE_DECLARATION_SET  
) : boolean

implicit definition
if pos.path /= empty_string
then
  if DEFINED_IN (tds, first (pos.path))
  then
    if pos.body_is_open and |pos.path| = 1
then
    IS_VALID_TDS_POSITION (pos, tds) = 
    IS_VALID_TE_POSITION (  
        pos.body_position,  
        EXTRACT_BODY (tds, first (pos.path)))
else
    IS_VALID_TDS_POSITION (pos, tds) = 
    IS_VALID_TDS_POSITION (  
        all_but_first (pos.path), pos.body_is_open,  
        pos.body_position),  
        EXTRACT_CHILDREN (tds, first (pos.path)))
else
    IS_VALID_TDS_POSITION (pos, tds) = false
else
    IS_VALID_TDS_POSITION (pos, tds) = true

math operation IS_VALID_TE_POSITION ( 
    pos: TE_POSITION  
    te: TYPE_EXPRESSION  
) : boolean

implicit definition
if pos /= empty_string
then
  if first (pos) < |EXTRACT_CHILDREN (te)|
then
  IS_VALID_TE_POSITION (pos, te) = 
  IS_VALID_TE_POSITION (  
      all_but_first (pos),  
      EXTRACT_CHILD (te, first (sel.path)))
else
  IS_VALID_TE_POSITION (pos, te) = false
else
  IS_VALID_TE_POSITION (pos, te) = true

end POSITION_TEMPLATE
and $n_2$ are integers in the appropriate ranges, would refer to the $n_2$-th child of the $n_1$-th child of the root. And so on for longer strings of integers. In Figure 37 we formalize this intuitive definition with the math subtype $\text{TE\_POSITION}$.

To model a position in a $\text{TYPE\_DECLARATION\_SET}$ we need to keep track of where in the hierarchy we are. This is done with a string of IDIs. But that is not enough. Since the editor will need access also to the bodies of the type declarations, we need to include in the model a $\text{TE\_POSITION}$ (i.e., a position in the current $\text{TYPE\_DECLARATION}$'s body), and also a Boolean flag that tells us whether the position is currently in a $\text{TYPE\_DECLARATION\_SET}$ or in a $\text{TYPE\_EXPRESSION}$ (a $\text{TYPE\_DECLARATION}$'s body). The math subtype $\text{TDS\_POSITION}$ (Figure 37) formally describes this notion. Note that depending on the value of the Boolean flag $\text{body\_is\_open}$, the last IDI in the $\text{path}$ field has one of two different meanings. If $\text{body\_is\_open}$ is false, the last IDI in $\text{path}$ is interpreted just as one more step to follow in the path to get to the current $\text{TYPE\_DECLARATION\_SET}$. If $\text{body\_is\_open}$ is false, however, the last IDI in $\text{path}$ specifies the $\text{TYPE\_DECLARATION}$ in the current $\text{TYPE\_DECLARATION\_SET}$ whose body is currently "open", i.e., the $\text{TYPE\_EXPRESSION}$ in which the position is currently located. This should explain the meaning of the constraint on the $\text{TDS\_POSITION}$ subtype: if $\text{body\_is\_open}$ is true, the $\text{path}$ cannot be empty. It must contain at least one IDI that determines which $\text{TYPE\_DECLARATION}$'s body we are currently in.

Now that we have a model for the position in $\text{TYPE\_DECLARATION\_SETS}$ and $\text{TYPE\_EXPRESSIONS}$, we can define one more math subtype that connects a $\text{TDS\_POSITION}$ and a $\text{TYPE\_DECLARATION\_SET}$. We call this subtype

---

7. We could define the position in a $\text{TYPE\_DECLARATION}$ (instead of a $\text{TYPE\_DECLARATION\_SET}$). But since we will use a set of type declarations as the model of the main data structure in the type declaration editor, we need the position to refer to $\text{TYPE\_DECLARATION\_SET}$. 

101
TYPE_DECLARATION_SET_WITH_POSITION (Figure 37), and we impose on it a constraint that ensures that the TDS_POSITION field is a valid position in the TYPE_DECLARATION_SET field. This is formally expressed by the two predicates IS_VALID_TDS_POSITION and IS_VALID_TE_POSITION (Figure 38). A TE_POSITION is valid if each integer in the string is in the appropriate range for the given TYPE_EXPRESSION (i.e., between 0 and the number of subexpressions at the corresponding level in the TYPE_EXPRESSION). The definition of a valid TDS_POSITION has two distinct cases depending on whether the body_is_open field is true or false. If it is false, then the TDS_POSITION is valid if each ID1 in the path field is defined in the TYPE_DECLARATION_SET at the corresponding level in the given TYPE_DECLARATION_SET. If the body_is_open field is true, then the TDS_POSITION is valid if in addition to the previous condition, the body_position field is a valid TE_POSITION for the body of the TYPE_DECLARATION specified by the path field.

The TYPE_DECLARATION_SET_WITH_POSITION subtype so defined is going to be the model of the main data structure in the type declaration editor described in Section 3.6.

Now let's look at the POSITION_MACHINERY module (Figures 39-41). It has all the parameters of the POSITION_TEMPLATE module plus an instance of POSITION_TEMPLATE itself. Here is an informal description of the operations defined in this module:

- **EXTRACT_CURRENT** — Given a TYPE_DECLARATION_SET_WITH_POSITION tdswp, if the current position is in the TYPE_DECLARATION_SET hierarchy (tdswp.pos.body_is_open is false), it gives the current TYPE_DECLARATION_SET, i.e., the TYPE_DECLARATION_SET obtained by
mathematics POSITION_MACHINERY

context

global context

mathematics TYPE_EXPRESSION_TEMPLATE
mathematics TYPE_EXPRESSION_MACHINERY
mathematics TYPE_DECLARATION_TEMPLATE
mathematics TYPE_DECLARATION_MACHINERY
mathematics POSITION_TEMPLATE

parametric context

math type ID1

math type ID2

math operation NIL : ID2

math operation ARITY (tid: ID2): integer

math facility TYPE_EXPRESSION_FACILITY is
  TYPE_EXPRESSION_TEMPLATE (ID2, NIL, ARITY)

math facility TYPE_EXPRESSION_ENHANCED_FACILITY is
  TYPE_EXPRESSION_MACHINERY (ID2, NIL, ARITY,
  TYPE_EXPRESSION_FACILITY)

math facility TYPE_DECLARATION_FACILITY is
  TYPE_DECLARATION_TEMPLATE (ID1, ID2, NIL, ARITY,
  TYPE_EXPRESSION_FACILITY)

math facility TYPE_DECLARATION_ENHANCED_FACILITY is
  TYPE_DECLARATION_TEMPLATE (ID1, ID2, NIL, ARITY,
  TYPE_EXPRESSION_FACILITY,
  TYPE_EXPRESSION_ENHANCED_FACILITY,
  TYPE_DECLARATION_FACILITY)

math facility POSITION_FACILITY is
  POSITION_TEMPLATE (ID1, ID2, NIL, ARITY,
  TYPE_EXPRESSION_FACILITY,
  TYPE_EXPRESSION_ENHANCED_FACILITY,
  TYPE_DECLARATION_FACILITY,
  TYPE_DECLARATION_ENHANCED_FACILITY)

Figure 39: The POSITION_MACHINERY Module
restriction

for all tid: TYPE_ID (ARITY (tid) >= -1) and
ARITY (NIL) = -1

**math operation** EXTRACT_CURRENT (tdswp: TYPE_DECLARATION_SET_WITH_POSITION): TYPE_DECLARATION_SET

**implicit definition**
if not tdswp.pos.body_is_open
then
EXTRACT_CURRENT (tdswp) =
EXTRACT_CURRENT (tdswp.tds, tdswp.pos.path)
else
EXTRACT_CURRENT (tdswp) = TYPE_DECLARATION_SET.base_point

**math operation** EXTRACT_CURRENT (tdswp: TYPE_DECLARATION_SET_WITH_POSITION): TYPE_EXPRESSION

**implicit definition**
if tdswp.pos.body_is_open
then
EXTRACT_CURRENT (tdswp) =
EXTRACT_SUBEXPRESSION (EXTRACT_BODY (tdswp.tds, tdswp.pos.path),
   tdswp.pos.body_position)
else
EXTRACT_CURRENT (tdswp) = TYPE_EXPRESSION.base_point

**math operation** EXTRACT_TID (tdswp: TYPE_DECLARATION_SET_WITH_POSITION): ID2

**explicit definition**
EXTRACT_LABEL (EXTRACT_CURRENT (tdswp))

**math operation** EXTRACT_CHILDREN (tdswp: TYPE_DECLARATION_SET_WITH_POSITION): TYPE_EXPRESSION_STRING

**explicit definition**
EXTRACT_CHILDREN (EXTRACT_CURRENT (tdswp))

---

Figure 40: The POSITION_MACHINERY Module (continued)
math operation DIFFER_ONLY_AT_CURRENT {
    tdswpl: TYPE_DECLARATION_SET_WITH_POSITION
    tdswp2: TYPE_DECLARATION_SET_WITH_POSITION
}: boolean

implicit definition
if tdswpl.pos = tdswp2.pos
then
    if tdswpl.pos.path = empty_string
    then DIFFER_ONLY_AT_CURRENT (tdswpl, tdswp2) = true
    else if |tdswpl.pos.path| = 1 and tdswpl.pos.body_is_open
    then
        DIFFER_ONLY_AT_CURRENT (tdswpl, tdswp2) =
        (tdswpl.tds - {(first (tdswpl.pos.path),
            APPLY (first (tdswpl.pos.path)))} =
         tdswp2.tds - {(first (tdswp2.pos.path),
            APPLY (first (tdswp2.pos.path)))} and
         EXTRACT_CHILDREN (tdswpl.tds, first (tdswpl.pos.path)) =
         EXTRACT_CHILDREN (tdswp2.tds,
             first (tdswp2.pos.path)) and
         EXTRACT_LABEL (APPLY (first (tdswpl.pos.path))).header =
         EXTRACT_LABEL (APPLY (first (tdswp2.pos.path))).header and
         DIFFER_ONLY_AT_CURRENT (EXTRACT_LABEL (APPLY (first (tdswpl.pos.path))).body,
             EXTRACT_LABEL (APPLY (first (tdswp2.pos.path))).body,
             tdswpl.pos.body_position))
    else
        DIFFER_ONLY_AT_CURRENT (tdswpl, tdswp2) =
        (tdswpl.tds - {(first (tdswpl.pos.path),
            APPLY (first (tdswpl.pos.path)))} =
         tdswp2.tds - {(first (tdswp2.pos.path),
            APPLY (first (tdswp2.pos.path)))} and
         EXTRACT_LABEL (APPLY (first (tdswpl.pos.path))) =
         EXTRACT_LABEL (APPLY (first (tdswp2.pos.path))) and
         DIFFER_ONLY_AT_CURRENT (
             (EXTRACT_CHILDREN (tdswpl.tds, first (tdswpl.pos.path)),
                 all_but_first (tdswpl.pos.path),
                 tdswpl.pos.body_is_open,
                 tdswpl.pos.body_position),
             (EXTRACT_CHILDREN (tdswp2.tds, first (tdswp2.pos.path)),
                 all_but_first (tdswp2.pos.path),
                 tdswp2.pos.body_is_open,
                 tdswp2.pos.body_position))))
    else DIFFER_ONLY_AT_CURRENT (tdswpl, tdswp2) = false

end POSITION_MACHINERY

Figure 41: The POSITION_MACHINERY Module (continued)
following the TDS_POSITION through the given
TYPE_DECLARATION_SET_WITH_POSITION. Otherwise, it gives
TYPE_DECLARATION_SET.base_point.

• **EXTRACT_CURRENT** – Given a TYPE_DECLARATION_SET_WITH_POSITION tdswp,
if the current position is in a TYPE_EXPRESSION hierarchy
(tdswp.pos.body_is_open is true), it gives the current TYPE_EXPRESSION, i.e.,
the TYPE_EXPRESSION obtained by following the TDS_POSITION through the given
TYPE_DECLARATION_SET_WITH_POSITION. Otherwise, it gives
TYPE_EXPRESSION.base_point.

• **EXTRACT_TID** – Given a TYPE_DECLARATION_SET_WITH_POSITION, it gives the
ID2 label of the current TYPE_EXPRESSION.

• **EXTRACT_CHILDREN** – Given a TYPE_DECLARATION_SET_WITH_POSITION, it
gives the TYPE_EXPRESSION_STRING made up of the children of the current
TYPE_EXPRESSION.

• **DIFFER_ONLY_AT_CURRENT** – A Boolean function that, given two
TYPE_DECLARATION_SET_WITH_POSITIONs, is true if and only if the two
TYPE_DECLARATION_SET_WITH_POSITIONs differ only at the structure
(TYPE_DECLARATION_SET or TYPE_EXPRESSION) determined by the current
positions in the TYPE_DECLARATION_SET_WITH_POSITIONs.

This concludes our description of the concepts of position in TYPE_EXPRESSIONs and
TYPE_DECLARATION_SETs and the related math models and definitions.
3.5.2 Type Context

As we explained in Chapter 2 (Section 2.3.3), the context is the set of all the available entities that are either built-in or user-declared, i.e., those entities that can be used at a certain point in a program to build a new piece of the program. In our simplified conceptual editor, the only entities that can be declared and used are type constructors. So the context (*type context*) is going to be fairly simple.

We want the context to hold all the information necessary to use an entity. In the case of type constructors, the only piece of information necessary to use a type constructor, in addition to its identifier, is its arity. The model we choose for the type context is a partial function from the space of type constructor identifiers to valid arities, i.e., integers greater than or equal to -1. The *type_context_template* module (Figure 42) formally defines the two math subtypes, *valid_arity* and *type_context*. It is parameterized by the math type *type_id* which represents the possible type constructor identifiers.

Although the model of the type context is conceptually simple, formally expressing the available context at some place in a *type_declaration* or *type_declaration_set* takes more work. In the *type_context_machinery* module (Figures 45-48) we provide the mathematical machinery necessary to express the idea of entity "valid with respect to the context". This is necessary to be able to constrain the models of *type_expression* and *type_declaration_set* to only allow valid, meaningful values (in the sense of Sections 2.3.1 and 2.3.2). Note that, of course, only a new *type_declaration* contributes to the context by defining a new type constructor and possibly introducing some others in its argument list. *Type_expressions* only use the available type constructors, they don't add to the context.
mathematics TYPE_CONTEXT_TEMPLATE

context

  global context

    mathematics PARTIAL_FUNCTION TEMPLATE

parametric context

  math type TYPE_ID

interface

  math subtype VALID_ARITY is integer
    exemplar n
    constraint
      n >= -1

math facility TYPE_ENVIRONMENT_FACILITY is
  PARTIAL_FUNCTION TEMPLATE (TYPE_ID, VALID_ARITY)

renaming
  TYPE_ENVIRONMENT_FACILITY.PARTIAL_FUNCTION as TYPE_CONTEXT
  TYPE_ENVIRONMENT_FACILITY.EMPTY_PARTIAL_FUNCTION as
    EMPTY_TYPE_CONTEXT

end TYPE_CONTEXT TEMPLATE

Figure 42: The TYPE_CONTEXT TEMPLATE Module
As we mentioned before (Section 3.4.1), the scope rules for type declarations are similar to the scope rules for nested Pascal procedures with the only difference that the order of the local declarations does not matter. This means that the context can be built by descending the TYPE_DECLARATION_SET hierarchy and adding to the current context any newly declared type constructors.

Figure 43 shows the type context at different places in the sample TYPE_DECLARATION introduced in Figure 17. Due to space constraints we have omitted the NIL type constructor (with arity -1) from the type contexts in the figure. In the bottom left corner is the built-in context. For each of the three TYPE_DECLARATIONS in the picture we show the corresponding type context.

Figure 44 shows how the context is built (the reader may want to refer back to Figure 43 to see the resulting contexts). At the top of the hierarchy, the context available is the built-in context (INITIAL_TYPE_CONTEXT). To obtain the context available in the body of the top-level TYPE_DECLARATION (Example), we add to the built-in context the Example type constructor, the parameter profile context (PP_CONTEXT) made up of the two formal parameters Item1 and Item2, and the nested context (CURRENT_CONTEXT) made up of the two locally declared type constructors Point3D and Segment3D.

To build the contexts available in the bodies of the two nested TYPE_DECLARATIONS we add the appropriate type constructors to the context of the parent. In this case, since neither of the two local declarations contains any other nested declarations, the only type constructors that get added are those from the parameter profile of the corresponding TYPE_DECLARATION. Both nested TYPE_DECLARATIONS have one formal parameter called Item in the example. In Figures 43-44 we have used fully qualified names for all identifiers to distinguish different instances of the same name (e.g., a.Item and b.Item).
Figure 43: Example of Type Declaration and the Corresponding Type Context
Figure 44: How the Type Context is Built
In the formal model of type context and the related math machinery we don't use fully qualified identifiers since we constrain well-formed TYPE_DECLARATION_SETS to only define type constructor identifiers that are unique throughout the structure.

We can now describe the TYPE_CONTEXT_MACHINERY module (Figures 45-48). The module is parameterized by instances of TYPE_EXPRESSION_TEMPLATE, TYPE_EXPRESSION_MACHINERY, TYPE_DECLARATION_TEMPLATE, TYPE_DECLARATION_MACHINERY, and TYPE_CONTEXT_TEMPLATE, plus the parameters needed by these modules. In addition, the module is parameterized by a constant, INITIAL_TYPE_CONTEXT, of type TYPE_CONTEXT that is meant to represent the built-in context (Figure 45).

The math operations defined in the TYPE_CONTEXT_MACHINERY module are the following:

- **IS_WELL_FORMED_WRT_CONTEXT** – This Boolean function is true if and only if the given TYPE_EXPRESSION is valid with respect to the given TYPE_CONTEXT. This is true if and only if:
  1. The given TYPE_EXPRESSION's root label is defined in the given context with the appropriate arity, and
  2. All its children TYPE_EXPRESSIONS are also valid with respect to the given context.

- **IS_WELL_FORMED_WRT_CONTEXT** – This Boolean function is true if and only if the given TYPE_EXPRESSION_STRING is valid with respect to the given TYPE_CONTEXT. This is true if and only if each TYPE_EXPRESSION in the given TYPE_EXPRESSION_STRING is valid with respect to the given context.
context

global context

mathematics TYPE_EXPRESSION_TEMPLATE
mathematics TYPE_EXPRESSION_MACHINERY
mathematics TYPE_DECLARATION_TEMPLATE
mathematics TYPE_DECLARATION_MACHINERY
mathematics TYPE_CONTEXT_TEMPLATE

parametric context

math type ID1

math type ID2

math operation NIL : ID2

math operation ARITY (tid: ID2): integer

math facility TYPE_EXPRESSION_FACILITY is
  TYPE_EXPRESSION_TEMPLATE (ID2, NIL, ARITY)

math facility TYPE_EXPRESSION_ENHANCED_FACILITY is
  TYPE_EXPRESSION_MACHINERY (ID2, NIL, ARITY, TYPE_EXPRESSION_FACILITY)

math facility TYPE_DECLARATION_FACILITY is
  TYPE_DECLARATION_TEMPLATE {
    ID1, ID2, NIL, ARITY, TYPE_EXPRESSION_FACILITY
  }

math facility TYPE_DECLARATION_ENHANCED_FACILITY is
  TYPE_DECLARATION_MACHINERY (ID1, ID2, NIL, ARITY, TYPE_EXPRESSION_FACILITY, TYPE_EXPRESSION_ENHANCED_FACILITY, TYPE_DECLARATION_FACILITY)

math facility TYPE_CONTEXT_FACILITY is
  TYPE_CONTEXT_TEMPLATE (ID2)

math operation INITIAL_TYPE_CONTEXT : TYPE_CONTEXT

Figure 45: The TYPE_CONTEXT_MACHINERY Module
restriction
  for all tid: ID2 (ARITY (tid) >= -1) and
  ARITY (NIL) = -1

interface

math operation IS_WELL_FORMED_WRT_CONTEXT (  
    te: TYPE_EXPRESSION
    tc: TYPE_CONTEXT
  ): boolean

explicit definition
  ((EXTRACT_LABEL (te), ARITY (EXTRACT_LABEL (te)))) is in tc and
  IS_WELL_FORMED_WRT_CONTEXT (EXTRACT_CHILDREN (te), tc))

math operation IS_WELL_FORMED_WRT_CONTEXT (  
    tes: TYPE_EXPRESSION_STRING
    tc: TYPE_CONTEXT
  ): boolean

explicit definition
  for all te: TYPE_EXPRESSION
    where (te is in elements (tes))
    (IS_WELL_FORMED_WRT_CONTEXT (te, tc))

math operation IS_WELL_FORMED_WRT_CONTEXT (  
    td: TYPE_DECLARATION
    tc: TYPE_CONTEXT
  ): boolean

explicit definition
  (IS_WELL_FORMED_WRT_CONTEXT (  
    EXTRACT_LABEL (td).body,
    tc union
      PP_CONTEXT (EXTRACT_LABEL (td).header.pp) union
      CURRENT_CONTEXT (EXTRACT_CHILDREN (td))) and
  IS_WELL_FORMED_WRT_CONTEXT (  
    EXTRACT_CHILDREN (td),
    tc union
      PP_CONTEXT (EXTRACT_LABEL (td).header.pp) union
      CURRENT_CONTEXT (EXTRACT_CHILDREN (td))) and
  tc intersect
    PP_CONTEXT (EXTRACT_LABEL (td).header.pp) = empty_set and
  tc intersect
    CURRENT_CONTEXT (EXTRACT_CHILDREN (td)) = empty_set and
  PP_CONTEXT (EXTRACT_LABEL (td).header.pp) intersect
    CURRENT_CONTEXT (EXTRACT_CHILDREN (td)) = empty_set)

Figure 46: The TYPE_CONTEXT_MACHINERY Module (continued)

114
math operation IS_WELL_FORMED_WRT_CONTEXT (  
  tds: TYPE_DECLARATION_SET  
  tc: TYPE_CONTEXT  
): boolean  
explicit definition  
for all td: TYPE_DECLARATION  
where (td is in RANGE (tds))  
(IS_WELL_FORMED_WRT_CONTEXT (td, tc)

math operation IS_WELL_FORMED_WRT_CONTEXT (  
  tds: TYPE_DECLARATION_SET  
): boolean  
explicit definition  
IS_WELL_FORMED_WRT_CONTEXT (  
  tds, INITIAL_TYPE_CONTEXT union CURRENT_CONTEXT (tds)) and  
INITIAL_TYPE_CONTEXT intersect CURRENT_CONTEXT (tds) = empty_set

math operation CURRENT_CONTEXT (  
  tds: TYPE_DECLARATION_SET  
): TYPE_CONTEXT  
explicit definition  
{tid: ID2, arity: ARITY  
where (there exists id: ID1  
  (EXTRACT_HEADER (tds, id).tid = tid and  
  |EXTRACT_HEADER (tds, id).pp| = arity))} (tid, arity)

math operation PP_CONTEXT (  
  pp: string of ID2  
): TYPE_CONTEXT  
explicit definition  
{tid: ID2  
where (tid is in elements (pp))} (tid, 0)

Figure 47: The TYPE_CONTEXT_MACHINERY Module (continued)
math operation CONTEXT {
    tds: TYPE_DECLARATION_SET
    path: string of IDl
} : TYPE_CONTEXT

implicit definition
if path = empty_string
then
    CONTEXT (tds, path) = INITIAL_TYPE_CONTEXT union
    CURRENT_CONTEXT (tds)
else
    CONTEXT (tds, path) =
    CURRENT_CONTEXT (tds) union
    PP_CONTEXT (
        EXTRACT_LABEL (APPLY (tds, first (path))).header.pp) union
    CONTEXT (EXTRACT_CHILDREN (APPLY (tds, first (path))),
        all_but_first (path))
end TYPE_CONTEXT_MACHINERY
• **IS_WELLFORMED_WRT_CONTEXT** – This Boolean function is true if and only if the given **TYPE_DECLARATION** is valid with respect to the given **TYPE_CONTEXT**. This is true if and only if:

1. The body of the given **TYPE_DECLARATION** is valid with respect to the context obtained by taking the union of the given context, the nested context, and the parameter profile context, and

2. Every **TYPE_DECLARATION** in the local scope of the given **TYPE_DECLARATION** is itself valid with respect to the context obtained by taking the union of the given context, the nested context, and the parameter profile context, and

3. The given context, the nested context, and the parameter profile context are pairwise disjoint.

• **IS_WELLFORMED_WRT_CONTEXT** – This Boolean function is true if and only if the given **TYPE_DECLARATION_SET** is valid with respect to the given **TYPE_CONTEXT**. This is true if and only if every **TYPE_DECLARATION** in the range of the given **TYPE_DECLARATION_SET** is itself valid with respect to the given context.

• **IS_WELLFORMED_WRT_CONTEXT** – This Boolean function is true if and only if

1. The given **TYPE_DECLARATION_SET** is valid with respect to the context obtained by taking the union of the built-in context (**INITIAL_TYPE_CONTEXT**) and the nested context, and

2. The given context and the nested context are disjoint.
• **CURRENT_CONTEXT** – Given a `TYPE_DECLARATION_SET`, it gives the 
`TYPE_CONTEXT` made up of the type constructors defined at the top level of each 
`TYPE_DECLARATION` in the range of the given `TYPE_DECLARATION_SET`. Each type 
constructor is assigned an appropriate arity, depending on the number of arguments in 
the corresponding `TYPE_DECLARATION`.

• **PP_CONTEXT** – Given a string of ID2 (a parameter profile), it gives the 
`TYPE_CONTEXT` made up of the type constructors in the given string. All of them are 
assigned arity 0.

• **CONTEXT** – Given a `TYPE_DECLARATION_SET` and a string of ID1, it gives the 
`TYPE_CONTEXT` available at the position obtained by following the path defined by 
the given string of ID1 through the given `TYPE_DECLARATION_SET`.

This completes our description of the `TYPE_CONTEXT_MACHINERY` module, and of 
the math models and operations necessary to formally specify the type declaration editor.

### 3.6 Type Declaration Editor

In this section we put it all together and present the formal specifications for a 
conceptual type declaration editor. We start by giving an intuitive overview of the features 
of the editor, and the rationale behind the chosen design. Then we describe in some detail 
the context and interface sections in the `Type_Declaration_Editor` concept 
(Figures 49-60).

#### 3.6.1 Intuitive Model and Editor Overview

We want to design an editor for type declarations. The fundamental design principle 
we are going to follow is to provide only a minimal, yet functionally complete, set of 
primitive operations to build and manipulate type declarations. The rationale behind this
decision is that we want operations that are conceptually simple and clear, and it should be possible, using the operations provided, to implement any higher-level conceptual operation.

It is useful to think of the editor as being at all times in one of two distinct states: either we are editing a type declaration set or we are editing a type expression. This is determined by the current position in the edited structure. The type declaration set or the type expression being edited is called the current entity. Depending on the state the editor is in different operations are available to edit the current entity or to change the position and hence which entity is the current.

When editing a type declaration set we can move down and up the type declaration set hierarchy (i.e., we can open the nested scope of one of the type declarations in the current set, or we can close the current type declaration set and move one level up). We can add a new declaration to the set, delete an existing declaration, or modify an existing declaration. To modify a declaration we can either edit its body or its local context (a nested set of type declarations). At any time while in a type declaration set, we can open the body of one of the type declarations and start editing a type expression. Similarly, while editing a type expression (the body of a type declaration), we can close it and move back to the enclosing type declaration set.

When editing a type expression we can move down and up the type expression hierarchy by opening one of the children of the current type expression, or closing the current type expression and moving back to editing the parent. We can also change the root label (type constructor) of the current type expression, and add/remove children (within the validity constraints imposed by the model).
The editor also provides enough operations to inspect the structure. These are necessary to implement the user interface and higher-level operations since they allow the testing of the other operations' preconditions.

Let's take a look at the formal specification of this editor. It is expressed as a concept in the RESOLVE language, so it has a structure similar to the math modules we have seen so far: a context describing the other entities the concept depends on and any local entities not exported, and an interface defining the new entities exported by the component.

3.6.2 The Context

The context section (Figures 49-51) is divided in three subsections: the global context, the parametric context, and the local context.

The global context (Figure 49) simply lists the components the concept relies on. So, of course, it includes all the math modules we have described in this chapter. In addition, it also refers to three program components (Standard_Boolean_Facility, Standard_Integer_Facility, and ID_Template) needed by the concept. The two facilities provide the program types Boolean and Integer which are not built-in to the RESOLVE language. The ID_Template concept exports, in addition to a type ID, operations to generate a new unique ID value, to duplicate an ID value, and to compare two ID values (see Appendix C for the definition of the concept).

8. A concept in RESOLVE is an abstract component exporting 0 or more program types and 0 or more program operations. It is abstract in the sense that only the specification of the model of the type(s) and of the behavior of the operation(s) is present. No information is provided about the implementation of the component. See [19] for a detailed description of RESOLVE concepts and the language used.
concept Type_Declaration_Editor

context

global context

  mathematics TYPE_EXPRESSION_TEMPLATE
  mathematics TYPE_EXPRESSION_MACHINERY
  mathematics TYPE_DECLARATION_TEMPLATE
  mathematics TYPE_DECLARATION_MACHINERY
  mathematics POSITION_TEMPLATE
  mathematics POSITION_MACHINERY
  mathematics TYPE_CONTEXT_TEMPLATE
  mathematics TYPE_CONTEXT_MACHINERY

  facility Standard_Boolean_Facility
  facility Standard_Integer_Facility

concept ID_Template

parametric context

type ID1

facility ID2_Facility is ID_Template ()

Figure 49: The Type_Declaration_Editor Concept
local context

renaming
  ID2_Facility.ID as ID2

math facility TYPE_CONTEXT_FACILITY is
  TYPE_CONTEXT_TEMPLATE (math[ID2])

math operation INITIAL_TYPE_CONTEXT : TYPE_CONTEXT
  restriction
    (NIL, -1) is in INITIAL_TYPE_CONTEXT

math operation ARITY ( tid: math[ID2] ) : integer
  implicit definition
    if DEFINED_IN (INITIAL_TYPE_CONTEXT, tid) then ARITY (tid) = APPLY (INITIAL_TYPE_CONTEXT, tid)
    else ARITY (tid) = ARITY (tid, tds.wp.tds)

math facility TYPE_EXPRESSION_FACILITY is
  TYPE_EXPRESSION_TEMPLATE (math[ID2], NIL, ARITY)

math facility TYPE_EXPRESSION_ENHANCED_FACILITY is
  TYPE_EXPRESSION_MACHINERY (math[ID2], NIL, ARITY, TYPE_EXPRESSION_FACILITY)

math facility TYPE_DECLARATION_FACILITY is
  TYPE_DECLARATION_TEMPLATE ( math[ID1], math[ID2], NIL, ARITY, TYPE_EXPRESSION_FACILITY)

math facility TYPE_DECLARATION_ENHANCED_FACILITY is
  TYPE_DECLARATION_MACHINERY (math[ID1], math[ID2], NIL, ARITY, TYPE_EXPRESSION_FACILITY, TYPE_EXPRESSION_ENHANCED_FACILITY, TYPE_DECLARATION_FACILITY)

Figure 50: The Type_Declaration_Editor Concept (continued)
math facility POSITION_FACILITY is
POSITION_TEMPLATE {
    math[ID1], math[ID2], NIL, ARITY, TYPE_EXPRESSION_FACILITY, TYPE_DECLARATION_FACILITY
}

math facility POSITION_ENHANCED_FACILITY is
POSITION_MACHINERY {
    math[ID1], math[ID2], NIL, ARITY, TYPE_EXPRESSION_FACILITY, TYPE_EXPRESSION_ENHANCED_FACILITY
    TYPE_DECLARATION_FACILITY, TYPE_DECLARATION_ENHANCED_FACILITY,
    POSITION_FACILITY
}

math facility TYPE_CONTEXT_ENHANCED_FACILITY is
TYPE_CONTEXT_MACHINERY {
    math[ID1], math[ID2], NIL, ARITY, TYPE_EXPRESSION_FACILITY, TYPE_EXPRESSION_ENHANCED_FACILITY
    TYPE_DECLARATION_FACILITY, TYPE_DECLARATION_ENHANCED_FACILITY,
    TYPE_CONTEXT_FACILITY
}

state variables
tdswp: TYPE_DECLARATION_SET_WITH_POSITION
used: set of math[ID2]
constraint
    IS_WELL_FORMED_WRT_CONTEXT (tdsws.tds)
initialization
    ensures
        tdswp = (EMPTY_TYPE_DECLARATION_SET,
        INITIAL_TDS_POSITION) and
        used = DOMAIN (INITIAL_TYPE_CONTEXT)

Figure 51: The Type_Declaration_Editor Concept (continued)
The parametric context (Figure 49) lists the template parameters for the Type_Declaration_Editor concept. The concept is parameterized by two program types, ID1 and ID2_Facility.ID (which is renamed to ID2). ID1 is going to be the type used to identify type declarations. ID2 is the type used to identify type constructors (built-in, declared in type declarations, and appearing as formal arguments in type declarations). The reason that ID2 is not any program type (like ID1), but instead a type exported by an instance of the ID_Template concept, is that the editor has to be in charge of generating unique ID2s whenever necessary, so it requires the extra operations provided by the ID_Template concept.

The local context (Figures 50-51) has three parts. In the first part we rename ID2_Facility.ID as ID2 for consistency with the naming scheme used in the math modules. In the next part we declare local instances of all the necessary math modules. In addition two operation declarations, INITIAL_TYPE_CONTEXT and ARITY, are introduced here. The first one is only an operation header with a restriction. This implies that it the responsibility of the implementation to provide a suitable definition for the operation. In this case, the implementation will choose what type constructors to define in the built-in type context. The restriction guarantees that it will include at least one type constructor called NIL with arity -1. The second declaration uses the implementation-supplied INITIAL_TYPE_CONTEXT to define the ARITY math operation.

The third part of the local context declares the state variables used by the editor (Figure 51). There are only two state variables, tdswp and used. tdswp is used to model the edited set of type declarations with a record of the current position at which the editor operations are going to be applied; used is a set of ID2s that is used to keep track of what identifiers have already been used so that the editor can ensure that all identifiers are going
to be different and thus unique. A constraint is imposed on the state variable \( \text{tdswp} \) that guarantees that the edited set of type declarations is always valid in that all used symbols (type constructors) are either built-in or they are declared where they can be accessed.

The \( \text{tdswp} \) state variable is initialized to model an empty set of type declarations. The \( \text{used} \) state variable is initialized to record that all the identifiers for type constructors in the built-in type context have already been used (and should not be used for any used-declared type constructors).

At all times the value of the state variable \( \text{tdswp} \) determines whether the user is editing a TDS or a TE (the body of a TD). The flag \( \text{tdswp.pos.body_is_open} \) keeps track of this. When the user is editing a TDS, there is exactly one of them that is the current one, i.e. the target of the operations. Similarly, when the user is editing a TE, there is exactly one of them that is accessible at one time.

### 3.6.3 The Interface

The interface section of the `Type_Declaration_Editor` concept (Figures 52-60) lists all the editor's exported operations, and their specifications. For each operation, the specifications include the operation header (with the name, the formal parameters, the formal parameter conceptual modes), the list of the state variables referenced by the operation (also with conceptual modes specifying how the operation is going to manipulate the state variables), and \textit{requires} and \textit{ensures} clauses formally describing the operation's behavior.

The operations in the type declaration editor can be divided into three groups: the operations to traverse and modify the `TYPE_DECLARATION_SET` (TDS) hierarchy (Figures 52-54), the operations to traverse and modify a `TYPE_EXPRESSION` (TE, the
body of a TYPE_DECLARATION, TD) (Figures 54-56), and the operations to observe the state of the editor (i.e., the values of the tdswp state variable), needed to test the preconditions of other operations (Figures 57-60). All the operations in the first group require that tdswp.pos.body_is_open be false. All those in the second group require that tdswp.pos.body_is_open be true.

3.6.3.1 TDS Operations

Open_TDS, Close_TDS, Reset_TDS allow the user to move around the TDS hierarchy (see Figure 52):

- **Open_TDS** moves one level down the hierarchy, or, in other words, makes the local scope of the specified TD the current TDS. It requires that the given ID1 be defined in the current TDS. It consumes the given ID1.

- **Close_TDS** moves one level up the hierarchy, i.e., it makes the TDS that contains the TD whose local scope is the current TDS the new current TDS. It requires that the current TDS not be the top one. It produces the ID1 of the TD whose local scope was the original current TDS.

- **Reset_TDS** moves back to the top of the hierarchy and makes the current TDS the top one. It does not have any other precondition other than that tdswp.pos.body_is_open be false.

New_TD, Delete_TD allow the user to create new type declarations and delete existing ones (see Figure 53), and Open_TD_Body switches the editor to the type expression editing state (see Figure 54):
interface

operation Open_TDS (  
    consumes id: IDl  
)  
referenced state variables  
alters tdswp  
requires  
    not tdswp.pos.body_is_open and  
    DEFINED_IN (EXTRACT_CURRENT (tdswp), id)  
ensures  
    tdswp.tds = #tdswp.tds and  
    tdswp.pos = (#tdswp.pos.path * <id>, false,  
        INITIAL_TE_POSITION)  

operation Close_TDS (  
    produces id: IDl  
)  
referenced state variables  
alters tdswp  
requires  
    not tdswp.pos.body_is_open and  
    tdswp.pos.path /= empty_string  
ensures  
    tdswp.tds = #tdswp.tds and  
    tdswp.pos = (all_but_last (#tdswp.pos.path), false,  
        INITIAL_TE_POSITION) and  
    id = last (#tdswp.pos.path)  

operation Reset_TDS ()  
referenced state variables  
alters tdswp  
requires  
    not tdswp.pos.body_is_open  
ensures  
    tdswp.tds = #tdswp.tds and  
    tdswp.pos = INITIAL_TDS_POSITION

Figure 52: The Type_Declaration_Editor Concept (continued)
operation New_TD {
    consumes id: ID1
    consumes arity: Integer
}

referenced state variables
alters tdswp
alters used
requires
not tdswp.pos.body_is_open and
arity >= 0 and
not DEFINED_IN (EXTRACT_CURRENT (tdswp), id)
ensures
DIFFER_ONLY_AT_CURRENT (tdswp, #tdswp) and
there exists tid: ID2, pp: string of ID2
(tid is not in #used and
|pp| = arity and
|pp| = |elements (pp)| and
#used intersect elements (pp) = empty_set and
used = #used union {tid} union elements (pp) and
EXTRACT_CURRENT (tdswp) =
    EXTRACT_CURRENT (#tdswp) union
    {(id, INITIAL_TYPE_DECLARATION ((tid, pp)))}

operation Delete_TD {
    preserves id: ID1
}

referenced state variables
alters tdswp
requires
not tdswp.pos.body_is_open and
DEFINED_IN (EXTRACT_CURRENT (tdswp), id) and
|EXTRACT_CURRENT (tdswp.tds, tdswp.pos.path * <id>)| = 0 and
DEPENDENCY_COUNT (tdswp.tds,
    EXTRACT_HEADER (EXTRACT_CURRENT (tdswp), id).tid) = 0)
ensures
DIFFER_ONLY_AT_CURRENT (tdswp, #tdswp) and
there exists td: TYPE_DECLARATION
    (EXTRACT_CURRENT (tdswp) =
    EXTRACT_CURRENT (#tdswp) without {(id, td)})

Figure 53: The TypeDeclaration_Editor Concept (continued)
operation Open_TD_Body (  
    consumes  id: ID1  
  )

  referenced state variables
  alters    tdswp

  requires
    not tdswp.pos.body_is_open and
    DEFINED_IN (EXTRACT_CURRENT (tdswp), id)

  ensures
    tdswp.tds = #tdswp.tds and
    tdswp.pos = (#tdswp.pos.path * <id>, true, INITIAL_TE_POSITION)

operation Close_TD_Body (  
    produces  id: ID1  
  )

  referenced state variables
  alters    tdswp

  requires
    tdswp.pos.body_is_open

  ensures
    tdswp.tds = #tdswp.tds and
    tdswp.pos = (all_but_last (#tdswp.pos.path), false, INITIAL_TE_POSITION) and
    id = last (#tdswp.pos.path)

operation Open_TE (  
    consumes  pos: Integer  
  )

  referenced state variables
  alters    tdswp

  requires
    tdswp.pos.body_is_open and
    0 <= pos < 
    |EXTRACT_CHILDREN (EXTRACT_BODY (tdswp.tds, tdswp.pos.path))|

  ensures
    tdswp.tds = #tdswp.tds and
    tdswp.pos = 
    (#tdswp.pos.path, true, #tdswp.pos.body_position * <pos>)

Figure 54: The Type_Declaration_Editor Concept (continued)
• **New_TD** creates a new "initial" TD and adds it to the current TDS (without modifying anything else). An initial TD is one that has a number of arguments specified by the user, an empty local scope, and a **NIL** leaf as its body. The operation requires that the given ID1 for the new operation not be defined in the current TDS, and that the arity provided (number of arguments for the new type constructor) be non-negative. A new ID2 is generated by the operation to be used as the new type constructor in TE, and a new ID2 is also generated for each of the arguments. All these ID2s, of course, are generated to be unique.

• **Delete_TD** removes the specified TD from the current TDS (without modifying anything else). It requires that the given ID1 be defined in the current TDS, that the specified TD be a leaf in the TDS hierarchy, and also that deleting the TD not break the constraint on the whole TDSWP, i.e., the ID2 defined by the chosen TD must not be used anywhere outside the TD itself.

• **Open_TD_Body** moves the editor to the body of the specified TD, so that now the editor is in TE editing mode (tdswp.pos.body_is_open is set to true). It requires that the given ID1 be defined in the current TDS. It consumes the given ID1.

### 3.6.3.2 TE Operations

Since we concluded the section on TDS operations with **Open_TD_Body**, let's start this section with its counterpart **Close_TD_Body** to switch the editor to the type declaration set editing state (see Figure 54):

• **Close_TD_Body** moves the editor back to TDS editing mode, and the current TDS becomes the one containing the TD whose body was currently being edited. It does not have any other requirements other than tdswp.pos.body_is_open be true.

130
operation Close_TE {
  produces pos: Integer
}

referenced state variables
alters tdswp
requires
tdswp.pos.body_is_open and
tdswp.pos.body_position /= empty_string
ensures
tdswp.tds = #tdswp.tds and
tdswp.pos = (#tdswp.pos.path, true,
          all_but_last (#tdswp.pos.body_position)) and
  pos = last (#tdswp.pos.body_position)

operation Reset_TE ()

referenced state variables
alters tdswp
requires
tdswp.pos.body_is_open
ensures
tdswp.tds = #tdswp.tds and
tdswp.pos = (#tdswp.pos.path, true, INITIAL_TE_POSITION)

operation Insert_NIL_Leaf {
  preserves pos: Integer
}

referenced state variables
alters tdswp
requires
tdswp.pos.body_is_open and
  ARITY (EXTRACT_TID (tdswp)) = -1 and
  0 <= pos <=
  |EXTRACT_CHILDREN (EXTRACT_BODY (tdswp.tds, tdswp.pos.path))|
ensures
  DIFFER_ONLY_AT_CURRENT (tdswp, #tdswp) and
  there exists left, right: TYPE_EXPRESSION_STRING
  (EXTRACT_CHILDREN (#tdswp) = left * right and
  |left| = pos and
  EXTRACT_CURRENT (tdswp) =
          COMPOSE (EXTRACT_TID (#tdswp), left * <NIL_LEAF> * right))

Figure 55: The Type_Declaration_Editor Concept (continued)
operation Delete_TE (  
    preserves pos: Integer  
)  
referenced state variables  
alters tdswp  
requires  
tdswp.pos.body_is_open and  
ARITY (EXTRACT_TID (tdswp)) = -1 and  
0 <= pos <  
|EXTRACT_CHILDREN (EXTRACT_BODY (tdswp.tds, tdswp.pos.path))|  
ensures  
DIFFER_ONLY_AT_CURRENT (tdswp, #tdswp) and  
EXTRACT_CURRENT (tdswp) =  
    COMPOSE (EXTRACT_TID (#tdswp),  
    all_but_nth (EXTRACT_CHILDREN (#tdswp), pos))  

operation Swap_Type_ID (  
    alters tid: ID2  
)  
referenced state variables  
alters tdswp  
requires  
tdswp.pos.body_is_open and  
(ARITY (tid) = -1 or  
  ARITY (tid) = |EXTRACT_CHILDREN (tdswp)|) and  
DEFINED_IN (CONTEXT (tdswp.tds, tdswp.pos.path), tid)  
ensures  
DIFFER_ONLY_AT_CURRENT (tdswp, #tdswp) and  
tid = EXTRACT_TID (#tdswp) and  
EXTRACT_CURRENT (tdswp) = COMPOSE (#tid,  
    EXTRACT_CHILDREN (#tdswp))  

Figure 56: The Type_Declaration_Editor Concept (continued)
Open_TE, Close_TE, Reset_TE allow the user to move around a TE's hierarchy (see Figures 54-55):

- **Open_TE** moves one level down the hierarchy, or, in other words, makes the specified subtree's root the current node. It requires that the given position be in the right range (between 0 and the number of children of the current node).

- **Close_TE** moves one level up the hierarchy, i.e., it makes the parent node of the current node the new current node. It requires that the current node not be the top one.

- **Reset_TE** moves back to the top of the hierarchy and makes the current node the top one. It does not have any other precondition other than \( \text{tdswp.pos.body_is_open} \) be true.

Insert_NIL_Leaf, Delete_TE, Swap_Type_ID are the operations that allow the user to build and modify a TE. The first two have a precondition that the specified position at which to perform the operation be in the appropriate range \( 0 \leq \text{position} \leq \text{number of children of the current node} \), for Insert_NIL_Leaf, and \( 0 \leq \text{position} < \text{number of children of the current node} \), for the Delete_TE) (see Figures 55-56):

- **Insert_NIL_Leaf** adds a new child subexpression (a NIL leaf) to the current node at the specified position. It requires that the current node be labeled with a type constructor of arbitrary arity (NIL or tuple).

- **Delete_TE** removes the subexpression at the specified position. It requires that the current node be labeled with a type constructor of arbitrary arity (NIL or tuple).

- **Swap_Type_ID** swaps the root label of the current TE with the given ID2. It requires that the given ID2 be in the context at the current position (i.e., it must be in scope), and also arity of the given ID2 be compatible with the number of children of the current TE (so that swapping the root label will not break the validity of the structure).
3.6.3.3 Other Operations

These are all functions, and therefore they do not modify the state variables. They simply return information about the \texttt{tdswp} state variable\textsuperscript{9}. All the operation in the first group have the precondition that \texttt{tdswp.pos.body_is_open} is false (Figures 57-58):

- \texttt{IsDefined} returns true if and only if the given \texttt{ID1} is defined in the current TDS. It requires that \texttt{tdswp.pos.body_is_open} be false.

- \texttt{Current_Size} returns the size of the current TDS. It has no other prerequisites.

- \texttt{TDS_Path_Length} returns the depth of the current position in the TDSWS structure. It has no other prerequisites.

- \texttt{TDS_Type_ID} returns the type id of the type constructor defined by the TD specified by the given \texttt{ID1} in the current TDS. It requires that the given \texttt{ID1} be defined in the current TDS.

- \texttt{TDS_Argument} returns the type id of the argument at the given position in the TD specified by the given \texttt{ID1} in the current TDS. It requires that the given \texttt{ID1} be defined in the current TDS, and that the given position be in the appropriate range (between 0 and the arity of the corresponding type constructor).

- \texttt{TDS_PP_Length} returns the length of the parameter profile of the TD specified by the given \texttt{ID1} in the current TDS (i.e., the arity of the corresponding type constructor). It requires that the given \texttt{ID1} be defined in the current TDS.

\textsuperscript{9} There is no reason for the editor to supply any information about the \texttt{used} state variable, since its only role is for the editor to keep track of what \texttt{ID2}s have already been used.
operation Is_Defined (
    preserves id: ID
) : Boolean

referred state variables
preserves tdswp
requires
  not tdswp.pos.body_is_open
ensures
  Is_Defined =
    DEFINED_IN (EXTRACT_CURRENT (tdswp), id)

operation Current_Size (): Integer
referred state variables
preserves tdswp
requires
  not tdswp.pos.body_is_open
ensures
  Current_Size = |EXTRACT_CURRENT (tdswp)|

operation TDS_Path_Length (): Integer
referred state variables
preserves tdswp
requires
  not tdswp.pos.body_is_open
ensures
  TDS_Path_Length = |tdswp.pos.path|

operation Dependency_Count (
    preserves tid: ID2
) : Integer
referred state variables
preserves tdswp
requires
  not tdswp.pos.body_is_open and
  IS_DEFINED (tid, tdswp.tds)
ensures
  Dependency_Count = DEPENDENCY_COUNT (tdswp.tds, tid)

Figure 57: The Type_Declaration_Editor Concept (continued)
operation TD_Type_ID (  
    preserves id: IDI  
) : ID2  
referenced state variables  
    preserves tdswp  
requires  
    not tdswp.pos.body_is_open and  
    DEFINED_IN (EXTRACT_CURRENT (tdswp), id)  
ensures  
    TD_Type_ID = EXTRACT_HEADER (EXTRACT_CURRENT (tdswp), id).tid

operation TD_Argument (  
    preserves id: IDI  
    preserves pos: Integer  
) : ID2  
referenced state variables  
    preserves tdswp  
requires  
    not tdswp.pos.body_is_open and  
    DEFINED_IN (EXTRACT_CURRENT (tdswp), id) and  
    0 <= pos < |EXTRACT_HEADER (EXTRACT_CURRENT (tdswp), id).pp|  
ensures  
    TD_Argument =  
    nth (EXTRACT_HEADER (EXTRACT_CURRENT (tdswp), id).pp, pos)

operation TD_PP_Length (  
    preserves id: IDI  
) : Integer  
referenced state variables  
    preserves tdswp  
requires  
    not tdswp.pos.body_is_open and  
    DEFINED_IN (EXTRACT_CURRENT (tdswp), id)  
ensures  
    TD_PP_Length = |EXTRACT_HEADER (EXTRACT_CURRENT (tdswp), id).pp|

Figure 58: The TypeDeclaration_Editor Concept (continued)
operation TE_Path_Length () : Integer
    referenced state variables
    preserves tdswp
    requires
        tdswp.pos.body_is_open
    ensures
        TE_Path_Length = |tdswp.pos.body_position|

operation TE_Type_ID () : ID2
    referenced state variables
    preserves tdswp
    requires
        tdswp.pos.body_is_open
    ensures
        TE_Type_ID = EXTRACT_TID (tdswp)

operation Current_Length () : Integer
    referenced state variables
    preserves tdswp
    requires
        tdswp.pos.body_is_open
    ensures
        Current_Length = |EXTRACT_CHILDREN (tdswp)|

operation Is_Defined_In_Context ( tid: ID2 ) : Boolean
    referenced state variables
    preserves tdswp
    requires
        tdswp.pos.body_is_open
    ensures
        Is_Defined_In_Context =
            DEFINED_IN (CONTEXT (tdswp.tds, tdswp.pos.path), tid)

Figure 59: The Type_Declaration_Editor Concept (continued)
operation Body_Is_Open () : Boolean
  referenced state variables
  preserves tdswp
  ensures
  Body_Is_Open = tdswp.pos.body_is_open

operation Arity ( tid: ID2 ) : Integer
  referenced state variables
  preserves tdswp
  requires
  DEFINED_IN (INITIAL_TYPE_CONTEXT, tid) or
  IS_DEFINED (tid, tdswp.tds)
  ensures
  Arity = ARITY (tid)

end Type_Declaration_Editor
• **Dependency_Count** returns the number of uses in the whole TDSWP structure of the given ID2 (outside of the its TD). In other words, it returns the number of dependencies that would be broken if we deleted the TD for the given ID2 from the TDSWP structure. It requires that the given ID2 be declared somewhere in the TDSWP structure.

All the operations in the second group have only one precondition that `tdswp.pos.body_is_open` is true (Figure 59):

- **TE_Path_Length** returns the depth of the current position in the currently edited TE.
- **TE_Type_ID** returns the label (ID2) of the current node in the edited TE.
- **Current_Length** returns the number of subexpressions of the current TE.
- **Is_Defined_In_Context** returns true if and only if the given ID2 is scope at the current TE.

Finally, here are the last two operations (Figure 60):

- **Body_Is_Open** returns true if and only if the value of `tdswp.pos.body_is_open` is true (i.e., if a TE is being edited). It has no requires.
- **Arity** returns the arity of the given type constructor. It requires that the given ID2 be in the context somewhere (i.e., it is either a built-in type constructor or it is declared somewhere in the TDSWP structure).

This concludes our description of the Type_Declaration_Editor concept. In the next section we discuss several issues connected with the limitations of this editor, and how the ideas presented in this chapter could be extended to a more general setting.
3.7 Generalization and Extensions

The model and the editor described in this chapter have two obvious limitations (by design): they only involve a few of the many conceptual entities in a real programming language, and they provide a restricted set of primitive operations to build and manipulate the edited structure. In this section we explore some possible extensions to the model and the editor, pointing out some of the issues in the design of conceptual editors that were not addressed by our earlier discussion.

3.7.1 Extending the Model

To generalize the model to that of programs written in a complete programming language, we need to discuss how to model the following three items:

- the conceptual entities that make up a program,
- the position (for editing), and
- the context.

Our approach in this chapter, and the recommended approach in building the formal description of the conceptual model for a conceptual editor, is to model each conceptual entity individually. The objective is to model exactly the relevant aspects of the entity and nothing else. So, for instance, we choose to model type declarations with nested maps instead of settling for the classical tree model. An alternative approach would be to use a single, uniform model (e.g., the abstract syntax tree).

However, in conceptual editors, and we want the model to reflect the conceptual differences among distinct conceptual entities and among distinct programming tasks. Thus, the first approach is the desirable choice. However, the former approach does suffer
from one major drawback when compared with the latter, and that is the complexity of the model. This becomes apparent when we try to generalize the model to programs that might be written in a complete programming language.

We will now consider some issues involved in the modeling of each of the three items listed above, as they apply to the RESOLVE language.

3.7.1.1 The Program’s Building Blocks

Informally, a RESOLVE module is made up of a context section followed by an interface section. The context section is usually divided into three parts: the global context, the parametric context, and the local context. It specifies all the components and entities required by the module. Each of the three context sections is a list of items that can include other modules, constants, types, etc. The interface section defines the entities exported by the module, namely, types and operations.

Each entity declared in a module is made up of other smaller pieces. For instance, an operation has a header and a body. The header includes the operation’s name, the parameter profile (the list of formal parameters where for each parameter the abstract parameter mode, name, and type is specified), a return type if it is a function, and requires and ensures assertions. The body has an optional nested context section followed by a list of statements. Statements are very similar to those from other Pascal-like languages, so they can include control structures which contain nested lists of statements.

10. In this section we are just providing a high-level overview of the conceptual entities that make up a RESOLVE component. This suffices for the current discussion. A more detailed description of the RESOLVE language is included in [94].

141
It is possible to give such an informal description of all the entities describable in a RESOLVE module. What is important to note here is that, although the RESOLVE language includes ways of describing tens of entities, all these entities can be modeled by a few mathematical structures. In particular:

- We need to be able to group several heterogeneous entities such as those in an operation header, and this can be done by using the tuple mathematical type.
- We need to express sequences of homogeneous entities in which the order is important such as a formal parameter list or a sequence of statements, and this can be done by using the string math type.
- We need to be able to model groups of homogeneous entities whose relative order is unimportant such as the declaration of a group of operations, and for this purpose we can use the set math type.
- We need to describe recursive, nested structures where the order of the children at any level in the hierarchy is important, and the tree model introduced in Section 3.3.2 is clearly the appropriate choice. This structure was used in modeling type expressions. Another example of where this structure would be useful is in modeling nested statements resulting from the use of control structures.
- Finally, we need to model recursive, nested structures where the order of the children is unimportant, and the nested map model introduced in Section 3.4.4 was designed exactly for this purpose. Examples of where this model may be needed is to model nested operations, or nested modules.
These mathematical structures together with primitive types like integer and boolean are all that is needed to model the complete structure of a RESOLVE module. In other words, the theories and models developed in this chapter possess all the necessary features to allow us to build an accurate model for programs written in a complete programming language such as RESOLVE.

3.7.1.2 The Position

Our approach to formalizing conceptual models of programs results not in a uniform hierarchy, but in a mixture of distinct nested hierarchies, and other grouping mechanisms (tuples, sets, strings, etc.). Recall that the position structure allows the user of an editor to express the location, in an edited program, where editor operations will take place. Thus, the model of position must be able to specify a location in any part of a program’s very diverse structure.

The position we modeled for the type declaration editor required us to first describe the position in a type expression, and then to model the position in a type declaration set. Since type expressions are nested inside type declarations, the position for type declaration included a type expression position in its model.

This is going to be the general structure of the position in a model of programs. Every time one hierarchical structure is nested inside another, the position for the nested structure must be part of the position for the outer structure. This means that deeper nesting of distinct hierarchical structures will result in a more complex model of position. The good news is that RESOLVE modules do not have too many levels of nesting of distinct hierarchical structures. Four is the deepest level any RESOLVE structure can reach. As an example consider the following nesting:

143
(i) We start with nested operations, modeled by nested maps.

(ii) Each operation's body is made up of a string of nested statements, modeled by a string of trees.

(iii) Some statements (e.g., loops) contain mathematical assertions, that are nested structures modeled by trees.

(iv) Assertions, like all mathematical expressions, include the "declaration" of new math variables (through the use of quantifiers), and these can include type expressions, that are modeled by trees. As we have seen earlier in the chapter, type expressions do not include any other hierarchy.

So after four nested levels we reach the bottom of this hierarchy of hierarchies, and the position in an operation or set of operations would have to include a position in a string of nested statements, which in turn would have to include a position in an assertion, which finally would include a position in a type expression.

There are similar examples in the RESOLVE math modules, but these levels of nesting are not common and not deep. So, although the complexity of the model of the position grows with the complexity of the structure of programs, it is still manageable.

3.7.1.3 The Context

The context in a complete model of programs is also going to be more complicated than the simple type context we considered in the type declaration editor. However, in this case too, the extra complexity is manageable.

First of all, the context would probably be modeled as a tuple, where each field represents the context for a specific kind of entity. We need a context for the defined constants, one for the variables, one for the types, one for the operations, etc. For each of
these, the context can be modeled as a partial function from a set of identifiers for the corresponding set of entities, to a description of all the properties of the entity that are needed to correctly use the entity.

For example, the constant context would map a constant id into its type id and possibly a value for the constant. The variable context would map a variable id into its type id. The type context would map a type id into some description of the important properties of the corresponding type. If the language defines type equivalence as structural equivalence, then the type context might map a type id into a type expression or some similar structure. Finally, the operation context would map an operation id into some description of the operation header, including the operation’s parameter type information and, if it is a function, the returned type id.

This brief analysis reveals that, although more complicated, models of the conceptual entities in programs for a complete programming language would still look very much like the models discussed in this chapter, and there are no essential issues that have not been addressed.

3.7.2 Extending the Operations

The type declaration editor offers a very primitive set of operations to build and manipulate type declarations and type expressions. This was done by design: we first need to be able to understand and formally specify simple conceptual operations before we can hope to be successful at handling more complex operations.
In this section we provide a glimpse into the problems that arise when we try to formally specify more complex operations. Specifically, we consider whether there are other conceptual entities that we might need to model in order to specify more powerful operations.

3.7.2.1 The Selection

As we have seen in Section 3.5.1, the position in the type declaration editor serves the purpose of specifying some of the arguments to the editor's operations. Since the primitive operations there only act on one entity at a time, the position is clearly adequate for the task. However, suppose we wanted to be able to *copy* or *move* a group of entities (assuming that this was done within the constraints imposed by the context, so that we avoid the conceptual problems described in Section 2.4).

It is clear that the position alone is not enough to specify the argument(s) to such operations. We can introduce the notion of *selection*. The selection can be thought of as an extension to position or as an addition to it. Either way it must capture the idea of a group or range of entities at a certain location in the structure.

Since the only two grouping mechanisms in our model are strings and sets (or actually partial functions), if we assume that only entities of the same kind could be selected together, the model of selection can be as simple as a pair of integers, to specify a range of consecutive entities in a string, or a set of ids, to specify a set of entities in a set (or nested map). Of course, more complex models are possible, but it is not clear how useful, meaningful, or comprehensible they would be.
3.7.2.2 The Clipboard

As we mentioned in Chapter 2, it is possible that some highly constrained form of cut and paste may be allowed in a conceptual editor, e.g., a purely syntactic cut and paste, where only structures that have no dependencies on the existing context and on which no other entity depends could be deleted and then pasted at a different location.

Specifying such an operation would probably require some sort of auxiliary “storage area” where the deleted structure could be held until the user is ready to paste it back. The traditional term used for such an entity is the clipboard.

The model of the clipboard depends on what structures and entities need to be stored in it. The two most natural alternatives are to either model the clipboard as a tagged union of the models of all the entities that can be stored in it, or to keep a separate clipboard for each kind of entity that needs to be stored. Again, more complex models of the clipboard are possible especially if in addition to the copied structures we also want to store some context information. But in any event, all the mathematical machinery needed to specify this is already available.

3.7.2.3 The Dependency Graph

The only operation in the type declaration editor that could breach the validity of the edited structure (because existing dependencies might be broken as a result of the operation) is \texttt{Delete_{TD}} to delete a type declaration from the current type declaration set. Its requires clause guarantees that this cannot happen. Only a type declaration that is not used anywhere outside its own declaration can be removed, so that no dependencies can be left “dangling”.

147
To specify this operation we simply need to keep track of the number of dependencies that exist outside the declaration itself. However, if we wanted to specify an operation that involves other entities and the dependencies among them (again, assuming that there are such operations that are desirable and not conceptually unmanageable), the notion of a dependency count may not suffice. For instance, suppose we wanted to change the specs for the `Delete_TD` operation to say that any use of the type constructor defined by the type declaration we want to delete should be replaced by `NIL` (and we removed the current `requires` clause). Now it would be convenient to have direct access to all the occurrences of the type constructor we are removing from the context.

In this case, we may need to model a *dependency graph*. This is a structure that explicitly keeps track of the dependencies among entities in the edited structure. In nature, the dependency graph is similar to the context: all the information necessary to build it is contained in the edited structure, but it may be convenient to model it explicitly to make the resulting specifications easier to write and read.

A dependency graph would probably have to be modeled with a mathematical directed graph whose nodes represent the conceptual entities in the edited structure and whose edges connect the node declaring an entity (e.g., a type declaration declaring a new type constructor) to the nodes using the entity (e.g., a type expression). This would allow us to easily refer to all the nodes that depend on a given entity and determine how many such dependencies exist.

This concludes our discussion of some of the ways in which the model described in this chapter could be generalized and extended to allow us to model programs of a complete programming language and more powerful operations for the editor.
3.8 Chapter Summary

This chapter describes a formally specified editor for type declarations. Intuitive and formal models for all the necessary conceptual entities are provided, including models for type expressions, type declarations, position in the type expression and type declaration hierarchies, and type context. Important issues in the design of conceptual program editors are discussed. Finally some possible generalizations and extensions to the editor are presented.
CHAPTER 4

RELATED WORK

This chapter provides an overview of existing approaches to program editor design. It establishes their limitations with respect to the conceptual models of programs and program construction they convey and their support for the programmer's task (or lack thereof).

4.1 Introduction

As we have seen in Chapter 2, a programmer using a program editor forms a mental model of program structure that is directly affected by the editing system. The main contention of this dissertation is that due to the substantial impact that the editor can have on the user's mental model, the editor should be designed to support, promote, and even enforce an appropriate conceptual model of software and software construction.

Other designers of program editors have also recognized the importance of the conceptual model presented by the editor. Minör [58], for example, points out that one of the weaknesses of hybrid editors\(^1\) is the lack of a clear, consistent model. He also states the importance of choosing an appropriate model and an appropriate user interface as the

---

1. Also called mixed-mode editors, hybrid editors allow both textual and structural editing of programs. See Section 4.2.2 for some details.
means to a good editor. Finally, he describes his view that (static) semantic information available in language-based environments should be used beyond its obvious support for semantically correct (meaningful) programs. It should "be further exploited to support the user, both for presentation, navigation and editing", but he does not delve into how this may be accomplished.

Welsh and Toleman [108] note that programmers have different models for different parts of a program. So it is appropriate for the editor to provide distinct views for distinct entities. They also state the very important point that a text-only representation of programs is bound to bias the user towards operations defined in textual terms. In our own terms, this is tantamount to saying that a text-only representation is bound to break any conceptual view the editor may be trying to convey.

In spite of these valid comments and the large number of editors and environments that have been developed, most existing systems have clearly not been designed with a well-thought-out conceptual model of software in mind. With a few exceptions, programs are considered to be textual in nature and editor designs reflect that both in the chosen representations and in the operations allowed by the editor.

In the remainder of the chapter we present the most common and popular approaches to program editor design: text editors, syntax-directed and structure editors, visual environments, the Smalltalk environment, and editor generators. We also describe a relatively new research effort—intentional programming—that appears to be a step in the right direction. Finally, we briefly examine the relationship between general productivity tools and conceptual editors.
4.2 Limitations and Problems with Existing Approaches

Program editors have been a rich field of research for over thirty years. Many different approaches have been explored. The main classes of program editors are general-purpose editors and language-based editors. General-purpose editors, such as plain text editors (emacs, vi), can be used to edit textual representations of programs as well as any other text document. Language-based editors instead are designed to build and manipulate program descriptions only, and take advantage of knowledge of the language to provide extra support to the programmer. Language-based editors can further be classified based on the program representations they manipulate. Syntax-directed editors and structure editors usually employ a textual representation, though through their knowledge of the language's syntax they help the programmer build syntactically correct programs. Visual environments use a mostly graphical notation for program representation, but they are still considered language-based environments since they define their own notation and language.

4.2.1 Text Editors

Text editors are general-purpose editing tools, i.e., they are not necessarily designed with the programming task in mind. However, they are still among the most popular editing tools used by programmers. Their only concrete representation is plain text, and they are used to edit strings of characters regardless of what they mean. Operations are provided that allow users to manipulate this representation in any way that makes sense in the realm of text strings. This means that the user can cut and paste any range of text...
regardless of where it appears and what it means. This also means that the programmer is responsible for fixing all the inconsistencies and flaws that are introduced in the code by most text operations.

In general, text editors do not provide any language support (e.g., vi [50]). However, there are text editors (e.g., emacs [96]) that, being extensible and customizable, can be made to support almost any language-based feature. Common extensions include capabilities such as allowing the user to enter syntactically correct templates for the various language constructs or checking the syntactic correctness of the edited representation from within the editor. In spite of this they remain text editors at heart.

Text editors are designed to edit text. Therefore there is no higher-level, structured, intended model of the edited object that the editor tries to convey. Programs are displayed as text and treated as text by all operations. The only conceptual model that these editors can convey is that programs are essentially strings of characters.

4.2.2 Syntax-Directed and Structure Editors

These are language-based editors that come in two different flavors. All editing is performed on some kind of structural representation of the program. The most common structure is the abstract syntax tree of the program. The defining characteristic for the editors in this class is that regardless of whether a textual representation [16, 23] or a graphical one [30, 58] is used for the edited structure, all the operations allowed by the editor operate at a structural level and not at a textual level. Editors in this class may allow operations such as “expand a node in the syntax tree” or “remove a subtree”. They allow only syntactically correct programs to be constructed, and they implicitly provide some guidance in the choice of syntactic constructs.
For instance, NSEDIT [30] is a programming environment for Pascal, based on the notion of Nassi-Shneiderman (N-S) charts. Designed for use by novice programmers in a teaching situation, it enables structural features of Pascal to be manipulated in terms of their N-S chart box representation.

The intended conceptual model of programs is clearly a hierarchical, nested structure—the abstract syntax tree. However, most editors in this class (and there are not too many of them for the reasons explained below) choose text as the only program representation. Programmers, given a textual object, feel constrained by structure editing and tend to resent the restrictions imposed by the editor on their mode of operation [58]. This is a clear sign either that the abstract syntax tree is an inadequate, if not inappropriate, model of software, or that structure editors with only a textual representation fail to convey the intended, higher-level conceptual model of programs and program construction.

Why are pure structure editors considered too restrictive? Programmers appear to want the freedom and flexibility of essentially building and manipulating any possible program description regardless of whether it makes any sense. In addition to that, Minör [58] lists several other common objections to structure-oriented editing for program construction and offers some possible solutions. His conclusion is that poor design of the interaction and presentation of structure-oriented editors is at the heart of the problem. This seems to be the fundamental reason why pure structure editors are not particularly popular, and hybrid (or mixed-mode) editors are a much more frequent choice in language-based editor design.

Hybrid editors are editors that allow the user to view and manipulate the concrete representation of a program in more than one way. The most common approach is to allow both a textual and a structural view of the program [3, 69, 81, 102]. Although the details
and the level of integration of the two modes may vary from editor to editor, the general idea is that the programmer can edit the program description as text or as a syntax tree, with the constraint that before something that has been edited textually can be edited structurally, the editor needs to parse it and verify that there are no syntax errors.

As an example of this kind of editor, Pan [102] is a language-based editing and browsing system designed to support development and maintenance of complex software documents. Pan's implementation combines several approaches: unrestricted text editing, language-based editing and browsing, description-driven language definition for incremental analysis and support for multiple languages per session.

In the case of mixed-mode editors it is not clear what conceptual model of programs the designers are trying to convey, if any. Their main goal is to provide the flexibility of text editing together with the potential convenience of structure editing. However, due to unrestricted text editing capability, it is unlikely that any programmer using such an editor will view a program as anything more than plain text.

4.2.3 Visual Environments

There is not yet general agreement as to what constitutes a visual programming language (VPL). Some refer to visual approaches to traditional programming languages, such as executable flowcharts, as visual programming. Others refer to visual approaches that depart from traditional programming, such as programming by demonstrating the desired actions on the screen. Others use the term to describe programming environments that support textual languages and that are used for graphical user interface programming, graphically depicting relationships and data-structure behavior, and visually combining textually programmed units to build new programs. Examples of environments providing
support for visual GUI programming include Microsoft's Visual Basic and Visual C++, and ParcPlace Systems' Smalltalk environment VisualWorks. These are general purpose visual programming environments (VPEs) that are built around a specific programming language and do not promote any specific model of software structure other than the one implicit in the language itself.

Yet another view is that of visual programming through domain-specific approaches. Developers of domain-specific VPLs and VPEs use visual expressions and terminologies that reflect the needs, problem-solving diagrams, and vocabulary specific to that domain. This approach has produced numerous successes in the research arena and in the marketplace, such as National Instruments LabView (laboratory data acquisition), Advanced Visual Systems AVS (scientific visualization), and Cypress Research PhonePro (telephone and voice-mail behavior).

Here, however, we are interested in general-purpose software editing environments, the models of software they convey, and the level of support for the programmer's task they provide. Therefore, we consider only "truly" visual programming environments, which use graphical information such as diagrams, icons, or tables in the actual process of programming. In these environments, the program meaning is as bound to its visual representation as a text-based language is tied to its syntax and semantics.

Prograph [72] is an example of such a general-purpose visual programming environment. It differs from other programming systems in that text is not the primary representation used to display a program. The system promotes a high-level, object-oriented conceptual model of programs. A Prograph application is made of one or more objects, which are instances of one or more classes. Each class defines a number of attributes and methods that all instances of the class will have. Prograph uses different
editors for different kinds of conceptual entities. For instance, there is a class hierarchy editor where the edited representation is a graphical display of the class tree, and a method editor where data flow diagrams are the chosen representation. There are also editors to modify a class's attributes, to add and remove methods, etc. The major shortcoming of Prograph as a prototypical conceptual editor is that it allows the programmer to edit descriptions that are not meaningful, relying on its run-time environment to reveal the errors and inconsistencies in the program.

4.2.4 Smalltalk

A programming system that has gained more and more popularity in the software production world is the multi-windowed, highly interactive, Smalltalk environment.

An executing Smalltalk program consists of messages sent to objects. Objects are instances of classes, which are considered to be abstract descriptions of the structure of an object, organized within a functional hierarchy. But these "abstract" descriptions do not include mathematical modeling or formal specifications, so it is not clear that they can be understood without looking at actual implementation details. Thus, for each class, the set of messages its instances respond to describes its functionality. The class hierarchy is built on the inheritance language mechanism. The basic Smalltalk system [26, 25] provides a large library of classes. Smalltalk programming consists of finding classes and methods within the hierarchy, understanding these methods or adding new classes and methods to the hierarchy. The main system tool for supporting these tasks is the class hierarchy browser. Actual coding of new classes or methods, however, takes place in a more traditional text-based or structure-based editing environment. The Smalltalk system provided an early object-oriented programming environment, conveying a more structured
model of programs than previous editors, but it suffers from the limitations of its editor, and from its model of "executing programs" that allows the programmer to edit descriptions that are not meaningful, relying on its run-time environment to reveal the errors and inconsistencies in the program.

4.2.5 Editor Generators

A very popular approach to the creation of language-based editors is the design and use of so called editor generators. These are tools that use descriptions of the abstract and concrete syntax of the target language, and, at times, also information about the static semantic properties of the language, to generate language-based program editors for the target language. They vary mainly in the range of textual vs. structural editing they can provide, and in the kinds of user interfaces (textual vs. graphical) they allow in the resulting editors. Here is a brief overview of some of the most popular systems in this class.

The Gandalf system [69] is a software development environment generator. It uses the ALOE [54] editor generation system to produce syntax-directed editors from the languages' syntax and static semantics, and a set of action routines and extended commands. The generated editors can have any combination of text and structure editing, including purely structural editing. In addition, multiple views of the same structure are possible (in principle, even graphical ones). Adding semantic checking can be difficult, but is possible. It appears that, in principle, this generator might be used to create conceptual editors, but we know of none that have been published.
Editors generated with the Synthesizer Generator [81] can span the full range of editing modes, from purely structural to purely textual. They can be generated to detect semantic errors, but they can't prevent them. They only use a textual representation. So, at best, they behave like syntax-directed editors; at worst, they behave like text editors.

PSG [2, 3] is a generator for semantic-based programming environments. From a definition of the syntax, and the static and dynamic semantics of the language, PSG generates an interactive, language-specific environment, which includes an editor and an interpreter. The basic building block in the system is the “incomplete program fragment” (that replaces our notion of meaningful program). The editor provides fully integrated text and structure hybrid editing capabilities, and although it apparently allows pure structural editing with prevention of static semantic errors, it is not clear whether it promotes a consistent, high-level view of programs because of its text-only representation of programs, the potential for textual editing of the representation, and the dangerous notion of incomplete program fragment.

Mentor [16, 17] is a structured document manipulation system. It is parameterized by the language to be manipulated. The basic document structuring mechanism is a labeled tree that emphasizes semantic structure. All manipulations of documents are performed on, and according to, their unique abstract tree representation. This includes interactive editing as well as program execution. Though all manipulations are performed on the abstract representation, the documents are always displayed in some user chosen concrete form (e.g., trees or text).
Mjølner/Orm [52, 59, 35] is a structure-based environment supporting incremental software and language development. The environment functionality is specified by grammars developed to support object-oriented languages, and these grammars are interpreted by the environment allowing both syntax and semantics to be checked on the fly.

4.3 Intentional Programming

Intentional programming (IP) is a recent interesting development in the landscape of programming language and program construction research. Simonyi describes the motivations and the fundamental ideas in intentional programming in [93].

Programming languages have serious limitations in their ability to express new abstractions. These are due mainly to the expressive constraints imposed by the language syntax and by the language's fixed set of abstraction mechanisms. The chosen language may not be expressive enough to say what we need; and domain specific knowledge seems to be confined to domain specific languages, so that normal systems that span several domains are usually written in general purpose languages and are not able to benefit from domain specific features.

*Intentions* are meta-abstractions with the following properties: they are reusable, high-level, domain-specific, and have a "concretion" process that hides the implementation details and that is allowed to result in any conceivable implementation. From the programmer's point of view, intentions are what would remain once the accidental details, as well as the notational encoding have been factored out.
A program is expressed as an abstract syntax tree (AST) where each node corresponds to an (instance of an) intention, and a method which describes the semantics of the intention by specifying the process of transforming the subtree headed by the intention instance into a tree containing only primitive executable nodes with fixed semantics.

IP is not a programming language. What makes it similar to the idea of a conceptual editor is the view that programs are not made up of syntactic constructs, but of conceptual entities (intentions) that have a meaning and a purpose that goes beyond their concrete syntax, and that these are the building blocks the programmer should be thinking about.

Simonyi also describes a prototype IP environment. Here, however, the differences are substantial. The IP editor is essentially a pure structure editor manipulating the IP tree. Temporarily invalid program descriptions are allowed, and apparently general cut and paste operations are provided. Furthermore, the paper does not discuss in detail how the intentions are to be described, but the examples provided seem to use a C-like language. The idea of using a mathematical language to describe models and behavior is not mentioned. Also, the paper does not provide any description or picture of the used representation(s), so it is hard to judge the quality of the editor's interface.

4.4 Productivity Tools as Editors

It is interesting to observe that designers of productivity tools such as word processors, spreadsheets, and presentation software have apparently recognized the importance of a high-level conceptual model of the edited object and of the editing process.
For instance, Microsoft PowerPoint [90], a program to build presentations, defines a presentation as a conceptual entity, made up of other conceptual entities, the slides, each of which has its own structure. This program allows the user to view the presentation in different conceptual ways, e.g., as an outline, or as a sequence of slides.

Microsoft Excel [15], a spreadsheet application, allows the user to organize data on one or more worksheets in a workbook. A worksheet is a grid of rows and columns. Each box in a worksheet (called a cell) can contain a piece of information such as a text value, a numeric value, or a formula that calculates a value. The user can arrange data in a worksheet and can also display it graphically in charts. However, it is not possible to edit the data in these graphical views. Excel presents a consistent view to the user, and never allows the data to be edited or manipulated outside the constraints of a worksheet.

FrameMaker [22], a word processor, allows the user to edit a book, where a book is viewed as a sequence of chapters, a chapter as a sequence of sections, a section as a sequence of subsections, and so on down to paragraphs. Each paragraph has a specific style that determines how the paragraph is formatted, what font is used, the spacing, etc. So the user can edit a document at a conceptual level, with the main advantage that the organization of the document is basically taken care of by the tool. However, FrameMaker also provides a textual view. This means that it is possible to edit a document in FrameMaker as plain text. So the user can, for example, override the structural properties of a book or of a paragraph by simply selecting a chunk of text (with no relationship to the conceptual model of the document) and applying some kind of modifier. This clearly compromises the conceptual view with the result that FrameMaker may lose its ability to guarantee the document's uniformity and integrity.
Finally, although this dissertation has essentially argued in favor of domain specific editing environments, where knowledge of the domain’s conceptual model can be used to bridge the gap between the edited descriptions and the user's mental model, it is worth briefly mentioning a tool that has taken a radically different approach.

MacSTILE [95], a schematic diagramming and system design tool, allows the design of systems of components (parts) according to an engineering design metaphor. Parts are characterized by an external interface that determines the part's external appearance and behavior, and by an internal description of how the part is constructed from other components. MacSTILE is not committed to any specific application domain. This makes it a tool of very general applicability. But, although it is possible to interpret the boxes, ports, and links in the diagrams in any appropriate application-dependent way, the generality of the tool (and its lack of domain specific knowledge) prevents it from providing anything beyond purely syntactic checks on the structure of the diagrams. Any domain specific semantic information is usually encoded in plain text that the user has to interpret.

4.5 Chapter Summary

In this chapter we have reviewed the main approaches to program editor design, and pointed out why they fall short of the requirements of a conceptual editor. We have also compared intentional programming as a way of building software and a prototype IP editor to the conceptual editor's own model of software construction. Finally we noted that editing environments in other domains such as word-processing and presentation software seem to be moving in the direction of trying to provide a high-level, conceptual view of the edited structures.
CHAPTER 5

CONCLUSIONS

In this chapter we review the ideas explored in this dissertation and the conclusions drawn from them. We summarize the contributions, and explore possible directions for future research.

5.1 Summary

Traditionally program editors have been designed as tools to enter and modify program descriptions without much concern about what these descriptions stand for and what they mean. As a result, most existing editors allow manipulations of program descriptions as if they were character strings with no attention to their meaning in the realm of programs. The user of the editor (the programmer) is allowed to build invalid descriptions and bears full responsibility for fixing any and all problems and inconsistencies.

Donald Norman, in his research in human-machine interaction, has explored important issues in the design and usability of artifacts and systems. This research has resulted in several important principles and guidelines that should be the bedrock of any good design.
This dissertation explores the implications of Norman's research for the design of program editors. It defends the following thesis:

(i) Norman's conclusions suggest that program editors should be designed with different goals from existing program editors.

In Chapter 2 we argue that since program editors convey a model of programs and program structure to the programmer, they have an impact on how the programmer views and builds software. Therefore they have the responsibility to help the programmer form an appropriate model of software, and should be designed with the goal of supporting, promoting, and, if necessary, enforcing a high-level consistent conceptual model of programs and program construction.

(ii) Conceptual program editors—editors designed with these new goals in mind—are substantially different from existing approaches.

Chapter 2 also describes the impact of several of Norman's design principles on the design program editors. It defines some specific requirements that will have to be satisfied by the editor to meet the demands of the new role. In reviewing related work, Chapter 4 summarizes existing approaches to program editor design, and points out their limitations and why they fall short of the conceptual editor's requirements and goals.

(iii) It is possible and desirable to formally model the conceptual entities that make up programs and to formally specify conceptual editors for them.

Chapter 3 motivates the need for a precise, unambiguous description of a conceptual model of programs and of the functionality of a conceptual editor for them. It shows the feasibility of the approach by modeling a small but significant set of program conceptual entities, and by specifying a primitive, yet functionally complete,
conceptual editor for them. It also argues that both the model and the editor could be
generalized and extended to model the concepts of a complete programming language
and to formally specify a full-fledged conceptual editor for such a language.

5.2 Conclusions

Program editors convey to the programmer a model of software and software
construction. So they have an impact on how the programmer thinks about and builds
programs. Therefore editors should be designed to support, promote, and enforce a high-
level, consistent conceptual model of programs and program construction.

Conceptual editors must be designed around an appropriate conceptual model of
software. This differentiates them from most existing editors, which are designed around a
chosen programming language instead of a conceptual model of programming.

Three other key properties characterize conceptual editors, and clearly distinguish
them from existing structure editors and other language-based environments:

1. **Validity of the edited description.** To preserve the consistency of the conceptual
   model at all times, it is necessary to make sure that the edited description is valid at all
times, i.e., conceptually meaningful at all times. This is has an impact on the
operations allowed by the editor and on the interaction between the user and the editor.

2. **The chosen operations.** Only conceptually meaningful operations are allowed. They
   must make sense in the realm of the conceptual model of programs supported by the
editor. They must be conceptually simple and not involve far reaching side effects that
would strain the user's ability to keep track of detail. This effectively rules out
operations such general conceptual cut and paste, that in most cases affect many,
widely-separated dependencies among involved entities.
3. The chosen representation(s). The way program conceptual entities are represented by the editor affects the user's perception of the meaning of the descriptions and of the manipulations that should be allowed. In other words, the representation affects the mental model of program descriptions that users form as a result of interaction with the editor. The wrong representation may doom the editor to failure simply because the programmer finds it too restrictive or unnatural. On the other hand, a "flexible" purely textual representation is unlikely to promote any high-level conceptual model of programs.

In the design of a conceptual editor it is desirable to provide a precise description of the conceptual entities and of the functionality that should be supported, independently of the design of the user interface and of the actual implementation. This allows the designer to concentrate on the key issue of supporting a chosen conceptual model of programs one piece at a time. First, model the conceptual entities and specify the operations. Then, use this unambiguous description to design the system image, i.e., choose the appropriate representation(s) for each conceptual entity, and to design the human-computer interaction required by each editor operation.

Providing a precise (formal) description of conceptual entities and of the behavior of the operations can be a difficult task, but we have shown that it is feasible, at least in principle. Formal descriptions are precise and unambiguous, as opposed to informal descriptions that are neither. Formal descriptions can be constructed so that only the essential properties of the model are captured (as we have done in Chapter 3 for type expressions and type declarations).
5.3 Contributions

The primary contributions of this research are the definition of a new, more fundamental role for program editors and the notion of conceptual editors as the answer to the demands and responsibilities of this enhanced role. The program editor is not just a tool to build and manipulate program descriptions but also a tool designed and expected to help the programmer form and nurture an appropriate model of software and software construction. Conceptual editors are not just another variation on the theme of language-based structure editors. Although conceptual editors enforce the syntactic and static semantic validity of the edited program description, this is only part of what makes an editor “conceptual”. In this dissertation we have provided the definition of and requirement analysis for this new class of program editors.

We have also addressed some of the ideas and issues in the design of conceptual editors by formally modeling conceptual entities and specifying the functionality of an editor for type declarations. This has served as proof of concept that it is feasible to formally specify a conceptual editor with the advantages that this entails:

- It provides a precise, unambiguous description of the intended conceptual model of program structure.
- It separates the role of the designer from that of the implementors of the interface and the functionality.
- It allows the designer to choose the editor’s operations based on the conceptual model and not on the used representation.
- It allows the implementation and separate evaluation of multiple views/interfaces (and also multiple implementations of the editor’s internal data structures and functionality).

168
• It provides the necessary framework to support formal reasoning and verification.

As a secondary result, we have demonstrated the expressive power of the RESOLVE mathematical language, through one of the largest and most complex specification efforts in RESOLVE to date.

The design and implementation of the type declaration editor has also resulted in the design of several general-purpose reusable components. These components, in addition to being useful in the implementation of conceptual editors, should also be particularly beneficial in the design and development of other language-based applications.

5.4 Future Research

There is still plenty of room for research on this topic. First and foremost, it would be interesting to build a conceptual editor for a complete language and to experimentally test its impact on programmers. A conceptual editor can be viewed as a composition of separate but integrated tools that allow the programmer to perform one of many conceptually distinct tasks. Thus, it should be possible to build it one tool at a time, and test and improve each tool in isolation, before integrating them into one application programming environment.

Three very interesting aspects of the editor's design need further research and are bound to have the most direct impact on the success of a conceptual editor:

• A complete conceptual model of programs and program construction. In Chapter 2 we have provided a glimpse into the RESOLVE conceptual model of software. RESOLVE is the result of a 10 year research effort by the Reusable Software Research Group at The Ohio State University. Many interesting issues are still under study, such as
defining the useful and desirable relationships among software components and their meanings, and establishing the role and full potential of templates in software construction.

- Careful design of the user interface, with particular attention to the used notations and an appropriate combination of graphical and textual information. As we have repeatedly pointed out, the choice of the representation(s) and of the user-computer interaction is crucial to the success or failure of a conceptual editor. Although we have notations for some of the entities and activities involved in the program construction process (e.g., the component coupling diagrams and instantiation diagrams used in the examples in Chapter 2), they have not been tested as programming notations in interactive tools. There is also the challenging issue of dealing with both the programming notation and the mathematical notation, which are both part of RESOLVE and our conceptual model of software.

- Choice of an appropriate set of operations: powerful enough not to hinder or frustrate programmers, and yet conceptually simple enough to allow programmers (possibly with the editor’s help) to understand what’s going on. As we have seen in Chapter 2, some of the operations that seem universally accepted as necessary in less structured editing environments (e.g., cut and paste) have serious problems from a conceptual point of view. It is clear that if these operations are omitted from a conceptual editor, other suitable operations must be provided to facilitate the programmer’s editing task.

At the experimental stage, some of the hypotheses implicitly stated in this dissertation that would be interesting to test include:
• The program editor does have an impact on how programmers think. More specifically, a conceptual editor will help novices form an appropriate model of software and software construction.

• It is possible to build a usable editor that enforces strict validity constraints on the edited representation required by a conceptual editor.

• Using a conceptual editor reduces the number of bugs in the code. (Since programmers think in conceptual terms at all times and have no concern about the validity of the edited description, they can concentrate on the real task of producing software with required behavior).

Finally, it would be interesting to continue and extend the work started in Chapter 3 in the ways described in Section 3.7. Modeling other conceptual entities and specifying more powerful operations are the primary candidates for further work.

Since the models and specifications used by the Type_Declaration_Editor are already of considerable complexity, it would be worthwhile exploring how far the TREE and NESTED_MAP models can go in modeling other conceptual entities. It is not desirable to introduce a new theory for every new conceptual entity, and it should not be necessary since the most common parts of program descriptions are nested, hierarchical structures that hopefully can be adequately modeled by one of the mathematical structures used in this dissertation.
APPENDIX A

NESTED LIST

This appendix contains the math modules and concepts relating to the Nested_List components. It includes:

- TREE_ENHANCED_MACHINERY
- Nested_List_Template
- TREE_POSITION_MACHINERY
- Nested_List_With_Position_Template
A.1 TREE_ENHANCED_MACHINERY

mathematics TREE_ENHANCED_MACHINERY

context

global context

mathematics TREE THEORY TEMPLATE

parametric context

math type LABEL

math facility TREE FACILITY is
TREE THEORY TEMPLATE (LABEL)

interface

math operation EXTRACT_LABEL ( 
  t : TREE 
): LABEL
  implicit definition
  t = COMPOSE (EXTRACT_LABEL (t), EXTRACT_CHILDREN (t))

math operation EXTRACT_CHILDREN ( 
  t : TREE 
): string of TREE
  implicit definition
  t = COMPOSE (EXTRACT_LABEL (t), EXTRACT_CHILDREN (t))

math operation SIZE ( 
  t : TREE 
): integer
  implicit definition
  if t = EMPTY_TREE 
  then 
    SIZE (t) = 0 
  else 
    SIZE (t) = 1 + SIZE (EXTRACT_CHILDREN (t))

math operation SIZE ( 
  s : string of TREE 
): integer
  implicit definition
  if s = empty_string 
  then 
    SIZE (s) = 0 
  else 
    SIZE (s) = sum n: integer 
    where (0 <= n < |s|) 
    (SIZE (nth (s, n)))

end TREE_ENHANCED_MACHINERY
A.2 Nested_List_Template

concept Nested_List_Template

context

global context

mathematics TREE_THEORY_TEMPLATE
mathematics TREE_ENHANCED_MACHINERY
facility Standard_Integer_Facility

parametric context

type Label

local context

math facility TREE_FACILITY is
    TREE_THEORY_TEMPLATE (math[Label])

math facility TREE_ENHANCED_FACILITY is
    TREE_ENHANCED_MACHINERY (math[Label], TREE_FACILITY)

interface

type Nested_List is modeled by TREE
exemplar nl
initialization
    ensures
        nl = EMPTY_TREE

operation New (
    alters nl: Nested_List
    consumes label: Label
)
    requires
        nl = EMPTY_TREE
    ensures
        nl = COMPOSE (#label, empty_string)

operation Swap_Label (    
    alters nl: Nested_List
    alters label: Label
)
    requires
        nl /= EMPTY_TREE
    ensures
        #nl = COMPOSE (label, EXTRACT_CHILDREN (#nl)) and
        nl = COMPOSE (#label, EXTRACT_CHILDREN (#nl))
operation Add {
    alters nl: Nested_List
    preserves pos: Integer
    consumes l: Nested_List
}
requires
0 <= pos <= |EXTRACT_CHILDREN (nl)|
ensures
there exists a, b: string of TREE
(|a| = pos and
EXTRACT_CHILDREN (#nl) = a * b and
nl = COMPOSE (EXTRACT_LABEL (#nl), a * <#1> * b))

operation Remove {
    alters nl: Nested_List
    preserves pos: Integer
    produces l: Nested_List
}
requires
0 <= pos < |EXTRACT_CHILDREN (nl)|
ensures
there exists a, b: string of TREE
(|a| = pos and
EXTRACT_CHILDREN (#nl) = a * <1> * b and
nl = COMPOSE (EXTRACT_LABEL (#nl), a * b))

operation Number_Of_Children {
    preserves nl: Nested_List
} : Integer
ensures
Number_Of_Children = |EXTRACT_CHILDREN (nl)|

operation Size {
    preserves nl: Nested_List
} : Integer
ensures
Size = SIZE (nl)

end Nested_List_Template
A.3 TREE_POSITION_MACHINERY

mathematics TREE_POSITION_MACHINERY

context

global context

mathematics TREE_THEORY_TEMPLATE

parametric context

math type LABEL

math facility TREE_FACILITY is
TREE_THEORY_TEMPLATE (LABEL)

interface

math subtype SITE is {
  label: LABEL
  left, right: string of TREE
}

math subtype TREE_WITH_POSITION is {
  position: string of SITE
  current: TREE
}

math operation EMPTY_TREE_WITH_POSITION : TREE_WITH_POSITION
  explicit definition
  (empty_string, EMPTY_TREE)

math operation SUTURE ( ss: string of SITE
  t: TREE
): TREE
  implicit definition
  if ss = empty_string
  then
    SUTURE (ss, t) = t
  else
    SUTURE (ss, t) =
    SUTURE (all_but_last (ss),
      COMPOSE (last (ss).label,
        last (ss).left * <t> * last (ss).right)

math operation RESET ( twp: TREE_WITH_POSITION
  ): TREE
  explicit definition
  SUTURE (twp.position, twp.current)

end TREE_POSITION_MACHINERY
A.4 Nested_List_With_Position_Template

concept Nested_List_With_Position_Template

context

global context

  mathematics TREE_THEORY_TEMPLATE
  mathematics TREE_ENHANCED_MACHINERY
  mathematics TREE_POSITION_MACHINERY
  facility Standard_Integer_Facility

parametric context

  type Label

local context

  math facility TREE_FACILITY is
    TREE_THEORY_TEMPLATE (math[Label])

  math facility TREE_ENHANCED_FACILITY is
    TREE_ENHANCED_MACHINERY (math[Label], TREE_FACILITY)

  math facility TREE_POSITION_FACILITY is
    TREE_POSITION_MACHINERY (math[Label], TREE_FACILITY)

interface

  type Nested_List_With_Position is modeled by TREE_WITH_POSITION
  exemplar nlwp
  initialization
    ensures
      nlwp = EMPTY_TREE_WITH_POSITION
  operation New ( alters nlwp: Nested_List consumes label: Label )
  requires
    nlwp.current = EMPTY_TREE
  ensures
    nlwp = (#nlwp.position, COMPOSE (#label, empty_string))
operation \textit{Swap\_Label} (  
  alters mlwp: \text{Nested\_List\_With\_Position}  
  alters label: Label  
)  
\textit{requires}  
mlwp.current /= \text{EMPTY\_TREE}  
\textit{ensures}  
\#mlwp.current =  
\text{COMPOSE} (label, \text{EXTRACT\_CHILDREN} (\#mlwp.current))  
mlwp = (\#mlwp.position,  
\text{COMPOSE} (\#label, \text{EXTRACT\_CHILDREN} (\#mlwp.current)))  

operation \textit{Add} (  
  alters mlwp: \text{Nested\_List\_With\_Position}  
  preserves pos: Integer  
  consumes Iwp: \text{Nested\_List\_With\_Position}  
)  
\textit{requires}  
0 \leq pos \leq |\text{EXTRACT\_CHILDREN} (mlwp.current)|  
\textit{ensures}  
mlwp.position = \#mlwp.position  
\textit{there exists} a, b: \text{string of TREE}  
\{|a| = pos \text{ and } \text{EXTRACT\_CHILDREN} (\#mlwp.current) = a \ast b \text{ and } mlwp.current = \text{COMPOSE} (\text{EXTRACT\_LABEL} (\#mlwp.current), a \ast \text{\texttt{<RESET}} (\#lwp) \ast b)\}  

operation \textit{Remove} (  
  alters mlwp: \text{Nested\_List\_With\_Position}  
  preserves pos: Integer  
  produces Iwp: \text{Nested\_List\_With\_Position}  
)  
\textit{requires}  
0 \leq pos < |\text{EXTRACT\_CHILDREN} (mlwp.current)|  
\textit{ensures}  
mlwp.position = \#mlwp.position  
\textit{there exists} a, b: \text{string of TREE}  
\{|a| = pos \text{ and } \text{EXTRACT\_CHILDREN} (\#mlwp.current) = a \ast \text{\texttt{<RESET}} (lwp) \ast b \text{ and } lwp.position = \text{empty\_string} \text{ and } mlwp.current = \text{COMPOSE} (\text{EXTRACT\_LABEL} (\#mlwp.current), a \ast b)\}  

operation \textit{Number\_Of\_Children} (  
  preserves mlwp: \text{Nested\_List\_With\_Position}  
) : Integer  
\textit{ensures}  
\text{Number\_Of\_Children} = |\text{EXTRACT\_CHILDREN} (mlwp.current)|
operation Size (  
    preserves nlwp: Nested_List_With_Position  
 ) : Integer  
ensures  
    Size = SIZE (RESET (nlwp))

operation Open (  
    alters   nlwp: Nested_List_With_Position  
    consumes pos: Integer  
 )  
requires  
    0 <= pos < |EXTRACT_CHILDREN (nlwp.current)|  
ensures  
    there exists s: SITE  
    (nlwp.position = #nlwp.position * <s> and  
    SUTURE (<s>, nlwp.current) = #nlwp.current and  
    |s.left| = #pos)

operation Close (  
    alters   nlwp: Nested_List_With_Position  
    produces pos: Integer  
 )  
requires  
    nlwp.position /= empty_string  
ensures  
    there exists s: SITE  
    (#nlwp.position = nlwp.position * <s> and  
    SUTURE (<s>, #nlwp.current) = nlwp.current and  
    pos = |s.left|)

operation Reset (  
    alters   nlwp: Nested_List_With_Position  
 )  
ensures  
    nlwp.position = empty_string and  
    nlwp.current = RESET (#nlwp)

operation Size_Of_Current (  
    preserves nlwp: Nested_List_With_Position  
 ) : Integer  
ensures  
    Size_Of_Current = SIZE (nlwp.current)

operation Position_Length (  
    preserves nlwp: Nested_List_With_Position  
 ) : Integer  
ensures  
    Position_Length = |nlwp.position|

end Nested_List_With_Position_Template
APPENDIX B

NESTED MAP SET

This appendix contains the math modules and concepts relating to the Nested_Map_Set components. It includes:

- NESTED_MAP_ENHANCED_MACHINERY
- Nested_Map_Set_Template
- NESTED_MAP_SET_POSITION_MACHINERY
- Nested_Map_Set_With_Position_Template
- NESTED_MAP_PATH_MACHINERY
- NESTED_MAP_EXTRACT_PATH_MACHINERY
- Nested_Map_Set_Path_Template
B.1 NESTED_MAP_ENHANCED_MACHINERY

context

global context

mathematics NESTED_MAP_THEORY_TEMPLATE

parametric context

math type ID

math type LABEL

math facility NESTED_MAP_FACILITY is

NESTED_MAP_THEORY_TEMPLATE (ID, LABEL)

interface

math operation EXTRACT_LABEL (nm: NESTED_MAP) : LABEL
implicit definition
nm = COMPOSE (EXTRACT_LABEL (nm), EXTRACT_CHILDREN (nm))

math operation EXTRACT_CHILDREN (nm: NESTED_MAP) : NESTED_MAP_SET
implicit definition
nm = COMPOSE (EXTRACT_LABEL (nm), EXTRACT_CHILDREN (nm))

math operation SIZE (nm: NESTED_MAP) : integer
implicit definition
if nm = EMPTY_NESTED_MAP then SIZE (nm) = 0
else SIZE (nm) = 1 + SIZE (EXTRACT_CHILDREN (nm))

math operation SIZE (nms: NESTED_MAP_SET) : integer
implicit definition
if nms = EMPTY_NESTED_MAP_SET then SIZE (nms) = 0
else SIZE (nms) = sum id: ID where (id is in DOMAIN (nms))
    (SIZE (APPLY (nms, id)))

end NESTED_MAP_ENHANCED_MACHINERY
B.2 Nested_Map_Set_Template

concept Nested_Map_Set_Template

case

global context

mathematics NESTED_MAP_THEORY_TEMPLATE
mathematics NESTED_MAP_ENHANCED_MACHINERY
facility Standard_Integer_Facility
facility Standard_Boolean_Facility

parametric context

type Id

type Label

local context

math facility NESTED_MAP_FACILITY is
  NESTED_MAP_THEORY_TEMPLATE (math[Id], math[Label])

math facility NESTED_MAP_ENHANCED_FACILITY is
  NESTED_MAP_ENHANCED_MACHINERY (math[Id], math[Label], NESTED_MAP_FACILITY
  )

interface

type Nested_Map_Set is modeled by NESTED_MAP_SET

exemplar nms

initialization
  ensures
  nms = EMPTY_NESTED_MAP_SET

operation Define (  
  alters nms: Nested_Map_Set
  consumes id: Id
  consumes label: Label
  consumes nm: Nested_Map_Set
)

requires
  not DEFINED_IN (nms, id)

ensures
  nms = #nms union {(#id, COMPOSE (#label, #nm))}
operation Undefine (  
    alters nms: Nested_Map_Set  
    preserves id: Id  
    produces id_copy: Id  
    produces label: Label  
    produces nm: Nested_Map_Set  
)

requires  
    DEFINED_IN (nms, id)

ensures  
    (id, COMPOSE (label, nm)) is in #nms and  
    nms = #nms - { (id, COMPOSE (label, nm)) } and  
    id_copy = id

operation Undefine_Any (  
    alters nms: Nested_Map_Set  
    produces id: Id  
    produces label: Label  
    produces nm: Nested_Map_Set  
)

requires  
    nms /= empty_set

ensures  
    (id, COMPOSE (label, nm)) is in #nms and  
    nms = #nms - { (id, COMPOSE (label, nm)) }

operation Is_DEFINED (  
    preserves nms: Nested_Map_Set  
    preserves id: Id  
)

ensures  
    Is_DEFINED iff DEFINED_IN (nms, id)

operation Size_Of_Current (  
    preserves nms: Nested_Map_Set  
)

ensures  
    Size_Of_Current = |nms|

operation Size (  
    preserves nms: Nested_Map_Set  
)

ensures  
    Size = SIZE (nms)
B.3 NESTED_MAP_SET_POSITION_MACHINERY

**mathematics** NESTED_MAP_SET_POSITION_MACHINERY

context

**global context**

**mathematics** NESTED_MAP_THEORY_TEMPLATE

parametric context

**math type** ID

**math type** LABEL

**math facility** NESTED_MAP_FACILITY is
NESTED_MAP_THEORY_TEMPLATE (ID, LABEL)

interface

**math subtype** SITE is (  
id: ID  
label: LABEL  
rest: NESTED_MAP_SET
)

**exemplar** s

**constraint**
not DEFINED_IN (s.rest, s.id)

**math subtype** NESTED_MAP_SET_WITH_POSITION is (  
position: string of SITE  
current: NESTED_MAP_SET
)

**math operation** EMPTY_NESTED_MAP_SET_WITH_POSITION:

**explicit definition**

(EMPTY_NESTED_MAP_SET)
math operation SUTURE {
    ss: string of SITE
    nms: NESTED_MAP_SET
}: NESTED_MAP_SET

implicit definition
if ss = empty_string
then
    SUTURE (ss, nms) = nms
else
    SUTURE (ss, nms) =
    SUTURE (all_but_last (ss),
    last (ss).rest union
    ((last (ss).id, COMPOSE (last (ss).label, nms)))

math operation RESET {
    nmswp: NESTED_MAP_SET_WITH_POSITION
}: NESTED_MAP_SET

explicit definition
SUTURE (nmswp.position, nmswp.current)

end NESTED_MAP_SET_POSITION_MACHINERY
B.4 Nested_Map_Set_With_Position_Template

concept Nested_Map_Set_With_Position_Template

context

global context

mathematics NESTED_MAP_THEORY_TEMPLATE
mathematics NESTED_MAP_ENHANCED_MACHINERY
mathematics NESTED_MAP_SET_POSITION_MACHINERY
facility Standard_Integer_Facility
facility Standard_Boolean_Facility

parametric context

type Id

type Label

local context

math facility NESTED_MAP_FACILITY is
NESTED_MAP_THEORY_TEMPLATE (math[Id], math[Label])

math facility NESTED_MAP_ENHANCED_FACILITY is
NESTED_MAP_ENHANCED_MACHINERY {
    math[Id], math[Label], NESTED_MAP_FACILITY
}

math facility NESTED_MAP_SET_POSITION_FACILITY is
NESTED_MAP_SET_POSITION_MACHINERY {
    math[Id], math[Label], NESTED_MAP_FACILITY
}

interface

type Nested_Map_Set_With_Position is modeled by

exemplar nms
initialization
ensures
    nms = EMPTY_NESTED_MAP_SET_WITH_POSITION

186
operation Define (  
  alters nms: Nested_Map_Set-With_Position  
  consumes id: Id  
  consumes label: Label  
  consumes nm: Nested_Map_Set-With_Position  
)
requires  
  not DEFINED_IN (nms.current, id)
ensures  
  nms.position = #nms.position and  
  (nms.current = #nms.current union  
    {(#id, COMPOSE (#label, SUTURE (#nm.position, #nm.current)))})

operation Undefined (  
  alters nms: Nested_Map_Set-With_Position  
  preserves id: Id  
  produces id_copy: Id  
  produces label: Label  
  produces nm: Nested_Map_Set-With_Position  
)
requires  
  DEFINED_IN (nms.current, id)
ensures  
  ( (id, COMPOSE (label, SUTURE (run.position, r u n . current ) )  )  is in  
    #nms. current)  and  
  (  r u n s . current = # r u n s . current -  
    { (id, COMPOSE (label, SUTURE (run.position, r u n . current) ) ) }  )  and  
  id_copy = id  and  
  nms.position = #nms.position and  
  nm.position = empty_string

operation Undefined_Any (  
  alters nms: Nested_Map_Set-With_Position  
  produces id: Id  
  produces label: Label  
  produces nm: Nested_Map_Set-With_Position  
)
requires  
  nms.current /= empty_set
ensures  
  ( (id, COMPOSE (label, SUTURE (run.position, r u n . current ) )  )  is in  
    #nms. current)  and  
  (  r u n s . current = # r u n s . current -  
    { (id, COMPOSE (label, SUTURE (run.position, r u n . current) ) ) }  )  and  
  nm.position = #nms.position and  
  nm.position = empty_string
operation Is_Defined {  
    preserves nms: Nested_Map_Set_With_Position  
    preserves id: Id  
    }: Boolean  
ensure  
    Is_Defined iff DEFINED_IN (nms.current, id)  
}

operation Size_Of_Current {  
    preserves nms: Nested_Map_Set_With_Position  
    }: Integer  
ensure  
    Size_Of_Current = |nms.current|  
}

operation Open {  
    alters nms: Nested_Map_Set_With_Position  
    consumes id: Id  
    }  
require  
    DEFINED_IN (nms.current, id)  
ensure  
    there exists s: SITE  
    (nms.position = #nms.position * <s> and  
     SUTURE (<s>, nms.current) = #nms.current)  
}

operation Close {  
    alters nms: Nested_Map_Set_With_Position  
    produces id: Id  
    }  
require  
    nms.position /= empty_string  
ensure  
    there exists s: SITE  
    (#nms.position = nms.position * <s> and  
     SUTURE (<s>, #nms.current) = nms.current and id = s.id)  
}

operation Reset {  
    alters nms: Nested_Map_Set_With_Position  
    }  
ensure  
    nms.position = empty_string and  
    nms.current = RESET (#nms)  
}

operation Size {  
    preserves nms: Nested_Map_Set_With_Position  
    }: Integer  
ensure  
    Size = SIZE (RESET (nms))
operation Position_Length {
  preserves nms: Nested_Map_Set_With_Position
  ): Integer
  ensures
  Position_Length = |nms.position|
end Nested_Map_Set_With_Position_Template
B.5 NESTED_MAP_PATH_MACHINERY

mathematics NESTED_MAP_PATH_MACHINERY

context

global context

  mathematics NESTED_MAP_THEORY_TEMPLATE
  mathematics NESTED_MAP_OTHER_MACHINERY

parametric context

  math type ID
  math type LABEL

  math facility NESTED_MAP_THEORY is
    NESTED_MAP_THEORY_TEMPLATE (ID, LABEL)

  math facility NESTED_MAP_OTHER_FACILITY is
    NESTED_MAP_OTHER_MACHINERY (ID, LABEL, NESTED_MAP_THEORY)

interface

  math subtype PATH is string of ID

  math operation DEFINED_IN (nms: NESTED_MAP_SET, p: PATH): boolean
  implicit definition
    (p = empty_string) or
    (DEFINED_IN (nms, first (p)) and
     DEFINED_IN (EXTRACT_CHILDREN (APPLY (nms, first (p))),
                       all_but_first (p)))

end NESTED_MAP_PATH_MACHINERY
B.6 NESTED_MAP_EXTRACT_PATH_MACHINERY

mathematics NESTED_MAP_EXTRACT_PATH_MACHINERY

context

global context

mathematics NESTED_MAP_THEORY_TEMPLATE
mathematics NESTED_MAP_OTHER_MACHINERY
mathematics NESTED_MAP_PATH_MACHINERY
mathematics NESTED_MAP_SET_POSITION_MACHINERY

parametric context

math type ID

math type LABEL

math facility NESTED_MAP_THEORY is
NESTED_MAP_THEORY_TEMPLATE (ID, LABEL)

math facility NESTED_MAP_OTHER_FACILITY is
NESTED_MAP_OTHER_MACHINERY (ID, LABEL, NESTED_MAP_THEORY)

math facility NESTED_MAP_PATH_FACILITY is
NESTED_MAP_PATH_MACHINERY (ID, LABEL, NESTED_MAP_THEORY, NESTED_MAP_OTHER_FACILITY)

math facility NESTED_MAP_SET_POSITION_FACILITY is
NESTED_MAP_SET_POSITION_MACHINERY (ID, LABEL, NESTED_MAP_THEORY)

interface

math operation EXTRACT_PATH (
    position: string of SITE
): PATH

implicit definition
    if position = empty_string
    then
        EXTRACT_PATH (position) = empty_string
    else
        EXTRACT_PATH (position) =
            <first (position).id> *
        EXTRACT_PATH (all_but_first (position))

end NESTED_MAP_EXTRACT_PATH_MACHINERY
B.7 Nested_Map_Set_Path_Template

concept Nested_Map_Set_Path_Template

context

global context

mathematics NESTED_MAP_THEORY_TEMPLATE
mathematics NESTED_MAP_OTHER_MACHINERY
mathematics NESTED_MAP_PATH_MACHINERY
mathematics NESTED_MAP_SET_POSITION_MACHINERY
mathematics NESTED_MAP_EXTRACT_PATH_MACHINERY
facility Standard_Boolean_Facility
concept Nested_Map_Set_With_Position_Template

parametric context

type Id

type Label

facility Nested_Map_Set_With_Position_Facility is
  Nested_Map_Set_With_Position_Template (Id, Label)

local context

math facility NESTED_MAP_THEORY is
  NESTED_MAP_THEORY_TEMPLATE (math[Id], math[Label])

math facility NESTED_MAP_OTHER_FACILITY is
  NESTED_MAP_OTHER_MACHINERY (math[Id], math[Label], NESTED_MAP_THEORY)

math facility NESTED_MAP_PATH_FACILITY is
  NESTED_MAP_PATH_MACHINERY (math[Id], math[R_Item], NESTED_MAP_THEORY, NESTED_MAP_OTHER_FACILITY)

math facility NESTED_MAP_SET_POSITION_FACILITY is
  NESTED_MAP_SET_POSITION_MACHINERY (math[Id], math[R_Item], NESTED_MAP_THEORY)
math facility NESTED_MAP_EXTRACT_PATH_FACILITY is
NESTED_MAP_EXTRACT_PATH_MACHINERY (  
math[Id], math[R_Item], NESTED_MAP_THEORY,  
NESTED_MAP_OTHER_FACILITY, NESTED_MAP_PATH_FACILITY,  
NESTED_MAP_SET_POSITION_FACILITY  
)

interface

type Path is modeled by PATH  
exemplar p  
initialization  
ensures  
p = empty_string  

operation Set_Current_Position (  
alters nm: Nested_Map_Set_With_Position  
preserves p: Path  
)  
requires  
DEFINED_IN (SUTURE (#nm.position, #nm.current), p)  
ensures  
(SUTURE (nm.position, nm.current) =  
SUTURE (#nm.position, #nm.current)) and  
EXTRACT_PATH (nm.position) = p  

operation Get_Current_Position (  
preserves nm: Nested_Map_Set_With_Position  
produces p: Path  
)  
ensures  
p = EXTRACT_PATH (nm.position)  

operation Is_Defined (  
preserves nm: Nested_Map_Set_With_Position  
preserves p: Path  
): Boolean  
ensures  
Is_Defined iff DEFINED_IN (SUTURE (nm.position, nm.current), p)  

operation Is_Subpath (  
preserves pl: Path  
preserves p2: Path  
): Boolean  
ensures  
Is_Subpath iff there exists p: PATH (pl * p = p2)  

end Nested_Map_Set_Path_Template
APPENDIX C

AN IMPLEMENTATION FOR THE TYPE DECLARATION EDITOR

This appendix contains the implementation modules for the type declaration editor specified in Chapter 3. It includes:

- ID_Template
- realization header 1 for Type_Declaration_Editor
- realization body 1 for Type_Declaration_Editor

For a description of the RESOLVE implementation language see [33, 7]. For a description of the components Record_Template, Sequence_Template, Set_Template, and Partial_Map_Template see [105].
C.1 ID_Template

concept ID_Template

context

global context

facility Standard_Boolean_Facility

local context

math operation NIL: integer

state variables
  used: set of integer
  initialization
    ensures used = {NIL}

interface

type ID is modeled by integer
  exemplar id
  initialization
    ensures id = NIL

operation New_ID: ID
  referenced state variables
  alters used
  ensures New_ID is not in #used and
    used = #used union {New_ID}

operation Are_Equal (  
  preserves id1: ID  
  preserves id2: ID  
): Boolean

  ensures Are_Equal iff id1 = id2

operation Replica (  
  preserves id: ID  
): ID

  ensures Replica = id

end ID_Template
C.2 Type_Declaration_Editor - Realization Header

realization header 1 for Type_Declaration_Editor

context
global context

concept Record_Template
concept Sequence_Template
concept Set_Template
concept Partial_Map_Template
concept Nested_List_With_Position_Template
concept Nested_Map_Set_With_Position_Template
concept Nested_Map_Set_Path_Template

parametric context

facility Type_Expression_Facility is
    Nested_List_With_Position_Template (ID2)

facility PP_Facility is
    Sequence_Template (ID2)

facility Label_Facility is
    Record_Template (ID2, Sequence, Nested_List_With_Position)

facility Type_Declaration_Set_Facility is
    Nested_Map_Set_With_Position_Template
        (ID1, Label_Facility.Record )

facility Type_Declaration_Set_Path_Facility is
    Nested_Map_Set_Path_Template
        (ID1, Label_Facility.Record, Type_Declaration_Set_Facility)

facility Current_Context_Facility is
    Set_Template (ID2)

facility TID_Info_Facility is
    Record_Template (Integer, Path)

facility Type_Context_Facility is
    Partial_Map_Template (ID2, TID_Info_Facility.Record)

facility Dependency_Count_Facility is
    Partial_Map_Template (ID2, Integer)
operation Are_Equal (  
    preserves id1: ID1  
    preserves id2: ID1  
): Boolean  
ensures  
    Are_Equal  iff  id1 = id2  

operation Replica (  
    preserves id: ID1  
): ID1  
ensures  
    Replica = id  

end I  

197
C.3 Type_Declaration_Editor - Realization Body

realization body I for Type_Declaration_Editor

context

local context

renaming
Nested_Map_Set_With_Position as Type_Declaration_Set
Nested_List_With_Position as Type_Expression
Label_Facility.Record as Label
Label_Facility.field1 as tid
Label_Facility.field2 as pp
Label_Facility.field3 as body
TID_Info_Facility.Record as TID_Info
TID_Info_Facility.field1 as arity
TID_Info_Facility.field2 as path
Type_Context_Facility.Partial_Map as TID_Info_Map
Dependency_Count_Facility.Partial_Map as Dependency_Count_Map
Sequence as Parameter_Profile
Set as Current_Context

math operation INITIAL_TYPE_CONTEXT : TYPE_CONTEXT
restriction
there exists tuple_id, boolean_id, integer_id,
character_id, string_id, set_id,
function_id : ID2

|INITIAL_TYPE_CONTEXT| = 8 and
INITIAL_TYPE_CONTEXT =

{(ID2_Facility.NIL, -1),
(tuple_id, -1),
(boolean_id, 0),
(integer_id, 0),
(character_id, 0),
(string_id, 1),
(set_id, 1),
(function_id, 2))

state variables
tdswp: Type_Declaration_Set
tewp: Type_Expression
tewp_id: ID1
body_is_open: Boolean
tid_info: TID_Info_Map
in_context: Current_Context
dependencies: Dependency_Count_Map
operation Count_Occurrences {
  preserves te: Type_Expression
  preserves tid: ID2
} : Integer
ensures
  Count_Occurrences = OCCURS_COUNT (te.current)
context
variables
  child: Type_Expression
  type_id: ID2
  count, pos: Integer
begin
  Swap_Label (te, type_id)
  if Are_Equal (tid, type_id) then
    count := 1
  end if
  Swap_Label (te, type_id)
  while pos < Number_Of_Children (te) do
    Remove (te, pos, child)
    count := count + Count_Occurrences (child, tid)
    Add (te, pos, child)
  end while
  return count
end Count_Occurrences

operation Initialize_Current_Context {
  produces context: Current_Context
  preserves info_map: TID_Info_Map
}
ensures
  context = DOMAIN (info_map)
context
variables
  tmp: TID_Info_Map
  tid, tid_copy: ID2
  info: TID_Info
begin
  Clear (context)
  while Size (info_map) > 0 do
    Undefine_Any (info_map, tid, info)
    tid_copy := Replica (tid)
    Add (context, tid_copy)
    Define (tmp, tid, info)
  end while
  info_map := tmp
end Initialize_Current_Context
operation Initialize_TID_Info_Map (
    produces info_map: TID_Info_Map
)
ensures
for all tid: ID2
    (if tid is in DOMAIN (INITIAL_TYPE_CONTEXT)
    then
        (tid, (APPLY (INITIAL_TYPE_CONTEXT, tid), empty_string))
        is in info_map
    else not DEFINED_IN (info_map, tid))
context
variables
    nil_id, tuple_id, boolean_id, integer_id: ID2
    character_id, string_id, set_id, function_id: ID2
    info: TID_Info
begin
    Clear (info_map)
    info.arity := -1
    Define (info_map, nil_id, info)
    tuple_id := New_ID()
    info.arity := -1
    Define (info_map, tuple_id, info)
    boolean_id := New_ID()
    info.arity := 0
    Define (info_map, boolean_id, info)
    integer_id := New_ID()
    info.arity := 0
    Define (info_map, integer_id, info)
    character_id := New_ID()
    info.arity := 0
    Define (info_map, character_id, info)
    string_id := New_ID()
    info.arity := 1
    Define (info_map, string_id, info)
    set_id := New_ID()
    info.arity := 1
    Define (info_map, set_id, info)
    function_id := New_ID()
    info.arity := 2
    Define (info_map, function_id, info)
end Initialize_TID_Info_Map
operation Add_PP_Context {
    alters context: Current_Context
    preserves pp: Parameter_Profile
}
requires
    |pp| = |elements (pp)| and
    for all tid: ID2 where (tid is in elements (pp))
        (tid is not in context)
ensures
    context = #context union elements (pp)
context
    variables
        tid, tid_copy: ID2
        pos: Integer
begin
    while pos < Length (pp) do
        Remove (pp, pos, tid)
        tid_copy := Replica (tid)
        Add (context, tid_copy)
        Add (pp, pos, tid)
        pos := pos + 1
    end while
end Add_PP_Context

operation Add_Current_Context {
    alters context: Current_Context
    preserves tds: Type_Declaration_Set
}
requires
    tds.position = empty_string and
    |tds.current| = |CURRENT_CONTEXT (tds.current)| and
    DOMAIN (CURRENT_CONTEXT (tds.current)) intersect context = empty_set
ensures
    context = #context union DOMAIN (CURRENT_CONTEXT (tds.current))
context
    variables
        tmp, child: Type_Declaration_Set
        id: ID1
        tid: ID2
        label: Label
begin
    while Size_Of_Current (tds) > 0 do
        Undefine_Any (tds, id, label, child)
        tid := Replica (label.tid)
        Add (context, tid)
        Define (tmp, id, label, child)
    end while
    tds :=: tmp
end Add_Current_Context
operation Remove_PP_Context (
    alters context: Current_Context
    preserves pp: Parameter_Profile
)
requires
    \(|pp| = |\text{elements} (pp)| \quad \text{and} \quad for\ \text{all}\ \text{tid: ID2 where (tid is in elements (pp))}
    (\text{tid is in context}) \ensures
    context = \#context without elements (pp)

context
variables
    \text{tid, tid\_copy: ID2}
    \text{pos: Integer}
begin
while pos < Length (pp) do
    Remove (pp, pos, tid)
    Remove (context, tid, tid\_copy)
    Add (pp, pos, tid)
    pos := pos + 1
end while
end Remove_PP_Context

operation Remove_Current_Context (
    alters context: Current_Context
    preserves tds: Type_Declaration_Set
)
requires
    tds.position = \text{empty\_string} \quad \text{and} \quad \text{|tds.current| = |CURRENT\_CONTEXT (tds.current)|} \quad \text{and} \quad \text{DOMAIN (CURRENT\_CONTEXT (tds.current)) is subset of context}
\ensures
    context = \#context without \text{DOMAIN (CURRENT\_CONTEXT (tds.current))}

context
variables
    \text{tmp, child: Type\_Declaration\_Set}
    \text{id: ID1}
    \text{tid: ID2}
    \text{label: Label}
begin
while Size\_Of\_Current (tds) > 0 do
    Undefine\_Any (tds, id, label, child)
    Remove (context, label.tid, tid)
    Define (tmp, id, label, child)
end while
tds := tmp
end Remove_Current_Context
interface

operation Initialize_Facility ()
  referenced state variables
  alters tdswp
  begin
  Initialize_TID_Info_Map (tid_info)
  Initialize_Current_Context (in_context, tid_info)
  end Initialize_Facility

operation Open_TDS (  
  consumes id: ID1
  )
  referenced state variables
  alters tdswp
  context
  variables
  tds:  Type_Declaration_Set
  id_copy: ID1
  label:  Label
  begin
  Undefine {tdswp, id, id_copy, label, tds}
  Add_PP_Context (in_context, label.pp)
  Add_Current_Context (in_context, tds)
  Define (tdswp, id_copy, label, tds)
  Open (tdswp, id)
  end Open_TDS

operation Close_TDS (  
  produces id: ID1
  )
  referenced state variables
  alters tdswp
  context
  variables
  tds:  Type_Declaration_Set
  id_copy: ID1
  label:  Label
  begin
  Close (tdswp, id)
  Undefine (tdswp, id, id_copy, label, tds)
  Remove_PP_Context (in_context, label.pp)
  Remove_Current_Context (in_context, tds)
  Define (tdswp, id_copy, label, tds)
  end Close_TDS
operation Reset_TDS ()
    referenced state variables
    alters tdswp
begin
    Reset (tdswp)
    Initialize_Current_Context (in_context, tid_info)
end Reset_TDS

operation New_TD ()
    consumes id: ID1
    consumes arity: Integer
    referenced state variables
    alters tdswp
context
    variables
        label: Label
        tds: Type_Declaration_Set
        arg, tid1, tid2, tid3: ID2
        pos: Integer
        info: TID_Info
        id_copy: ID1
begin
    label.tid := New_ID ()
    tid1 := Replica (label.tid)
    tid2 := Replica (label.tid)
    tid3 := Replica (label.tid)
    while pos < arity do
        arg := New_ID ()
        Add (label.pp, pos, arg)
        pos := pos + 1
    end while
    id_copy := Replica (id)
    Define (tdswp, id, label, tds)
    info.arity := arity
    Open (tdswp, id_copy)
    Get_Current_Position (tdswp, info.path)
    Close (tdswp, id_copy)
    Define (tid_info, tid1, info)
    Add (in_context, tid2)
    Define (dependencies, tid3, 0)
end New_TD
operation Delete_TD {
    preserves id: ID1
}

referenced state variables
    alters tds
context
    variables
        id_copy: ID1
        label: Label
        tds: Type_Declaration_Set
        info: TID_Info
        tid: ID2
        count: Integer

begin
    Undefine (tdswp, id, id_copy, label, tds)
    Undefine (tid_info, label.tid, tid, info)
    Remove (in_context, label.tid, tid)
    Undefine (dependencies, label.tid, tid, count)
end Delete_TD

operation Open_TD_Body {
    consumes id: ID1
}

referenced state variables
    alters tds
context
    variables
        id_copy: ID1
        label: Label
        tds: Type_Declaration_Set

begin
    Undefine (tdswp, id, id_copy, label, tds)
    label.body : = : tewp
    tewp_id : = : id_copy
    body_is_open : = : true
    Add_PP_Context (in_context, label.pp)
    Define (tdswp, id, label, tds)
end Open_TD_Body
operation Close_TD_Body {
  produces id: ID1
}

referenced state variables
alters tdswp
context
  variables
  id_copy: ID1
  label: Label
  tds: Type_Declaration_Set
begin
  Undefine (tdswp, tewp_id, id, label, tds)
  Reset (tewp)
  label.body := tewp
  body_is_open := false
  Remove_PP_Context (in_context, label.pp)
  Define (tdswp, tewp_id, label, tds)
end Close_TD_Body

operation Open_TE {
  consumes pos: Integer
}

referenced state variables
alters tdswp
begin
  Open (tewp, pos)
end Open_TE

operation Close_TE {
  produces pos: Integer
}

referenced state variables
alters tdswp
begin
  Close (tewp, pos)
end Close_TE

operation Reset_TE ()
referenced state variables
alters tdswp
begin
  Reset (tewp)
end Reset_TE
operation Insert_NIL_Leaf {
    preserves pos: Integer
}

referenced state variables
alters tdswp
context variables
te: Type_Expression
nil: ID2
begin
    New (te, nil)
    Add (tewp, pos, te)
end Insert_NIL_Leaf

operation Delete_TE {
    preserves pos: Integer
}

referenced state variables
alters tdswp
context variables
te: Type_Expression
tid, tid_copy1, tid_copy2: ID2
path: Path
info: TID_Info
count: Integer
id_copy: Label
tds: Type_Declaration_Set
begin
    Remove (tewp, pos, te)
    Undefine (tdswp, tewp_id, id_copy, label, tds)
    tid := Replica (label.tid)
    Define (tdswp, id_copy, label, tds)
    Get_Current_Position (tdswp, path)
    Undefine (tid_info, tid, tid_copy1, info)
    if not Is_Subpath (info.path, path) then
        Undefine (dependencies, tid, tid_copy2, count)
        count := count - Count_Occurences (te, tid)
        Define (dependencies, tid_copy2, count)
    end if
    Define (tid_info, tid_copy1, info)
end Delete_TE
operation Swap_Type_ID (
  alters  tid: ID2
)

referenced state variables
alters  tdswp
context
variables
  path:      Path
  tid, tid_copy1, tid_copy2: ID2
  info:      TID_Info
  count:     Integer
  te:        Type_Expression
  pos:       Integer

begin
  Get_Current_Position (tdswp, path)
  Undefine (tid_info, tid, tid_copy1, info)
  if not Is_Subpath (info.path, path) then
    Undefine (dependencies, tid, tid_copy2, count)
    count := count + 1
    Define (dependencies, tid_copy2, count)
  end if
  Define (tid_info, tid_copy1, info)
  Close (tdswp, pos)
  Remove (tdswp, pos, te)
  Swap_Label (te, tid)
  Add (tdswp, pos, te)
  Open (tdswp, pos)
  Undefine (tid_info, tid, tid_copy1, info)
  if not Is_Subpath (info.path, path) then
    Undefine (dependencies, tid, tid_copy2, count)
    count := count - 1
    Define (dependencies, tid_copy2, count)
  end if
  Define (tid_info, tid_copy1, info)
end Swap_Type_ID

operation Is_Defined (
  preserves  id: ID1
)  : Boolean
referenced state variables
preserves  tdswp
begin
  return Is_Defined (tdswp, id)
end Is_Defined

operation Current_Size ()  : Integer
referenced state variables
preserves  tdswp
begin
  return Size_Of_Current (tdswp)
end Current_Size
operation TDS_Path_Length () : Integer
    referenced state variables
    preserves tds wp
    begin
        return Position_Length (tdswp)
    end TDS_Path_Length

operation Dependency_Count ( ) : Integer
    preserves tid: ID2
    referenced state variables
    preserves tds wp
    context
    variables
        tid_copy: ID2
        count, result: Integer
    begin
        if Is Defined (dependencies, tid) then
            Undefine (dependencies, tid, tid_copy, count)
            result := count
            Define (dependencies, tid_copy, count)
        end if
        return result
    end Dependency_Count

operation TD_Type_ID ( ) : ID2
    preserves id: ID1
    referenced state variables
    preserves tds wp
    context
    variables
        id_copy: ID1
        label: Label
        tds: Type_Declaration_Set
tid_copy: ID2
    begin
        Undefine (tdswp, id, id_copy, label, tds)
tid_copy := Replica (label.tid)
Define (tdswp, id_copy, label, tds)
return tid_copy
end TD_Type_ID
operation TD_Argument (  
    preserves id: ID1  
    preserves pos: Integer  
) : ID2

referenced state variables
    preserves tdswp
context
    variables
        id_copy: ID1
        label: Label
        tds: Type_Declaration_Set

    tid, tid_copy: ID2

begin
    Undefine (tdswp, id, id_copy, label, tds)
    Remove (label.pp, pos, tid)
    tid_copy := Replica (tid)
    Add (label.pp, pos, tid)
    Define (tdswp, id_copy, label, tds)
    return tid_copy
end TD_Argument

operation TD_PP_Length (  
    preserves id: ID1  
) : Integer

referenced state variables
    preserves tdswp
context
    variables
        id_copy: ID1
        length: Integer
        label: Label
        tds: Type_Declaration_Set

begin
    Undefine (tdswp, id, id_copy, label, tds)
    length := Length (label.pp)
    Define (tdswp, id_copy, label, tds)
    return length
end TD_PP_Length

operation TE_Path_Length () : Integer

referenced state variables
    preserves tdswp
begin
    return Position_Length (tewp)
end TE_Path_Length
operation TE_Type_ID () : ID2
  referenced state variables
  preserves tdswp
  context
    variables
      tid, tid_copy: ID2
  begin
    Swap_Label (tewp, tid)
    tid_copy := Replica (tid)
    Swap_Label (tewp, tid)
    return tid_copy
  end TE_Type_ID

operation Current_Length () : Integer
  referenced state variables
  preserves tdswp
  begin
    return Number_Of_Children (tewp)
  end Current_Length

operation Is_DEFINED_In_Context (tid: ID2) : Boolean
  referenced state variables
  preserves tdswp
  begin
    return Is_Member (in_context, tid)
  end Is_DEFINED_In_Context

operation Body_Is_Open () : Boolean
  referenced state variables
  preserves tdswp
  begin
    return body_is_open
  end Body_Is_Open
operation Arity {
    preserves tid: ID2
} : Integer

referenced state variables
preserves tdswp

context
variables
    info: TID_Info
    tid_copy: ID2
    arity: Integer

begin
    Undefine (tid_info, tid, tid_copy, info)
    arity := info.arity
    Define (tid_info, tid, info)
    return arity
end Arity

end 1
BIBLIOGRAPHY


218


220


