A Framework for
Visual System Configuration

by

Michael Wallace Godfrey

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy.
Graduate Department of Computer Science
University of Toronto

© Copyright Michael Wallace Godfrey (1997)
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-27654-6
Abstract

A Framework for Visual System Configuration

Michael Wallace Godfrey
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
1997

System modelling is an important sub-area of software configuration management (SCM): it is concerned with how software systems are composed from their constituent elements. This dissertation investigates several problems associated with current approaches, and introduces the ConForm framework for visual system modelling.

A shortcoming of most SCM environments is that their system modelling notations incorporate only a small amount of the available information about a system and its components; in particular, architectural information, such as system structure and component interconnections, is rarely considered. There are several resulting problems:

- System models are often inscrutable and difficult to derive by visual inspection.
- Usually, there is no guarantee that the components listed in a system model can be combined in a meaningful way, or that all components are present.
- It is often difficult to reason about the effects of replacing components, and only limited automated support may be given for the construction of new systems.
- System modelling notations are often inherently textual, ad hoc, lack formal foundations, and are tied to a particular programming language.

The purpose of ConForm (Configuration Formalism) is to address these inadequacies within a single framework. In particular, ConForm:

- supports the modelling of software systems as a composition of versioned components;
- explicitly models architectural information, such as system structure and component interconnections;

- models relationships between versions of components based on architectural information;

- has a core formal model that is independent of any implementation language or platform, but may be extended to handle industrial languages such as C;

- allows user-defined design constraints to be modelled and enforced; and

- has both a textual and a visual representation.

ConForm's data model and calculus of system construction are formally defined using the Z specification language. A prototype version of ConForm has been constructed as a proof of concept, using the formal definition as a guide.
Acknowledgements

I owe my most sincere gratitude (and more) to many people. Since I cannot possibly express in a few words here how I feel, I won’t try. My thanks to:

- my supervisor, mentor, friend, and arbiter elegantius, Prof. Ric Holt;

- the rest of my PhD committee: Prof. Ric Hehner, Prof. Alberto Mendelzon, Prof. John Mylopoulos, and my external examiner, Dr. Susan Dart;

- my friends and advisors: Spiros Mancoridis, Dimitra Vista, Gary Farmaner, and Chrysanne DiMarco;

- the rest of holtgroup (past and present): Bil Tzerpos, Arthur Tateishi, Dave Penny, and Ron Wessels;

- and most of all, my family: my parents, my brothers Mark and Dan, my wife Anita, and Trevor the red-haired monster.

In the immortal words of Jerry Garcia, “What a long, strange trip it’s been.”

Now on with life!
## Contents

Abstract

1 Introduction

1.1 Configuration Management and System Modelling

1.2 Current Approaches to System Modelling

1.3 Related Areas

1.4 ConForm: A Framework for Visual System Configuration

1.5 Overview of the Dissertation

2 Previous Work

2.1 Module Interconnection Languages

2.1.1 MIL-75

2.1.2 Thomas’ MIL

2.1.3 Cooprider’s MIL

2.1.4 Intercol

2.1.5 NuMIL

2.1.6 Inscape

2.2 Software Configuration Management

2.2.1 RCS, Make and Related Tools

2.2.2 Commercial SCM Systems

2.2.3 Research SCM Systems

2.3 Program Understanding Systems

2.3.1 Rigi

Abstract

1 Introduction

1.1 Configuration Management and System Modelling

1.2 Current Approaches to System Modelling

1.3 Related Areas

1.4 ConForm: A Framework for Visual System Configuration

1.5 Overview of the Dissertation

2 Previous Work

2.1 Module Interconnection Languages

2.1.1 MIL-75

2.1.2 Thomas’ MIL

2.1.3 Cooprider’s MIL

2.1.4 Intercol

2.1.5 NuMIL

2.1.6 Inscape

2.2 Software Configuration Management

2.2.1 RCS, Make and Related Tools

2.2.2 Commercial SCM Systems

2.2.3 Research SCM Systems

2.3 Program Understanding Systems

2.3.1 Rigi
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.2</td>
<td>Scruple</td>
<td>24</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Refine/The Software Refinery</td>
<td>25</td>
</tr>
<tr>
<td>2.4</td>
<td>Summary</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>A Framework for Visual System Configuration</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>Motivation: Alice's Problem</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>Two Orthogonal Views of Software Systems</td>
<td>30</td>
</tr>
<tr>
<td>3.2.1</td>
<td>The Snapshot View</td>
<td>30</td>
</tr>
<tr>
<td>3.2.2</td>
<td>The Evolutionary View</td>
<td>32</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Current Approaches</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>ConForm and Abstract Program Components</td>
<td>33</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Resource Components</td>
<td>35</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Module Components</td>
<td>37</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Subsystem Components</td>
<td>41</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Summary: Abstract Program Components</td>
<td>44</td>
</tr>
<tr>
<td>3.4</td>
<td>Versions of Program Components</td>
<td>46</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Versions and Annotations</td>
<td>46</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Version Families and Repositories</td>
<td>47</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Versions of Composite Components</td>
<td>49</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Resource Version Families and Repositories</td>
<td>50</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Summary: Component Versions</td>
<td>52</td>
</tr>
<tr>
<td>3.5</td>
<td>Summary</td>
<td>53</td>
</tr>
<tr>
<td>4</td>
<td>Compatibility and Other Aspects of ConForm</td>
<td>54</td>
</tr>
<tr>
<td>4.1</td>
<td>Modelling Differences Between Versions</td>
<td>54</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Representing Differences Textually</td>
<td>55</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Inter-Version Relationships</td>
<td>59</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Relation to Previous Work</td>
<td>68</td>
</tr>
<tr>
<td>4.2</td>
<td>Additional Constraints</td>
<td>70</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Modelling Different Programming Languages</td>
<td>70</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Design Constraints</td>
<td>73</td>
</tr>
</tbody>
</table>
4.3 Summary ....................................................... 75

5 A Formal Model of ConForm 76

5.1 Introduction .................................................. 76
5.2 Motivation ..................................................... 77
5.3 Basics of Z ................................................... 78
5.4 Program Components ........................................ 80
  5.4.1 Basic Entities and Requirements ...................... 80
  5.4.2 Resources .................................................. 82
  5.4.3 Units: Modules and Subsystems ...................... 85
5.5 Example Programming Language Constraints .................. 89
  5.5.1 C Model ................................................... 89
  5.5.2 Object-Oriented Turing Model ......................... 97
5.6 Example Design Constraints .................................. 101
5.7 Versions of Program Components ............................... 104
  5.7.1 Component Versions ....................................... 104
  5.7.2 Inter-Version Relations ................................... 106
  5.7.3 Version Families ........................................ 113
  5.7.4 Repositories .............................................. 115
5.8 CalSyCon — A Calculus of System Construction ............... 116
  5.8.1 Modelling Operations in Z ................................. 116
  5.8.2 Creating and Adding Repositories and Version Families 118
  5.8.3 Checking in New Versions ................................ 120
  5.8.4 Constructing a New Subsystem ......................... 123
5.9 Summary ....................................................... 131

6 Implementation and Evaluation 132

6.1 Implementation Considerations .................................. 132
  6.1.1 A Typical Process ......................................... 132
  6.1.2 Specializing the ConForm Framework .................... 134
  6.1.3 Constituent Tools ......................................... 135
List of Figures

2.1 An example system model in Vesta. ........................................... 21

3.1 A snapshot view of MiniTunis. .................................................. 28
3.2 An evolutionary view of a MiniTunis component. ......................... 31
3.3 Source code of an example resource. .......................................... 35
3.4 Visual representation of an example resource. ............................. 36
3.5 Source code of an example module. ........................................... 38
3.6 Visual representation of an example module. ............................... 39
3.7 An example subsystem. ............................................................. 42
3.8 Visual representation of an example subsystem. ............................ 43
3.9 Visual representation of the Proto-ConForm system structure. .......... 45
3.10 Structure of the ConForm repository hierarchy. .......................... 48
3.11 Textual representation of an example subsystem version. ............... 50
3.12 Visual representation of an example subsystem version. ................ 51

4.1 Two similar versions of a source program. ............................... 56
4.2 Two ways of textually representing differences between versions. .... 58
4.3 Three versions of a stack pop procedure. ................................... 62
4.4 A simple C source program. ..................................................... 71
4.5 The ConForm visual representation of a C program. ....................... 74

6.1 The system architecture of a ConForm implementation. .................. 136
6.2 An example Proto-ConForm session. .......................................... 139
7.1 A snapshot view of MiniTunis version 95.0. ........................................ 144
7.2 Version 3.1 of the MiniTunis DEVICE_SYS subsystem. .......................... 146
7.3 The DEVICE_SYS subsystem of MiniTunis v95.0. ................................ 149
7.4 Source code for version 95.0 of the DeviceDriver module. .................. 151
7.5 ConForm description of a source module. ......................................... 152
7.6 ConForm description of a subsystem. .............................................. 152
7.7 The DeviceDriver module version family for MiniTunis. ....................... 154
7.8 CSTD representation of the differences between DeviceDriver versions 95.0
    and 3.1. .................................................................................. 155
7.9 Version 95.0 of the DEVICE_SYS subsystem. .................................... 157
7.10 CSTD representation of the differences between DEVICE_SYS versions 95.0 and 3.1. 160
7.11 Visual reconfiguration of MiniTunis. ............................................. 162
7.12 Version 95.1 of the DEVICE_SYS subsystem. .................................... 164
7.13 CSTD representation of the differences between DEVICE_SYS versions 95.1 and 95.0. 164

A.1 Grammar for the C-STD notation. .................................................. 181
Chapter 1

Introduction

In this chapter, we introduce the fundamental concepts of software configuration management (SCM). We begin with a discussion of basic terminology. Then, we consider the various sub-areas of SCM, including system modelling. Next, we briefly discuss the shortcomings of existing approaches to system modelling, and we consider other areas of software engineering research that have examined similar problems. Finally, we briefly outline a framework for visual software configuration called ConForm that addresses many of the shortcomings of existing system modelling notations.

1.1 Configuration Management and System Modelling

One of the key problems in industrial software production is the management of large systems comprised of many components that undergo constant evolution. Software configuration management is the area of software engineering that is concerned with these kinds of issues.

Large software systems are composed of and supported by many kinds of software artifacts, such as source code files, binary object files, requirements and design documents, formal specifications, user documentation, testing and verification data, and maintenance histories. Software artifacts may be divided into two classes: source artifacts, such as source code and raw documentation, which are created more or less directly by a human, and derived artifacts, such as binary object files or processed documentation, which are created by processing source artifacts (and possibly other derived artifacts) by automated tools.
Typically, the artifacts of large software systems change over time as bugs are fixed and the functionality evolves. Consequently, software artifacts often exist in multiple versions.

A software configuration is a collection of software artifacts that may be composed to form a coherent whole [15, 26, 93]. For example, source code files can be compiled into object code files, object code files can be linked together to form new object files, and units of user documentation can be interconnected to form a complete user manual. A system model describes what artifacts are part of a given configuration, and how the components are combined to form a whole.

A software objectbase is a repository of software artifacts. Two common approaches are the software archive and the workspace. Within a software archive, existing artifacts may not be changed or removed, but usually new artifacts may be added. Within a workspace, artifacts are usually mutable, and may be added or deleted. In practice, a combination of the two approaches is often used: archives store snapshots of all system artifacts at important checkpoints, and workspaces are used for day-to-day evolution of artifacts.

Software configuration management (SCM) is the activity of managing software artifacts and their configurations; it comprises several overlapping sub-areas, including component identification and classification, version control, system modelling, system building, process support/enforcement, co-ordination, auditing and control. Each of these areas has its own concerns:

Component identification and classification — How is a system broken down into components? Are there different kinds of components? What kinds of information about the artifacts are stored or understood by the SCM environment?

Version control — How are different versions of the same artifacts organized? What kinds of inter-version relationships are modelled? Can the version families be queried? Can versions be selected by attributes other than the version identifier?

System modelling — Is the system model simply a list of components? Is there more information? Can a system be composed of subsystems? Can the SCM environment detect if a system model is “well formed”? Does the system model “understand” the versioning system? How are composite artifacts modelled in terms of their (versioned) components? Are multiple views of a system supported?
CHAPTER 1. INTRODUCTION

System building — How do you construct the system from the system model? How do you select between versions of components? How do you ensure that the correct tools (e.g., compilers) are used in the construction? Are old object files stored by the environment?

Process support/enforcement — If a company uses a defined software development methodology, then its SCM tool may be expected to enforce various aspects of it, e.g., developers may not archive code until the design document has been approved.

Co-ordination — Are multiple users allowed to work on the same artifacts at the same time? If so, how are the results merged together afterwards? If not, how is this prohibition enforced? Can you determine which components each developer is currently working on?

Auditing and control — Is it possible to determine which developer has access to which artifact? Can you determine who made what change? Can you restrict access to artifacts?

HCI issues — Is the SCM environment easy to use? Are the system models easy to derive and understand? Does the environment support visual techniques? What kinds of queries are supported?

1.2 Current Approaches to System Modelling

Commercial SCM tools, such as those surveyed by Dart [15] and Feiler [26], have successfully addressed many of these areas, especially process support/enforcement, auditing and control, co-ordination, and some HCI issues. They have also been successful in addressing secondary issues such as efficiency, portability, management of derived artifacts, and distributed computing. Research-based SCM tools, such as Vesta [47], Jason [93, 94] and CAPITL [1], have concentrated mainly on system modelling and building, typically by creating a flexible repository for manipulating and querying databases of program components.

However, most SCM environments have not addressed system modelling beyond the basic requirements for automated system building. This is unfortunate, for there is much utility to be gained from a more comprehensive approach. Instead, we find several significant inadequacies with current approaches to system modelling:
• System models cannot be analyzed easily by visual inspection. Often, a system model consists of little more than a list of the components, and the only visible aspects of the system's design are the compilation dependencies.

• System models rarely indicate when a system description is "well formed".\(^1\) Automated support for replacing system components is uncommon.

• Little emphasis is placed on accessible, general-purpose, mathematically-precise abstractions for system modelling. Instead, most notations are \textit{ad hoc} and tied to particular implementation languages; also, they are usually textual, and lack a visual representation.

For example, one of the most common approaches to SCM within a UNIX environment is to use the tools RCS [88] and make [25].\(^2\) RCS efficiently stores multiple versions of components, and make is used to construct systems from its components. This approach is popular because the tools are simple to use, flexible, efficient, and are available without charge. However, a consequence of the flexibility of these tools is that they are not very sophisticated. For example, in RCS \textit{any} set of text files can be considered versions of each other, even if one file contains C source code and another contains hockey scores. Also, a Makefile has little idea of what is being constructed, and thus may appear arcane to anyone but its author.

1.3 Related Areas

One of the ways in which system modelling can be improved is to add architectural information, such as knowledge about program structure and interconnections between components, to system models and component descriptions. Previous work on adding architectural information to descriptions of software systems can be found in research on module interconnection languages (MILs), program understanding techniques, and software architecture. However, in each case the research in these areas has concentrated on solving other problems.

Research on module interconnection languages (MILs) initially concentrated on modelling static, structural properties of single systems, as exemplified by MIL-75 [19]. As ideas of

\(^{1}\)The idea of well-formedness is dependent on the programming language and construction tools used.

\(^{2}\)These tools are discussed in more depth in the next chapter.
interfaces and module interconnection became more mature, the emphasis of MILs shifted toward modelling syntactic properties of families of software components, as exemplified by INTERCOL [87]. Finally, incorporating descriptions of component semantics became the focus of MIL research, as exemplified by NuMIL [59] and Inscape [67].

Program understanding techniques are usually applied to single systems and are oriented toward programmer understanding of aspects of the software system, such as its structure, control and data flow. The usual aim in using a program understanding tool (e.g., Rigi [57], Refine [92], TXL [13], and SCRUPLE [63]) is to support sophisticated structural queries on source code that are used to detect errors or bad coding style. Sometimes, program understanding tools are used to perform reverse engineering; that is, the tool parses the code of the system to help to recover the original design, and to suggest possible improved high-level structuring.

The research area of software architecture still relatively new. While most of the current research is very high level, software architecture also concerns low-level interconnection ideas such as control and data flow. The basic motivation is to model commonalities (or architectures) of families of software systems in terms of high-level abstractions. The high-level abstractions often incorporate ideas such as interconnection bindings, control flow, and data flow. However, this knowledge can be constructed from the attributes of the constituent components: knowledge of the lower or concrete level architecture is often useful. As yet, work on software architecture description languages is still preliminary [81, 82, 56].

1.4 ConForm: A Framework for Visual System Configuration

ConForm (Configuration Formalism) is a framework for visual system configuration. Its main goal is to address the inadequacies of current SCM approaches to system modelling as noted above. In particular, ConForm:

- supports the modelling of software systems as a composition of persistent, versioned components;
- explicitly models architectural information, such as system structure and component interconnections, allowing for a rich system model;

5
CHAPTER 1. INTRODUCTION

- models relationships between versions of components based on architectural information;
- has a core formal model that is independent of any implementation language or platform, but may be extended to handle industrial languages such as C;
- allows user-defined design constraints to be modelled and enforced; and
- has both a textual and a visual representation.

ConForm's data model and calculus of system construction are formally defined using the Z specification language. This definition is presented in Chapter 5.

A prototype version of ConForm has been constructed as a proof of concept, using the formal definition as a guide. This prototype, which implements both the textual and visual aspects of ConForm, provides automated support for the browsing of version families of systems and components, as well as the creation of new systems from existing components.

1.5 Overview of the Dissertation

This chapter of the dissertation has introduced configuration management and system modelling, and has described how current techniques are inadequate in various important ways. The remainder of this dissertation is organized in the following way: Chapter 2 examines related previous work on these and similar problems by SCM systems, module interconnection languages, and program understanding systems. Chapter 3 introduces the ConForm visual notation for the modelling of versioned software systems. Chapter 4 discusses more advanced aspects of the ConForm framework, such as how compatibility of component versions can be modelled, and how the basic ConForm notation can be specialized with additional constraints. Chapter 5 presents the formal definition of the ConForm system modelling notation. Chapter 6 discusses implementation considerations and validation issues. Chapter 7 shows, by means of a typical software maintenance task, how a ConForm system can aid in the reconfiguration of a software system. Finally, Chapter 8 summarizes the research contributions of ConForm, and suggests future research directions.
Chapter 2

Previous Work

We now consider previous work that has addressed the issues raised in Chapter 1. The main body of research related to system modelling is found within literature on software configuration management (SCM). Additionally, approaches to modelling interconnections of software components can be found in research literature on module interconnection languages (MILs) and program understanding environments. We now examine each of these areas in more detail.

2.1 Module Interconnection Languages

Module interconnection languages, or MILs, arose in the mid-1970s. The original aim of MIL research was to gain an understanding into program structure at a more coarsely-grained level than that of the programming language statement. This was accomplished by focusing on how software systems were broken down into components and on the flow of information between the components.\(^1\) The gradual evolution of MILs research went from concentrating on static aspects of a single software system to modelling structural and semantic properties of families of systems comprised of multiple components. Prieto-Diaz [75] and Dean [16] have surveyed module interconnection languages.

Before considering individual MILs, some discussion of common terminology is required.

- A module is a contiguous segment of programming language code that forms a meaningful

\(^1\) At the time, software design ideas such as modularity and information hiding [61, 62] had only just been introduced, and most implementation languages did not provide sophisticated mechanisms to support them.
group; often, a module corresponds to a separate file in a file system.\(^2\)

- A **resource** is a named programming language entity — such as a procedure, variable, type, or constant — that modules may share with each other.

- A module **interface** lists the resources that a given module *provides* (or exports) for use by other modules. Depending on the MIL, a module interface may also list the module's **requirements**, which are the resources that the module requires from other modules.\(^3\)

- A **subsystem** is a collection of modules and possibly other subsystems. Often, a subsystem will contain other information as well, such as constraints on how its contained modules are allowed to share resources. A subsystem has an interface, which is derived from the interfaces of its components. Usually, a subsystem does not correspond directly to a programming language source entity.

In general, MILs model the interconnections between modules at a deeper level than system modelling notations found in configuration management systems. Also, MILs often have a strong idea of system coherence; since it is possible to determine what each module provides and requires, it is straightforward to determine when a system configuration is well formed. Later MILs supported the idea of families of evolving components, rather than modelling just a single system configuration. Relationships between versions of modules were usually based on syntactic identity and semantic compatibility. Early MILs provided limited, if any, automated tool support. Later MILs addressed the issues of tool support more seriously, and many ideas from MILs fertilized research in software development environments and configuration management systems.

We now discuss several MILs individually. We present the MILs in chronological order, so as to better illustrate the evolution of this research.

\(^2\)This definition should not be confused with the idea of a module as found in a modern programming language such as Modula, Ada, or Turing. In the context of a MIL, a module may be no more than a section of code; no enforced abstraction or information-hiding is assumed.

\(^3\)This somewhat limited idea of requirements should not be confused with the more general idea of software requirements and requirements analysis, which is a broad sub-area of software engineering [74, 82].
2.1.1 MIL-75

De Remer and Kron's landmark paper "Programming-in-the-Large Versus Programming-in-the-Small" [19] was notable for many reasons. The paper introduced the terms *programming-in-the-small* and *programming-in-the-large*, and suggested that while many good tools and notations existed for programming-in-the-small, there was a dearth of support for programming-in-the-large. They further suggested that the focus in programming-in-the-large should be on how modules interact, and that a special kind of language, which they called a module interconnection language, was needed. A MIL, they said, should concentrate on modelling coarsely-grained programming concepts, such as resources, modules, subsystems, and their interconnections. They provided an example of a MIL, called MIL-75.

MIL-75 considered that the modules and subsystems of a system form a tree, which they called a *System Tree*, with the modules/subsystems as nodes and edges indicating containment. When the structure of a system has been described by a System Tree, it is possible to construct a *Resource-Augmented System Tree*, which adds attributes to the module nodes that indicate which resources each module provides and requires. Additional constraints can then be added to the model to define desired visibility restrictions on modules and resources (*i.e.*, "who can see whom").

MIL-75 is important for being the first attempt at a MIL. Although considered elegant and successful, its simple approach had many inadequacies. No supporting tool was constructed to support MIL-75, although the authors hinted at a sort of compiler that would parse the system source code to validate it against the corresponding MIL-75 model. MIL-75 also considered program components simply to be named entities; no syntactic or semantic characteristics of resources or modules were modelled. Finally, MIL-75's basic paradigm of modelling only a single system after-the-fact was considered too inflexible. Any subsequent changes in system structure would entail significant changes to the MIL-75 system model, which in the absence of tools had to be performed by hand. Also, MIL-75 did not provide a means for modelling families of components that evolved over time. These issues were addressed in subsequent MILs.
2.1.2 Thomas’ MIL

Thomas’ MIL [75] was notable for two important contributions: first, he considered the practical aspects of incorporating interconnection information into a software development platform, including MIL tool design; second, he suggested a machine-processable notation for representing module interconnections. Thomas considered that the MIL-75 approach of modelling a single system after-the-fact to be too inflexible. His idea was to analyze module interconnections separately from and prior to that of (sub)systems (i.e., compiling, binding, and linking). Thus, it would then be easier to add new modules to a system with minimal recompilation.

The main weakness of Thomas’ approach is that it is heavily dependent on the ideas of compiling, binding, and linking. Later MILs considered more abstract ideas of subsystems and of families of components.

2.1.3 Cooprider’s MIL

Cooprider’s MIL [12, 75] was an important turning point in MIL research. Previously, the basic MIL paradigm was that of a post-mortem analysis of a single system, and the appropriate MIL tool was a sort of compiler. Cooprider shifted the emphasis to the construction of new systems from a repository of components. He was the first to consider modelling families of evolving components (i.e., versioning), rather than single systems. His MIL was the first to consider the version control and system building within an integrated software development environment.

Cooprider’s interconnection model was not significantly different from that of previous MILs. Also, his system construction notation did not exploit any understanding of the component versioning approach. Cooprider prototyped and tested his ideas; his work influenced Tichy’s Intercol (as well as the Gandalf software development environment of which Intercol was a component).

*Many of Thomas’ ideas were centred around making compilations of large systems efficient and minimizing unnecessary recompilation. Modern compilation tools now do a lot of this automatically.*
2.1.4 Intercol

Tichy's Intercol [87] was part of the Gandalf research project into software development environments at Carnegie-Mellon University. Intercol was the first MIL to consider the possible relationships between versions of the same component. In Intercol, all members of a (module or subsystem) version family shared the same abstract interface; that is, they should provide and require the same resources. Strict syntactic compatibility was not enforced at this level, however, as Intercol envisioned that a typical module version family would contain several components with identical semantics, but possibly written in different programming languages. Unlike its predecessors, Intercol had an interesting idea of subsystem version families, which allowed different versions to have different structures (i.e., different constituents) as long as the resulting subsystem interfaces were identical.

Intercol considered version control more seriously than Cooprider, and provided a mechanism for "checking in" new components to the repository. Like Thomas, Tichy spent considerable effort examining efficiency and compilation issues, most of which are now handled automatically by compilers. Unlike previous MILs, Intercol's build notation understood the versioning approach; to specify a concrete build plan, one indicated both the appropriate version family plus version identifiers for each component.

2.1.5 NuMIL

Narayanaswamy and Scacchi's NuMIL [59] generalized and expanded Tichy's work on Intercol several years later. In particular, NuMIL introduced interesting ideas of version families, inter-version relationships, and well-formedness of configurations. In most previous MILs, version family membership had been based on equivalence of interfaces: the provided resources had to have identical names and signatures, and the required resources had to be the same. NuMIL considered that these conditions were too strong: small differences in syntax should not rule out potential family members, and the requirements of a program component are really an implementation detail that do not belong as part of the abstract interface. NuMIL also considered that agreement on abstract syntax was not enough, that some modelling of semantics should

5Tichy was also responsible for RCS, which he developed subsequently.
be required as well. For example, the procedures MatrixTranspose and MatrixInvert might have the same syntax, but their semantics are very different, and consequently they should belong to different version families.

NuMIL considered that the unifying idea within a version family should be that all members satisfy the same abstract interface specification, or AIS. The AIS for a version family is written in a special-purpose notation that specifies the abstract syntax and semantics for each of the provided resources common to all member versions; required resources were not modelled in the AIS. The use of a special-purpose language allows evolution of signatures over time, as well as the use of multiple programming languages. Each member version also contains a concrete interface (a description of the syntax and semantics of the common resources, written in the programming language of the version), plus details of how the concrete interface can be mapped to the abstract one.6

It should be noted that while each member of a version family must provide all resources listed in the AIS, a member may also provide additional resources. A consequence of this is that version family members are not freely substitutable for each other. NuMIL thus introduced the idea of upward compatibility of module versions: module version $M_1$ is upwardly compatibility with $M_2$ iff

1. $M_1$ provides (at least) all of the resources provided by $M_2$;

2. $M_1$ requires no resources that are not required by $M_2$;

3. the syntactic properties of the resources common to $M_1$ and $M_2$ (both required and provided) are identical; and

4. for each operation $P$ in both $M_1$ and $M_2$, the following relationship holds:

$$\text{pre} (P, M_2) \Rightarrow \text{pre} (P, M_1)$$

$$\text{post} (P, M_1) \Rightarrow \text{post} (P, M_2)$$

where $\text{pre} (P, M)$ and $\text{post} (P, M)$ denote respectively the pre- and post-conditions that describe the concrete semantics of operation $P$ of module $M$.

6The Larch specification language [34] uses a similar mechanism to separate out implementation language details from the basic semantics of a formal specification. The "guts" of a Larch specification is written in the Larch Shared Language (LSL), an algebraic specification language. The LSL specification is then related to the actual implementation via an intermediate language that is specific to the implementation language of the program, such as the Larch/C interface language.
They also proved that if $M_1$ is upwardly compatible with $M_2$, then $M_1$ satisfies the AIS of $M_2$'s version family, and thus may be added as a member. This is useful because proving upward compatibility is usually easier than proving family membership.

NuMIL also provided a definition of well-formedness for configurations: A configuration $C = C_1, \ldots, C_N$, where each $C_i$ may be either a module or another configuration, is said to be well formed iff:

1. every resource provided by $C$ is provided by some $C_i$;
2. the resources required by $C$ are the set of unresolved requirements of the $C_i$s;
3. $C$ does not require and provide the same resource;
4. no $C_i$ provides and requires the same resource;
5. no resource is provided by more than one component;
6. uses of resources across module boundaries are syntactically consistent with their definitions; and
7. each $C_i$ satisfies its respective module family template (AIS).

These definitions give rise to a useful replacement rule:

If $C$ is a well-formed configuration and $M_1$ is a component of $C$ and if $M_2$ is upwardly compatible with $M_1$, then replacing $M_1$ by $M_2$ preserves well-formedness in the resulting configuration $C'$, as long as $M_2$ does not provide any additional resource whose name conflicts with a resource provided by another component of $C$.

The major weakness of NuMIL is that it concentrates on the semantics of program components. The addition of semantic information to interfaces meant that verifying upward compatibility or version family membership is undecidable; consequently, only limited automated support was possible. Some aspects of a NuMIL system were prototyped.

\footnote{Their definition extends an earlier idea suggested by Habermann and Perry [35].}
2.1.6 Inscape

Inscape [66, 67, 68] is notable for extending the level of semantic description of resources and modules: In Inscape, a module interface consists of the signatures of the provided operations (resources), pre- and post-conditions that describe each operation’s immediate semantic effects, and obligations, which are conditions that must be met eventually. Obligations model implied or assumed effects of an operation on the program state that might not be obvious: for example, a call to a procedure that opens a file might require a later call to a procedure that closes the file.

Inscape does not have an explicit idea of version families, but it does suggest several sophisticated kinds of inter-version relationships, which extend the ideas of compatibility and interchangeability of components previously presented by others [59, 35]. Most of these relations are based on the semantic properties of the operations (including obligations) and assume syntactic equivalence of signatures.

The major weakness of Inscape is its reliance on semantic ideas, which means that verifying that a relationship exists between two components is undecidable in general. No implementation of Inscape was built, although some of its ideas were prototyped and tested.

2.2 Software Configuration Management

As mentioned above, the main body of work related to system modelling is found within literature on software configuration management (SCM). Relevant approaches to SCM can be divided into three main sub-areas:

- RCS plus Make, and related tools,
- commercial SCM systems, and
- research SCM systems.

We now briefly examine several examples from each category.
CHAPTER 2. PREVIOUS WORK

2.2.1 RCS, Make and Related Tools

One method of configuration management that is popular within Unix\textsuperscript{8} environments is to combine the tools RCS, which manages the version families of the source artifacts, and Make, which is used to aid in system building. All of the tools listed in this section are available at no cost via ftp.

RCS — RCS [88, 23], or \textit{Revision Control System}, manages families of versions of source text files.\textsuperscript{9} Each family member (except for the first one) is considered to be a revision of a previous version — that is, the is-a-revision-of relationship forms a tree within the version family with the first version as root. RCS families are archives: once added, versions may not be altered or deleted. RCS implements access control via a strict check-in/out protocol: checking out a version means that the user gets a copy of an existing version; checking in a version means the user must supply a version identifier not already in use before being allowed to add a new version to the repository. RCS also performs simple concurrency control — users may optionally "lock" artifacts that they check out so that no one else may work on extensions to that artifact version.

RCS has no inner understanding of the member versions — any two text files may be considered versions of each other. Other than a version identifier (which may also indicate its ancestors in the revision tree), no additional annotations are stored with the versions.\textsuperscript{10}

It should be noted that RCS performs only version management; system modelling and system building are usually done with other tools, such as Make [25]. Several other SCM tools use RCS as a back-end to perform version management.

SCCS — SCCS [76, 23], or \textit{Source Code Control System} is very similar to RCS, although its user interface and functionality differ somewhat. SCCS is an older tool than RCS, and the design of RCS took into account some of the perceived deficiencies of SCCS.\textsuperscript{11} Like RCS, SCCS is used as a back-end for version management by several other SCM tools.

\textsuperscript{8}Unix is a registered trademark which currently belongs to the X/Open Company Ltd.

\textsuperscript{9}Recent versions of RCS also permit the managing of binary object files.

\textsuperscript{10}Recent versions of RCS allow some simple attributes, such as author and date, to be stored with versions.

\textsuperscript{11}Although SCCS still has a large user base, RCS has become much more popular.
The USENET comp.software.config-mgt Tool FAQ says of SCCS: "Although disputed, the general consensus has been that this tool is clumsy and not suited to large numbers of users working on the one project."[23]

CVS — CVS [7, 33, 23], or Concurrent Version System, extends RCS with features that make it more convenient for use in large projects with multiple developers. For example, CVS allows a set of files to be treated as a single unit, which can be checked in or out, whereas RCS requires that each file be checked in/out individually. Also, CVS allows multiple users to check out the same file at the same time when desired; this is not possible in RCS or SCCS. The results can then be merged and checked back in as a single entity. In industry, this is a common scenario; many commercial SCM systems provide sophisticated support for multi-version merging.

CVS requires RCS, which it uses as a back-end. Like RCS, CVS performs version management only; other tools are required to perform system modelling and building.

Make — Make [25] is a simple tool for system modelling and building. A system description is stored in a file called a Makefile. A typical entry in a Makefile has the following format:

\[
\langle \text{target} \rangle : \langle \text{td}_1 \rangle \cdots \langle \text{td}_k \rangle
\]

\[
\langle \text{build action} \rangle
\]

\( \langle \text{target} \rangle \) is the name of the goal target, \( \langle \text{td}_1 \rangle \cdots \langle \text{td}_k \rangle \) are targets on which the goal depends, and the build action is a command that is to be executed in the case that one or more of the target dependencies is out-of-date or missing.

For example, let us consider the following Makefile entry:

```
main: driver.o stack.o
    cc -o main stack.o driver.o
```

The command on the second line is executed if any of the following conditions is true:

1. If there is no file named `main` in the same directory as the Makefile.
2. If there is a file named `main`, but either `driver.o` or `stack.o` are missing.
3. If main, driver.o, and stack.o are all present, but the time of last modification of main is before that of one of the other two.

Although Make is often used in conjunction with RCS or SCCS, it has no inherent understanding of versioning. Unless all versions of a system have the same structure and compilation dependencies, versions of Makefiles must also be maintained.

Make also has no understanding of system model coherence; it assumes that the user has provided a meaningful and correct system model.\(^\text{12}\)

Nevertheless, because of its simplicity and flexibility, Make has become a very popular approach to system modelling and building. Many SCM tools offer Make-compatibility as a selling point. Other SCM tools, often referred to as “super-makes”, take Make as a starting point to offer additional functionality and flexibility.

**Odin** — Odin [11, 23] is system modelling and building tool. It consists of an SCM framework into which different external tools can be integrated, plus a mechanism for describing system models and constructing systems similar to that of Make. Odin does not perform version management directly; instead, RCS is frequently linked into Odin for this purpose.

Odin is intended to be a replacement for Make; it adds features not found in Make, while at the same time providing a simpler interface. Unlike Make, Odin has a limited understanding of how RCS manages versioning. Also, Odin archives file dependency information in a database. Like Make, Odin has no understanding of system model coherence.

**Shape/AFS** — The Shape toolkit [49, 46, 23] includes a version management facility, a “super-Make” build engine, plus other tools such as an integrated EMACS-compatible editor. Version management is performed using AFS (*Attributed File System*), a repository mechanism similar to RCS. Unlike RCS, arbitrary attributes may be attached to versions in AFS. System descriptions are stored in files called Shapefiles. Shapefiles are similar to Makefiles, except that version attributes may be referenced and used in selecting appropriate versions, and build constraints based on these attributes may be defined. For example, a Shapefile might give the construction recipe for a system and constrain the

\[^{12}\text{Some recent UNIX tools, such as imake, xmkmf and makedepend, allow for the automatic creation of Makefiles skeletons, given enough background information on the development platform and the system being constructed.}\]
version selection to search for the most recent component versions that support debugging and that run under X11 on a Sparc architecture.

AFS has no understanding of component versions beyond the user-supplied attributes. Shape has no innate idea of system coherence.

2.2.2 Commercial SCM Systems

**Aide De Camp** — Aide de Camp [23, 90], or ADC, differs from most other commercial SCM systems in two important ways. First, ADC is based on the *change-set* model. In this approach, entire systems are versioned instead of just modules. A *baseline* defines a snapshot of the entire system at a given moment. A developer may then work on his/her own version of the system, which is added to the repository as a change-set relative to the baseline. Since multiple developers may be working on a given system, facilities for multi-way merging of source code is an important feature of ADC.

Since ADC is based on the change-set model, it does not provide sophisticated facilities for reasoning about differences between versions of individual components. However, ADC does allow user-defined relations between change-sets.

The second way in which ADC differs from many of its competitors is that it supports limited understanding of the interconnections between components. This is accomplished by means of special language-specific scanners that can determine naive compilation dependencies. Also, some user-defined inter-component relationships can be modelled, and handled intelligently during system builds.

**ClearCase** — ClearCase [2, 23, 90] performs SCM by a kind of "big brother" approach [60]: ClearCase manages its own file system and tracks all changes automatically. ClearCase supports "transparent access" to versioned objects, which means that only one version of a component is visible to a developer at any given time. Version browsing and selection may be performed via a special graphical user interface; only the version history relation is modelled, however. ClearCase's system modelling notation is compatible with Make.

ClearCase provides limited "smart building". Since it monitors access to the files, it is able to keep track of what files are opened during compiles, and can thus generate an
CHAPTER 2. PREVIOUS WORK

accurate model of the compilation dependencies. The drawback to this approach is that it suffers from the "off-by-one" problem: if compilation dependencies change between invocations of the compiler, ClearCase will guess incorrectly and this may result in an incorrect compile. ClearCase's has no knowledge of component interconnections or system coherence beyond the information it gleans from "watching" compiles. ClearCase does not allow users to add user-defined links between components of a given system.

CMVC — IBM's CMVC [54, 23], or Configuration Management and Version Control uses an approach similar to the change-set approach of ADC; however, a change-set in CMVC must correspond to either a new feature or a "defect" (i.e., bug) fix. CMVC's major strength is in the ability to conveniently track the history of new features and bug fixes. CMVC has no inherent understanding of component versions, nor any innate idea of system coherence.

Continuus/CM — Continuus/CM [23, 90] is an SCM system that gives good support for development across distributed systems. Its system modelling language, ObjectMake, is very similar to Make. Within a system, Continuus/CM can perform limited compilation dependency analysis of components for programs written in C by means of a tool similar to the Unix program makedepend; apart from this information, Continuus/CM has no understanding of component versions and their interconnections, nor any idea of system coherence. Within a system, user-defined relationships between components may be added by the user. Limited support is given for taking advantage of this information during system builds.

2.2.3 Research SCM Systems

Adele — Adele [24, 23, 14] is a research SCM system from the University of Grenoble that has also become a commercial product. Essentially, Adele consists of an OODBMS with an event manager. Adele allows users to define attributes for component versions which are then stored with the versions in the database. Rules for constructing systems can then be defined in terms of these attributes using a limited logic. With user input, Adele can help in detecting incomplete and inconsistent system configurations. However, Adele has
no innate ideas of system coherence nor any understanding of component versions beyond the user-supplied attributes.

**Configuration Management Assistant** — The Configuration Management Assistant (CMA) [72, 14] from Tartan Laboratories provides a framework for creating SCM systems. As such, it is a "meta-SCM" system rather than an SCM system *per se*, as desired SCM "policies" must be defined by the user before it can be used. That is, CMA has no innate ideas of system coherence nor indeed any understanding of the entities it manages. Instead, a CMA implementation must be configured with particular ideas of what attributes system components have, and what kinds of relationships they may engage in.\(^{13}\)

In practice, CMA has been used successfully to manage software systems written in Ada.

**Vesta** — Vesta [47] is a research system developed at DEC's System Research Center. Vesta is based on earlier work done on the Cedar software development environment [86] and the Mesa programming language and environment [53, 85, 77].

Vesta's most notable feature is its build language, which is a powerful applicative\(^{14}\) programming language.

Within Vesta, the components of a system are individually versioned. Additionally, components may be grouped into "packages" (i.e., subsystems), which may also be versioned. A Vesta package description (i.e., system model) consists of three parts: the DIRECTORY clause, which gives convenient symbolic names to the actual source objects that comprise the package; the IMPORTS clause, which lists other packages whose contents are referenced, including tools; and the IN clause, which gives rules for constructing derived objects based on the source objects and tools referenced in the first two sections.

Figure 2.1 shows an example system model of a Vesta package. Each line in the DIRECTORY clause binds a convenient name, such as `Main.mod`, to a source object stored within the repository; this new name is used to refer to the source object within the system model. The IMPORT clause specifies which versions of which other packages are used within the system model; it also allows the binding of a short intuitive name (e.g., `building-env.v`)
CHAPTER 2. PREVIOUS WORK

Figure 2.1: An example system model in Vesta. The DIRECTORY clause lists the components of the package, the IMPORT clause lists the other packages that are used within the current package, and the IN clause lists sets of rules for constructing various derived objects.

to an object stored within the repository. We note that in this case, the imported package consists of a set of tools; in Vesta, tools are versioned and stored in packages within the repository. The IN clause specifies build rules using the tools and source objects referenced in the first two clauses. The rule presented here defines a function named build; it is a function of two parameters, a linker named M2$Prog and a compiler named M2$Compile. This function is evaluated by first compiling each module (i.e., M2$Compile(Main.mod) and M2$Compile(Subsystem.mod)) and then linking the results together.

While Vesta’s package mechanism supports a limited form of encapsulation and model inter-package interconnection information, its approach is limited to differentiating between versions based only on version identifiers — no other attributes are stored with component or package versions. Vesta’s idea of system coherence allows for limited well-formedness checking based on component version identifiers.

CAPITL — CAPITL [1], or Computer-Aided Programming-in-the-Large, is a research system developed at the University of Wisconsin.Basically, CAPITL consists of an OODB plus a flexible query language, Congress, that is a superset of Prolog. The OODB manages the storing of component versions, but only limited attributes — such as version identifier, timestamps, and file permissions — are stored with the versions.
A CAPITL system model is called a template. It consists of a list of generic components plus a set of constraints. A build is performed by first selecting component versions based on the given constraints via database lookup and then performing the desired build functions. Build tools, such as compilers and linkers, may also be versioned and selected by constraints defined in the template.

CAPITL does not model component interconnections, nor does it have an idea of system coherence; system models are not automatically generated or even verifiable by the system.

Jason — The Jason [93, 94] system, developed at the University of Washington by Douglas Wiebe, is an SCM framework rather than an environment. The author's fundamental hypothesis is that standard SCM notions — such as the underlying data model, what constitutes appropriate consistency constraints for system models, and how build plans may be specified — vary widely; consequently, no single environment is sufficient to cope intelligently with all SCM needs. Jason is intended to support generic SCM; Jason provides an object-oriented modelling language (similar to Telos [58]) in which the user must encode the desired "parameters" of his/her systems: object schemas (class definitions), consistency constraints, and build plans (dependency relations). These parameters are used by Jason to create a customized system that conforms to the particular needs of the user.

Jason provides some support for the intelligent managing of versioned components. For each kind of component to be stored, there must be an object schema that lists all of the expected attributes. All versions of a component must have these same attributes (although not necessarily the same values). New schemas may be added, but existing schemas may not be changed. Thus, for example, all instances of a given subsystem must have the same structure. First-order logic constraints may be added to object descriptions to ensure user-defined ideas of consistency.

A Jason build plan consists of a generic system model, version selection rules, a list of build dependencies, and a set of build actions; however, all of this information must be provided by the user. Jason does not explicitly model component interconnection information, but such information may be added by hand to form part of the component's defining object.
A novel aspect of Jason not found in other SCM systems is its formal algebraic model [94].

2.3 Program Understanding Systems

Program understanding systems, such as Rigi, Refine, and Scruple, are oriented toward providing the programmer with an in-depth view of the structure of a (single) software system and the interactions between its various components. Typically, a user of a program understanding system phrases sophisticated structural queries on the source code so as to find suspected errors or bad coding style, as well as to perform reverse engineering on the source. Reverse engineering [9] of software systems typically consists of two phases: first, the complete system source is parsed to determine the dependencies between source units; then, these dependencies are analysed and new system abstractions are made. The results of reverse engineering can then be used to restructure the system according to desired design principles.\textsuperscript{15} Reverse engineering is most commonly performed on legacy systems: large systems written long ago that have undergone significant changes without major reorganization of the structure.

Program understanding systems provide facilities to extract and examine the kinds of information flow between program components, as well as aid in constructing different architectural views of systems. However, versions of components and systems are not usually considered by these systems, nor is system construction. We now briefly examine several program understanding systems.

2.3.1 Rigi

Rigi [57, 4] is a system that uses program understanding techniques to allow the user to perform reverse engineering on large legacy systems written in languages such as COBOL and C. The use of Rigi usually proceeds in four steps:

1. The complete system source is parsed to extract the resource flow information between program components. The result of this step is a resource flow graph, where the nodes

\textsuperscript{15}This activity is called re-engineering.
are program modules and the edges denote information flow between modules. Rigi has parsers for COBOL and C.

2. The modules of the system are grouped into subsystems using Rigi's graphical editor. The grouping is done semi-automatically; that is, the grouping is done by the user with advice from Rigi.

3. Once the modules have been grouped into subsystems, the exact interfaces of the subsystems are computed (based on the resource flow of the contained modules to modules contained in other subsystems). The subsystem interfaces are based on actual inter-module resource usage, not just on what module imports/includes what other module.

4. Rigi then uses its own metrics to determine how “good” this arrangement is. Basically, Rigi favours small interfaces between subsystems (“low coupling”) as well as strong interactions between the components of a subsystem (“high cohesion”).

Rigi is notable for providing a visual environment for doing much of the work. Also, the use of alternative structural decompositions of a single system is supported and strongly encouraged.

2.3.2 Scruple

Scruple parses a program (written in a programming language that it understands, such as C) and then allows the user to pose queries of the source structure, using regular expressions and limited unification. For example, a user could pose the following queries of a program written in C:

1. Find all function declarations that return a value of the type COMPLEX_NUM.

2. Find all struct definitions containing a field with a pointer to the same struct.

3. Find any code segment where two variables swap values.

Scruple can thus be used to search through source code to find interesting patterns, known bugs, and inefficiencies in a way that a program like the Unix utility grep is unable to do.
2.3.3 Refine/The Software Refinery

The Software Refinery \cite{92, 4} is a commercial program understanding system similar in scope to Scruple. It consists of three major components:

- a generic parser, called DIALECT, that parses source code, extracts the program structure and resource flow information, and stores it in an abstract syntax tree (AST);

- an object-oriented database system, called REFINE, that stores the AST and supports queries on it; and

- a user interface, called Intervista.

The parser is generic in the sense that it must be configured to support the particular implementation language and its interconnection model. This allows the Software Refinery to be used in a variety of environments, but may require configuring by the user.

Once a system has been parsed, the user can then encode various kinds of "defects" in the REFINE query language. What constitutes a defect will depend on the requirements of the user; "defects" can include known kinds of errors, bad coding style, violations of project design or coding conventions, as well as enforcing desired project metrics.

2.4 Summary

In this chapter, we have examined previous research in several areas of software engineering that explore the modelling of software systems. In the next chapter, we motivate the need for an approach to system modelling that uses program understanding-like techniques to model interconnection information between program components.
Chapter 3

A Framework for Visual System Configuration

As previously discussed, existing approaches to system modelling within software configuration management (SCM) are inadequate in several ways. Most SCM systems do not consider system modelling beyond its immediate implications on system building; consequently, their system modelling notations often contain no more information than a list of the components plus instructions for compilation. Little attention has been paid to representing architectural information within system models.

Incorporating architectural information into system models has many potential benefits: it provides fast feedback about compatibility when developers must rebuild systems from older components; it allows easier reconfiguration of systems by the end-user; it provides a basis to aid in the understanding of system evolution; and architectural relationships between versions of components and systems can be defined and queried.

This chapter further motivates the need for a framework for visual system configuration that models component interconnection information, and introduces the ConForm system modelling notation. The first section describes a scenario that represents a common industrial situation: it provides the motivation for the rest of the chapter. Then, we consider two different ways of viewing software systems: the snapshot view, which shows the structured decomposition of a single version of a system, and the evolutionary view, which shows how a component (or set of
components) have changed over time. We then introduce the basic concepts of the ConForm system modelling notation, and discuss how both the snapshot and evolutionary views are supported. Some simple examples of using ConForm are presented, and a visual representation of the ConForm notation is given (subsequent chapters describe the ConForm model in more detail).

3.1 Motivation: Alice's Problem

We now motivate the need for a framework for system configuration that models component interconnection information by means of a scenario.

Suppose Alice is a software developer who has just been hired by NanoSoft Ltd., which markets a small operating system called MiniTunis. MiniTunis (Fig. 3.1) is a long-lived program comprising many software components, and each component exists in many versions, due to numerous bug fixes, system enhancements, and parallel development paths. Unfortunately, the system structure, syntax, and semantics are not constant across versions.

Now suppose Alice’s boss assigns her the following task: The device drivers in the most recent release of MiniTunis have been found to be unreliable (the device drivers are the four source code modules that make up the DEVICE_SYS subsystem in Fig. 3.1). Alice is to extract the device drivers from the previous version of MiniTunis along the same development path, and insert them into the current system, make whatever changes are needed, and ensure that all the desired functionality is present in the new system.

Alice's assigned task is far from straightforward. Just “plugging in” an old version of the DEVICE_SYS subsystem is likely to result in a number of compatibility problems:

- The old components might lack features that are provided by the newer versions (and required by clients of the subsystem).

- The old components might have “legacy” features that conflict with the rest of the new system.

\[1\text{We note that the scenario presented here is fictitious. While MiniTunis is a real software system, it is a model operating system used mainly as a teaching aid. Furthermore, both Alice and NanoSoft are inventions of the author. We have chosen the name Alice in honour of Lewis Carroll's Alice in Wonderland; while the software industry may not be a wonderland per se, it is full of eccentric characters who scurry about muttering, "I'm late. I'm late."} \]
Figure 3.1: A snapshot view of MiniTunis. The MiniTunis operating system consists of several source modules (small blue boxes with no visible subcomponents) organized into subsystems (large grey boxes that contain other boxes). The DEVICE_SYS subsystem is highlighted in this particular view.
• The interfaces of the modules and provided operations in the old subsystem might be incompatible with the current ones.

• The structure of the old version of the subsystem might be different from the new version. For example, the DiskMutex module does not exist in some of the older versions of MiniTunis (its basic functionality had originally been implemented by the Disk module). If any client of DEVICE_SYS in the current version of MiniTunis uses the module DiskMutex, then any replacement version of DEVICE_SYS must also have a compatible version of this module.

• The semantics of the old components might be different. Apart from "benign" semantic changes such as bug fixes, the semantics of the newer components might have changed considerably.

• The old components might not be compatible with the current construction tools and libraries.

Since Alice is new to the company, she is unlikely to have an intimate knowledge of the system structure or evolutionary history of MiniTunis. Consequently, she will require help in learning about the structure of MiniTunis and the different kinds of interconnections between the various components. Also, she will need help understanding how the system and its components have evolved over time and in understanding the precise differences between versions of the same components.

Alice’s task can be made much easier with good abstractions for modelling versioned software systems, and with appropriate supporting tools. In particular, Alice would be greatly aided by an approach that features informative system models; that is, system models that are concise and easily comprehensible through inspection and querying. Alice would also be aided by an approach that supports the convenient modelling of differences between component versions, including various kinds of inter-version relationships based on compatibility, interconnections, structure, and other features. Finally, research into visual representations of programs suggests that graphical techniques for representing and manipulating programs may be useful [70, 71, 8, 37, 38, 78, 48, 91].
Unfortunately, no mechanized tool exists that can aid in all of these areas. Configuration management systems concentrate on low-level version management and system building; they provide little support for reasoning about the well-formedness of configurations or about the difference between component versions beyond the simple textual level. Software understanding tools provide detailed information about program structure and interconnection information, but do not provide support for reasoning about the differences between versions of software components. Therefore, we now consider what kinds of techniques and tools might be able to help in solving “Alice’s Problem”. The rest of this chapter examines these issues in more detail, and introduces the ConForm system modelling notation. In appendix 7, we present a short example of how a ConForm tool might be used by Alice in adapting MiniTunis.

3.2 Two Orthogonal Views of Software Systems

Before proceeding further with discussion of possible solutions to Alice’s problem, it is useful to consider the different ways, both conceptually and visually, in which software systems can be viewed. We consider that there are two basic points of view:

1. the instantaneous snapshot view of a single version of the system, its components, and their inter-dependencies, and

2. the evolutionary or historical view of the system, its components, and the various relationships between successive versions of components.

Each of these points of view lends itself well to visual representation; Figure 3.1 shows an example snapshot of MiniTunis, and Fig. 3.2 shows a simple evolutionary view of a component of MiniTunis. We now consider these two kinds of views.

3.2.1 The Snapshot View

The structured view of MiniTunis shown in Fig. 3.1 is an example of a system snapshot. This diagram shows one possible design view of how the source modules may be grouped into logical subsystems; it is the sort of picture that an experienced developer might draw informally when trying to explain the structure of MiniTunis to Alice.
Figure 3.2: An evolutionary view of a MiniTunis component. Software configuration management environments commonly model only the historical or evolved-from relationship between versions of a component.

This particular snapshot does not reveal much detail about the MiniTunis system and its components; the decomposition into subsystems is shown, but no other architectural information is indicated. While this kind of informal diagram may be appropriate for high-level discussions of MiniTunis, it is not adequate if more detailed knowledge about component interconnections is desired. Without such detailed knowledge, it is difficult to reason about the effects of replacing components, as Alice has been asked to do. In our discussion of ConForm we will investigate modelling system views that contain significant interconnection information.
3.2.2 The Evolutionary View

Figure 3.2 shows an evolutionary view of a component of MiniTunis (the FileTable module). Each box represents a distinct version of this component, and the box's numerical label denotes its version number. An arrow drawn from one version to another indicates that the second version evolved—from the first; typically, the historical or evolved—from relation is the only inter-version relationship modelled by software configuration management systems.

While this picture gives a broad view of how the component has evolved, relatively little information about the individual component versions is shown; in particular, there is no modelling of the differences and commonalities between versions. If a configuration management system has a detailed understanding of the structure of the components, then it is possible to model inter-version relations based on various kinds of compatibility and similarity. Such an approach would greatly aid Alice in determining how the old device drivers differ from the newer versions; in section 3.4, we will examine how ConForm models the differences between component versions.

3.2.3 Current Approaches

Program understanding techniques, such as SoFi [6], Rigi [57], and TXL [13], as well as program analysis tools such as Refine [92] have been used successfully to analyze some inter-component dependencies within a system snapshot; however, relatively little work has been done in considering relationships between component versions.

Some software configuration management tools do address the evolutionary view of software systems, but not at a very sophisticated level. Simple SCM tools such as RCS and its relatives [88, 76, 7], consider only the evolved—from relation between component versions. Other more advanced SCM environments, such as commercial tools like Aide De Camp [23, 90] and research systems like CAPITL [1], allow the modelling of user-defined relations between component versions; however, all such relations must be derived and maintained by the user.

We now introduce the ConForm notation for modelling families of evolving software systems. We examine the snapshot view of systems in the next section by considering the nature of

---

2 Informal diagrams also have other problems: without the aid of tools, they are hard to verify against the actual system, and it can be difficult to keep them current as the system evolves.
abstract program components, both atomic and composite. Then, we consider the evolutionary view in section 3.4 by examining versions of abstract program components.

3.3 ConForm and Abstract Program Components

ConForm (Configuration Formalism) is an architectural framework for visual system configuration. It includes a visual notation for modelling software systems, and supports both snapshot and evolutionary system views. In this section, we introduce the basic concepts that underlie the ConForm model of software system structure. We begin by defining a taxonomy of program components at three levels of detail: resource, module, and subsystem.

1. A resource is a simple programming language entity, such as a procedure, variable, constant, or user-defined type. We consider resources to be atomic.

2. A module is a composite programming language entity; it corresponds to a unit of source code, such as a programming language module, class, monitor, file, or library. A module usually contains several resource definitions, and often restricts the visibility of its contained resources outside the module's boundaries.

3. A subsystem consists of a set of modules (plus possibly other subsystems). A subsystem may restrict the visibility of its contained modules (and transitively the modules' resources) outside its boundaries. Usually, subsystems are not programming language entities; consequently, subsystem visibility restrictions often cannot be enforced by programming-language-level tools.

Additionally, we use the term component to refer to a resource, module, or subsystem, and we use the term unit to refer to a composite component (i.e., a module or subsystem).

Although resources, modules, and subsystems differ as to their granularity in representing program entities, all three kinds of program components have a similar basic structure. In particular, we consider that every component has:

---

3Libraries are often linked into systems as binaries during compilation. Conceptually, however, they may be thought of as source code.
1. an externally visible interface that describes the subcomponents *provided by* the component,

2. a *body* that implements the provided subcomponents (and possibly other subcomponents), and

3. a set of *requirements* that correspond to unresolved dependencies on other program components.

Later, we will introduce a visual representation that takes advantage of this commonality of structure across all components. We will represent the body of a component visually as a box, we will represent the interface visually as a tab drawn on the top of the box, and we will represent the requirements visually as a set of slots drawn on the bottom of the box. The precise notion of what constitutes interfaces, implementations, and requirements (or alternatively tabs, bodies, and slots) depends on the component kind as well as the details of the underlying programming language.

It should be noted that some of the ideas and terminology presented here are well established. The basic taxonomy of program components as resources, modules, and subsystems originated in research on MILs (module interconnection languages), as did the idea of modules providing and requiring resources [19, 59, 75, 161]. The precise definitions of these terms vary between MILs; for example, many MILs do not consider subsystems to be distinct from modules. However, the ConForm approach to modelling software systems is novel in several ways. It differs from previous approaches in that it is *generic* (i.e., programming language independent), yet capable of being constrained to model the assumptions of industrial implementation languages, such as C. ConForm’s use of visual representations of program components is innovative. The modelling of relations between component versions based on interconnection information is significantly more detailed than with previous MILs. And its formal definition, presented in Chapter 5, gives ConForm a solid semantic basis that is uncommon to MILs.

The example program components we present in this chapter are written in the Object-Oriented Turing (OOT) programming language [36]. We have chosen OOT because it is a clean, high-level language with strong support for information-hiding, and because the prototype implementation of ConForm, Proto-ConForm, provides automated support for modelling
Figure 3.3: Source code of an example resource. The resource is the procedure Open from the MiniTunis source module Disk.

systems written in OOT. However, this does not mean that ConForm is useful only for programs written in OOT. The basic ConForm data model, which is formally defined in Chapter 5, is not dependent on a particular programming language or environment. Additional constraints may be added to the basic model to incorporate the assumptions of particular programming languages. This has been done for the OOT and C languages; this work is described in section 4.2.1 and section 5.5.

We now describe ConForm's program components in more detail.

### 3.3.1 Resource Components

We consider a resource to be an atomic program component. A resource consists of a name and signature (its tab), plus implementing code (its body), and a set of requirements (its slots): the slots correspond to the names of entities used but not declared within the resource definition.

For example, consider the procedure Open from the MiniTunis source module Disk (Fig. 3.3).

The tab of this procedure resource consists of the name Open plus its signature:

```
procedure Open (disk : MinorDeviceID)
  DiskMutex.Start (disk)
  openCount (disk) += 1
  DiskMutex.Finish (disk)
end Open
```

The body consists of the code that implements the procedure. The slots correspond to the names used within the procedure definition that are neither parameters nor sub-resources of the procedure: DiskMutex, Start, Finish, openCount, and MinorDeviceID. In the common case, the requirements of a resource can be derived automatically from the code by means of program understanding tools, such as a special-purpose parser but often the only information

---

4 Although a resource definition may contain sub-resources, such as local variables declared within a procedure, we are interested only in those resources that exist at the "top level" of a module, and are thus capable of being exported by and shared between modules.

5 Such a parser must be specific to the particular implementation language.
Figure 3.4: Visual representation of an example resource. The resource is the Open procedure from the MiniTunis source module Disk presented in Fig. 3.3. The tab of a resource consists of the resource name (Open) plus its signature. The body consists of the implementing code. The slots correspond to the unresolved component names within the code: for convenience of representation, the five requirements of Open have been visually represented here as a single slot.

that can be derived about the required component is its name.

Visual Representation of Resources

Figure 3.4 shows the ConForm visual representation of the Open procedure. Resources, modules, and subsystems all have the same basic visual shape (i.e., tab, body, and slots): to allow a component’s kind to be immediately discernible (i.e., if it is a resource, module, or subsystem), we colour code the tabs and slots: resource tabs and slots are green, module tabs and slots are red, and subsystem tabs and slots are blue.6 A neutral colour is used for the component body, so that a component’s internal details can be shown clearly.

Resources are contained in modules. The visual representation of a module typically contains the visual representations of several resources within its body. An interconnection between

---

6The particular choice of colours is arbitrary: green, red, and blue are simply striking, distinct colours.
resources within a module is indicated by an arrow drawn from the tab of one resource to the
slot of another; this indicates that the first resource resolves a requirement of the second. We
now discuss modules in more detail.

3.3.2 Module Components

A module is a composite program component; modules contain resources. The body of a module
consists of resource definitions, plus some syntactic or structural wrapping. The tab of a module
consists of the module name plus a set of provided or exported resources (each of which is defined
within the module). Usually, we consider that only resource names and signatures are exported
by the module, and therefore that a module tab consists of a name plus a set of resource tabs.
The resource bodies are considered to be hidden by the defining module, and resource slots are
an implementation detail that is not relevant to any potential “client” of the module. Note that
a module may contain local resources that are not exported, and hence may not be accessed
outside of the module.

There are two kinds of module slots or requirements. First, there are resource-level require-
ments; these are the requirements of the contained resources that are not resolved by other
resources within the module. Usually, resource-level requirements are not listed explicitly in
the code, but they can be derived automatically by special-purpose tools. Second, there are
module-level requirements; these are explicitly stated dependencies on other modules that are
listed in import, include, inherit, and similar programming language statements. Typically,
these modules are required as they provide resources that satisfy the main module’s resource-
level requirements.

Figure 3.5 shows the source code of an example module (the Disk module of MiniTunis)
with most of the detail elided. This module defines thirteen resources, but we have shown only
three in this view. The export statement lists which of the contained resources are provided to
clients of the module. The import statement lists the module-level dependencies or slots. The
resource-level slots are not stated explicitly; if these are desired, they must be derived from the
code. The rest of the module body contains resource definitions and initialization.
Figure 3.5: Source code of an example module. This is the Disk module from MiniTunis. Most of the detail has been elided.

Visual Representation of Modules

Figure 3.6 shows the ConForm visual representation of the Disk module whose source was given in Fig. 3.5. This visual representation is not dependent on the particular programming language, and is more immediately informative than the textual representation (Fig. 3.5), which requires special knowledge of the programming language syntax to deduce what is provided/required, and what resources are defined. As in the textual representation, not all of the module information is shown here: all of the module's tabs and slots are shown, but some of the body detail has been elided.

This module's tab consists of the module name, Disk, plus the tabs of six of the resources defined within the module body (the resource signatures have been hidden in this view). The module's slots are shown at the bottom. There are four resource-level slots (the green slots near the bottom) which correspond to the unresolved slots of the resources defined within the module, and there are three module-level slots (the red slots at the bottom) which correspond to the stated module dependencies given in the import list in the module's source code.
CHAPTER 3. A FRAMEWORK FOR VISUAL SYSTEM CONFIGURATION

Figure 3.6: Visual representation of an example module. This is the Disk module of MiniTunis presented in Fig. 3.5. All of the module's tabs and slots are indicated, but only three of the thirteen resources contained in Disk are shown. The green slots at the bottom of the diagram denote the resource-level requirements of the module, and the red slots denote the module-level requirements. A green arrow (from a resource tab to another resource's slot) indicates a resource interconnection. A purple arrow denotes a propagation relation: a resource tab that is also a tab of the module, or a resource slot that is also a slot of the module. The grey slot in Open indicates that other slot information of this resource is elided from this view.
CHAPTER 3. A FRAMEWORK FOR VISUAL SYSTEM CONFIGURATION

In the view presented in Fig. 3.6, three of the module's thirteen resources are shown within the module's body: the procedure Open, and the variables openCount and lastDisk. The remaining resources have been elided from view to make it easier to see the various relations involving these three. All of the tabs, slots, and bodies of these resources are shown, with the exception of the grey slot of Open which indicates that some slot information of this resource has been elided.

There are two kinds of arrows that can be drawn within the body of a composite component (i.e., module or subsystem). The first kind is called a resolution arrow, and is drawn from the tab of one subcomponent to the slot of another. Such an arrow denotes an interconnection between the two subcomponents; that is, the first subcomponent resolves or satisfies a requirement of the second subcomponent. There are two kinds of these arrows: resource-level resolution arrows, which are green (i.e., "resource colour") and drawn from a resource tab to a resource slot, and module-level resolution arrows, which are red (i.e., "module colour") and drawn from a module tab and module slot. For example, in Fig. 3.6 the green arrow drawn from the tab of openCount to the slot of Open is a resource-level resolution arrow. The procedure Open requires a resource named openCount; variable openCount satisfies this requirement. Module-level interconnection arrows are discussed in the next section.

The second kind of arrow is called a propagation arrow, and it denotes a relation between a composite component and one of its subcomponents. Propagation arrows are drawn in purple, and come in two flavours. The first kind of propagation arrow is drawn from the tab of a subcomponent to the tab of the containing component; this indicates that the subcomponent is exported by the container, i.e., that it is propagated out of the container. For example, in Fig. 3.6 the arrow drawn from Open's tab to the module tab indicates that Open is exported by the module. The second kind of propagation arrow is drawn from a slot of the container to the slot of a subcomponent; this indicates that the subcomponent slot is not resolved by another subcomponent of the container and is thus a slot of the container. For example, the purple arrow drawn to the slot of Open labelled MinorDeviceID from the green slot of the same name at the bottom indicates that this resource slot is propagated out of the module.

\footnote{In particular, it should be noted that the variable lastDisk has no slots.}
3.3.3 Subsystem Components

Subsystems, like modules, are composite program components; they contain modules and other subsystems. We consider subsystems to be an abstraction mechanism. Each subsystem groups a set of the system's modules (and subsystems) into a logical whole; as such, each subsystem defines a *snapshot* view of part of the system. We also consider that a software system may be decomposed into subsystems in many ways. This approach is useful, as it allows systems to be considered from different points of view.

Like resources and modules, a subsystem has a tab, a body, and slots. The tab of a subsystem consists of a name plus a set of module tabs (each of which in turn consists of a module name and a set of resource tabs). The body of a subsystem consists of module and subsystem definitions, plus some syntactic or structural wrapping. The slots of a subsystem correspond to the unresolved slots, both module-level and resource-level, of its components.

As we consider subsystems to be little more than an abstraction mechanism for grouping units and since subsystems do not correspond to entities of the underlying programming language, we make the simplifying assumption that a subsystem may not have stated unit requirements of its own. That is, a subsystem may not "import" other modules or subsystems. This means that the requirements of a subsystem correspond to the requirements of the (contained) source code components only. This assumption simplifies the resulting system models, as it implies there are only by two levels of inter-component interconnections, or resolved requirements, and they involve only programming language-level components (*i.e.*, resources and modules).

Like a module, a subsystem may restrict access to its subcomponents by means of an explicit export list. The default ConForm rule for subsystem exports is that a subsystem may provide (or export) any of its contained modules, as well as any module exported by any of its contained subsystems. We have chosen a fairly liberal rule of what modules a subsystem may export as the default. More conservative conventions may be imposed by adding user-defined scoping rules, which we call *design constraints*. Design constraints are introduced in section 4.2.2, and discussed in detail in Chapter 5.

For example, consider the FILE_SYS subsystem from Fig. 3.1. FILE_SYS consists of the
Figure 3.7: An example subsystem. A subsystem is a container of modules (and other subsystems). It may define restrictions on the allowed accesses of its contained units. This subsystem contains four source code modules, and permits two of them (DeviceDriver and Tty) to be used outside the subsystem boundary. The subsystem description notation used here is a variant of the SIL language [51].

modules File, Directory, and FileTable, plus the subsystem INODE_SYS. FILE_SYS may export any of its three top-level modules, plus any of the modules exported by INODE_SYS. The default ConForm rule for subsystem exports asserts that FILE_SYS may not export any sub-module of INODE_SYS that INODE_SYS does not explicitly export.8

It should be noted that since ConForm subsystems do not correspond directly to entities of the underlying programming language, subsystem-level access restrictions usually cannot be enforced by programming language tools (such as compilers) and, if they are to be enforced, must be checked by other means. An implementation of ConForm would provide such tools to verify the well-formedness of software systems according to user-defined design constraints.

Figure 3.7 shows a typical subsystem, the DEVICE_SYS subsystem of MiniThis. DEVICE_SYS contains four source code modules (DeviceDriver, Tty, Disk and DiskMutex), and exports two of them (DeviceDriver and Tty). The subsystem description notation used here is a variant of the SIL language [51]; however, this textual representation is very coarsely grained and provides little information about the internal components and their interdependencies. We now consider a more finely-grained approach to representing subsystems.

Visual Representation of Subsystems

Figure 3.8 shows the ConForm visual representation of the subsystem from Fig. 3.7. The subsystem's tab (shown in blue) consists of a name plus a set of module tabs. Like modules.

---

8It is syntactically legal for a subsystem to export a (top-level) contained subsystem, but this is simply a short-form for exporting all of the contained subsystem's components.
Figure 3.8: Visual representation of an example subsystem. A red arrow drawn from a module’s tab to another module’s slot denotes a module resolution. As in Fig. 3.6, purple arrows denote propagation relations: module tabs/slots that are also tabs/slots of the subsystem. Here, all resource-level information (i.e., the green tabs and slots) has been elided.
subsystems have both resource-level and module-level slots. However, as discussed above, a subsystem may not "import" other units; thus, the slots of a subsystem correspond to the unresolved resource-level and module-level slots of its contained units.

The module-level interconnection information is highlighted in this diagram; all of the contained modules are shown here, as are all of the module interconnections. To make the module interconnection information stand out clearly, all of the resource information has been elided; the resource tabs are shown as unlabelled green stumps, and the resource slots are "greyed out". Of course, resource information may be added to this diagram if more detail is desired.

As in the visual representation of modules (e.g., Fig. 3.6), both resolution and propagation arrows may appear within subsystems. For example, the red arrow from the tab of the contained module Tty to the left-most slot of the contained module DeviceDriver is a module-level resolution arrow which indicates that Tty satisfies a (module-level) requirement of DeviceDriver. The purple arrow from the tab of Tty to the tab within the blue subsystem tab at the top is a propagation arrow that indicates that Tty is exported by the subsystem. The purple arrows from the red slot at the bottom of the subsystem labelled System to red slots of three of the four modules are propagation arrows that indicates that System is a module-level requirement of the subsystem that corresponds to a module-level requirement in the three contained modules.

Figure 3.9 shows the system structure of the ConForm prototype implementation, Proto-ConForm. Proto-ConForm consists of six major subsystems, each of which consists of several modules. In this view, the internal details of three of the subsystems are shown, while the rest are elided. It is worth noting that even in a fairly small system such as Proto-ConForm, the amount of visual information in a ConForm diagram can quickly overwhelm the reader. We consider that this clearly demonstrates the need for tool support for interactive visual navigation and querying.

3.3.4 Summary: Abstract Program Components

In Fig. 3.1, we presented a high-level snapshot of the MiniTunis system. We argued that although such a simple diagram might be useful for high-level discussions of MiniTunis, it does not provide much information about component interconnections and is not adequate for Alice's needs. Figure 3.8 provides a snapshot view that shows a lot more information about
Figure 3.9: Visual representation of the Proto-ConForm system structure. Much of the detail has been elided from view so as to focus on the structure and dependencies of three of the major subsystems.
the DEVICE_SYS subsystem and its module-level interconnections. If desired, resource-level interconnections can also be shown.

We have now presented the basic ConForm model of program components, and have seen how detailed system snapshot views (i.e., subsystem decompositions) can be built from programming language resources and modules. We next consider the evolutionary view of systems by examining how program components may exist in multiple versions, and considering the possible relations between versions of the same component.

3.4 Versions of Program Components

As discussed above, software systems and their components typically evolve over time as bugs are fixed, new features are added, and different implementation strategies are employed. Usually, it is not sufficient for a software company to keep only the source code of the most recent version of a system. Different clients may have highly customized versions of the system, and it is often necessary to retrace one’s steps when a development path leads to intractable problems. Consequently, we must consider representing and reasoning about different versions of program components; this is the evolutionary view of software systems mentioned in section 3.2.

We begin this section by considering what constitutes a version of a program component. We then discuss how versions of the same component are grouped into version families, and how version families are composed into system repositories. Finally, we discuss various relationships between versions based on structural, syntactic, and compatibility information.

3.4.1 Versions and Annotations

A component version consists of a component definition plus some annotations; annotations are extra pieces of information about particular versions that may not be contained within the component definition itself. The most important annotation is the version identifier, or vID. This is a tag that is used to distinguish each version from the other versions of the same component; consequently, within a collection of component versions, version identifiers must be unique. Most versioning systems use a kind of decimal number as the version identifier, which is called the version number. For example, Fig. 3.2 shows a common naming scheme for
version identifiers. We will follow this convention in our examples, and use numbers as version identifiers.

Apart from the version identifier, a component version may have additional annotations, such as a timestamp, a set of descriptive keywords, or a short prose description of the component. Such annotations may simply provide extra information about the component version to someone browsing the version space, or they may be used to define relations between versions. A component's keywords might list any unusual features or requirements such as operating system dependencies. For example, a graphical program that runs on multiple platforms might add a keyword to all of its platform-dependent subcomponents that indicates if the component is Macintosh, MS-Window, or X specific.

3.4.2 Version Families and Repositories

Versions of the same component are grouped into a version family, or VF. Since in our model there are three kinds of components—resources, modules, and subsystems—we consider that there are three kinds of version families: resource VFs, module VFs, and subsystem VFs.

A system repository consists of the version families of all of the modules and subsystems of a software system. For example, the MiniTunis system repository has a VF for each module and subsystem that is contained in at least one version of the system. Thus, there is a VF that contains all the versions of the Disk module; within the Disk VF, different versions are distinguished by their version identifiers (which must be unique within the VF).

A collection of system repositories is called a software repository.

Figure 3.10 illustrates the basic structure of the ConForm repository hierarchy. For simplicity, we have shown only one example of each kind of component or component container. The NanoSoft software repository collects the system repositories of all of the company's active projects, including that of the MiniTunis operating system. That is, the software repository consists of a set of system repositories.

\footnote{Often, all of the component versions within a version family will have the same name, e.g., all of the members of the module VF Disk will be named Disk. However, this is not a requirement. For simplicity, in this dissertation we will assume that all versions of a component have the same name, which can also then serve as the name of the version family.}

\footnote{VFs of resources are considered to be contained within the VF of the modules in which the resources are defined. This is discussed below.}
Figure 3.10: Structure of the ConForm repository hierarchy. For simplicity, we have shown only one example of each kind of component version (i.e., resource, module, and subsystem) and component container (i.e., VF and repository). A software repository groups a set of system repositories. A system repository consists of a set of module and subsystem version families (VFs), each of which contains a set of versions of the same module or subsystem component. A module VF also contains a resource repository, which consists of a set of resource VFs. Finally, each resource VF consists of the set of all of the versions of a particular resource.
CHAPTER 3. A FRAMEWORK FOR VISUAL SYSTEM CONFIGURATION

The MiniTunis system repository consists of a set of subsystem and module version families, one for each subsystem and module that exists in any version of the MiniTunis operating system. Here, we have shown one subsystem VF (DEVICE_SYS) and one module VF (Disk). A subsystem or module VF contains the set of all the versions of that unit; in the diagram, we have shown one version of each unit.

Each module VF also contains a resource repository, which consists of a set of resource VFs, one for each resource contained in any version of the module. Each resource VF contains the set of all the different versions of that resource; in the diagram, we have shown one version of the Open procedure. In practice, the number of resource VFs within a resource repository can become large quickly, and it is not clear how often a typical user will wish to "browse the version space" at the resource level. However, we have chosen to model resource versions explicitly for two main reasons: first, resource interconnection information is an important part of system models; and second, we wish to be consistent with how module and subsystem versions are modelled. Additionally, we note that the creation and management of resource VFs requires little, if any, manual effort.

The ConForm model assumes a flattened view of unit VFs within the system repository; that is, we have not structured the system repository into a snapshot-like subsystem decomposition. In particular, in Fig. 3.10 we have not "nested" the Disk module VF within the DEVICE_SYS subsystem VF. The reason for this is that we wish to be able to represent all versions of the MiniTunis system, even if the subsystem structuring varies significantly. Although grouping all of the unit VFs at the top-level of the system repository is a "lowest-common denominator" approach, different views of the system hierarchy can also be stored in the system repository to aid browsing.

3.4.3 Versions of Composite Components

In addition to a component definition and annotations, a version of a composite component (i.e., module or subsystem) has a component list; this list gives the version family and version number of each of the subcomponents. For example, Fig. 3.11 shows a textual representation of version 2.0 of the DEVICE_SYS subsystem. The component list is given by the contain statement, which lists the version numbers of each of the subcomponents: version 2.0 of DeviceDriver
Figure 3.11: Textual representation of an example subsystem version. The version numbers of the subsystem and its subcomponents are indicated beside the component name.

and Tty, version 1.1 of Disk, and version 1.2 of DiskMutex.

As in Fig. 3.7, the textual representation of the subsystem version shows little information about the components and their interdependencies. The ConForm visual representation of this subsystem version (Fig. 3.12) shows all of the information from Fig. 3.8; additionally, the version number of each component has been indicated by a visual “tag”.

3.4.4 Resource Version Families and Repositories

As mentioned above, a system repository consists of the version families of the modules and subsystems of a given software system. Version families of resources, however, are not grouped at the top level with the module and subsystem VFs. Instead, resource VFs are grouped locally within the VF of the module in which the resources are defined. That is, we consider that each module VF contains a resource repository that contains the VFs of all of the resources defined within any version of that module. For example, the resource repository of the Disk module VF contains at least thirteen resource VFs, one for each resource contained in the version of Disk: a VF for all versions of the Open procedure, a VF for all versions of the Read procedure, etc. The resource repository may contain even more resource VFs if other versions of Disk contain additional resources.

Resource VFs are grouped locally by module VF instead of globally at the top-level of the system repository for two reasons: first, the same resource name may be used to refer to different resources in different modules, and second, the number of resources in even a small
Figure 3.12: Visual representation of an example subsystem version. This is the same subsystem version as in Fig. 3.11, and it is also the versioned view of the subsystem given in Fig. 3.8. The purple “tags” give the version numbers of each component.
system can be extremely large, making browsing of the resource versions inconvenient if there is a flat namespace of resources.

3.4.5 Summary: Component Versions

Since program components exist in multiple versions, we have had to consider both the nature of versioned program components as well as how such components can be organized coherently into a repository. Program component versions extend the basic component definition with annotations, such as a version identifier and a list of descriptive keywords. Additionally, composite components require a component list to indicate which version of each subcomponent they require. In this section, we also described the ConForm approach to structuring repositories of program components: the versions of each component are stored in version families (VFs); resource VFs are grouped within the VF of their defining module; module and subsystem VFs are grouped into system repositories; and a group of system repositories comprises a software repository.

Previous work in MILs and software configuration management systems have used approaches that are similar to ConForm in some ways. The basic ideas of repositories and version families are well known. Intercol [87] and NuMil [59] were among the first to address criteria for organizing modules into version families, although their ideas were fairly restrictive, and their support for representing subsystems and resources were limited. Recent research systems CAPITL [1] and Jason [93] have more mature and highly configurable ideas of version families. but significant effort is required to set up and maintain the repository structure. ConForm differs from previous approaches in that it gives significant support to modelling subsystem and resource version families (in addition to module version families). Since ConForm considers that resources, modules, and subsystem all share a similar basic structure (i.e., tabs, bodies, and slots), version families for each kind of component can be structured and manipulated in a similar manner.
3.5 Summary

This chapter has motivated the need for an approach to system modelling that supports both the snapshot and the evolutionary view of systems. We introduced ConForm which is a generic framework for versioned system modelling. We showed how ConForm can model program components at three levels of abstraction (resources, modules, and subsystems), and we presented a visual interpretation of both the snapshot and the evolutionary views of program components. In the next chapter, we consider other aspects of ConForm, such as modelling various kinds of inter-version relations, and the addition of constraints to model specific implementation languages and system structuring conventions.
Chapter 4

Compatibility and Other Aspects of ConForm

In the previous chapter, we motivated the need for an approach to system modelling that addressed both the snapshot and evolutionary views of software systems, and we introduced the ConForm notation for versioned system modelling. In this chapter, we examine other aspects of ConForm. First, having considered the nature of component versions and how versions may be organized into repositories, we discuss how differences and similarities between versions of the same component may be modelled. Then, we examine how constraints may be added to a ConForm implementation to model both interconnection assumptions of particular programming languages, as well as user-defined design constraints which specify rules of system structuring.

4.1 Modelling Differences Between Versions

In Chapter 3, we noted that all of the versions of a component are grouped into a version family. However, membership in a version family does not imply any enforced commonalities between the versions. For example, different versions of the subsystem DEVICE_SYS may have radically different structures, components, export lists, and requirements. Because of this, there is a need

\footnote{For simplicity, we have assumed that all versions have the same name, which is the same as the name of the version family.}
for techniques that can aid in representing and querying the various kinds of differences between versions of a component. In particular, we are interested in knowing if different versions of a component are compatible with each other. Other kinds of inter-version relations provide useful information, such as the historical, or evolved–from, relation as well as relations based on user-provided annotations. Such techniques are useful not only when “browsing the version space” looking for compatible replacement components, but also when trying to gain an understanding of how a software system has changed throughout its lifetime.

Typically, configuration management systems maintain only one relation between versions of the same component: the historical, or evolved–from relation. Differences between versions are not explicitly considered, except possibly as a textual diff. However, as these entities are members of the same version family, it should not be surprising that there is often a great deal of similarity between versions, in terms of the subcomponents, internal structure, and the provided and required entities. Furthermore, examining how versions of a component have changed can provide valuable information to someone trying to follow the system’s evolution. Therefore, it is useful to consider various ways of representing the possible differences and similarities between versions of the same component.

4.1.1 Representing Differences Textually

As mentioned above, most configuration management systems do not consider differences between versions beyond the textual level. For example, consider the two versions of a stack module shown in Fig. 4.1. The common approach to modelling differences between program components, as exemplified by the RCS utility rcsdiff, is to compare them line-by-line and record any lines where they do not agree character-for-character (Fig. 4.2). This approach considers “benign” changes, such as altering white space or rearranging the order of procedures, to be as significant as wholesale syntactic or structural changes. RCS and SCCS, which are used by many SCM systems to perform version management, use this approach to represent the differences between versions.

\footnote{Of course, arbitrarily rearranging the order of two procedures may not be benign, depending on the programming language and definitions of the procedures. We make the assumption that all modules have been verified by programming-language–specific tools.}
Figure 4.1: Two similar versions of a source program. Both are written in the Object-Oriented Turing programming language and implement a stack of strings. The version on the right (version 1.2) differs from the version on the left (version 1.1) in one important way — the addition of a new exported function, isEmpty — and one minor way — the order of appearance of push and pop within the module has been changed.
Different versions of a program component often have many properties in common; since they are variants of each other, they often share common subcomponents, internal structure, and interconnection information. Because of this, a system such as ConForm that records the structure and interconnections of program components can exploit this knowledge to give a more concise and informative textual representation of the differences between versions.

The ConForm textual notation for modelling the differences between versions of the same component is called C-STD, which stands for ConForm Structured Textual Differencing. A C-STD description of the differences between a pair of component versions consists of:

- the version identifier of the given component, and of the component to which it is being compared;
- a list of the relations that hold between the two versions; and
- a description of how the given component's interface (tab), requirements (slots), and subcomponents (body) differ from those of the other component.

Figure 4.2 shows a simple example of C-STD using the two module versions from Fig. 4.1. The text at the left of Fig. 4.2 shows the output from the RCS utility rcsdiff; SCCS and many other SCM tools use a similar technique. The text at the right shows the C-STD description of the differences between the two module versions. The C-STD description exploits its understanding of the components to present a concise and structured representation of the differences. The Relations clause lists the known inter-version relations between the two component versions: stack module version 1.2 evolved from version 1.1 and is tab compatible, slot identical, and weakly plug compatible with version 1.1.\(^3\) The Interface clause indicates that interfaces of the two versions are the same, except that version 1.2 exports an additional resource named isEmpty. The Resources clause indicates that the two module versions have exactly the same set of subcomponents (resources), except that version 1.2 adds a function named isEmpty whose complete definition is given. Finally, the absence of a Requirements clause indicates that the two module versions have the same resource- and module-level requirements.

The C-STD description of the differences between the two module versions is more concise and informative than the simple textual diff shown on the right. It highlights the differences in

\(^3\)Inter-version relations are discussed in the next section.
CHAPTER 4. COMPATIBILITY AND OTHER ASPECTS OF CONFORM

RCS file: RCS/stack,v
retrieving revision 1.1
retrieving revision 1.2
diff -ri.1 -ri.2
1c1
< % Version 1.1
---
> % Version 1.2
4c4
<   export push, pop
---
>   export push, pop, isEmpty
10a11,16
>   proc pop (var e : string)
>     pre top > 0
>     e := s (top)
>     top := top - 1
>   end pop
>
17,21c23,25
<   proc pop (var e : string)
<     pre top > 0
<     e := s (top)
<     top := top - 1
<   end pop
---
>   function isEmpty : boolean
>     result top = 0
>   end isEmpty

module stack : 1.2
  comparedTo : 1.1
  Relations:
    evolvedFrom,
    tabCompatible,
    slotIdentical,
    weaklyPlugCompatible
  Interface:
    exportList
      adds:
        isEmpty
  Resources:
    adds:
    function isEmpty : boolean
      result top = 0
    end isEmpty

Figure 4.2: Two ways of textually representing differences between versions. On the left is an example of the naive approach which uses a simple textual comparison; here, the RCS utility rcsdiff was used. On the right is an example of C-STD, an approach that ignores "benign" differences between versions, such as altered comments or whitespace, or a change in the order in which the procedures are listed. Since the ConForm system records the structure of the components, a structured view of the differences can be presented.
a structured way. We can see immediately that the major change from version 1.1 to version 1.2 is the addition of a new exported function, whereas the fact that the relative ordering of push and pop has changed is not considered noteworthy.

The details of the C-STD notation are given in the appendix A.

### 4.1.2 Inter-Version Relationships

Another method of representing and reasoning about the differences between component versions is the explicit modelling of relationships based on composition, internal structure, and interconnection information. This information can be derived automatically from the components by program understanding tools (such as a special-purpose parser) and stored within the appropriate version families.

The inter-version relations presented in this section denote various kinds of compatibility and similarity among component interfaces (tabs), requirements (slots), and implementations (bodies). Inter-version compatibility relations are possible at all three levels of components: resource, module, and subsystem.

Previous module interconnection languages (MILs) have considered various kinds of relations between program component versions [87, 59, 68]. However, the relations modelled by these MILs addressed mainly semantic properties of components. Here, we have concentrated on relations that model syntactic and structural commonalities of component versions.

In this section, we present several kinds of inter-version relations modelled by ConForm. Many of the relations are defined here informally, using natural language. A formal definition of these relations is presented in section 5.7.2 in terms of the formal model of ConForm given in the first part of Chapter 5. Some of the relations are formally defined in terms of other, more basic relations. These formal definitions are given in this section.

We have used various annotated mathematical symbols to denote ConForm’s inter-version relations. The “=” symbol is used to indicate that two versions are equivalent in some way. For example, if $M_1$ and $M_2$ are versions of the same module, then $M_1 =_{\text{tab}} M_2$ indicates that they are tab identical. Any relation based on the “=” symbol is an equivalence relation, i.e.,

---

4 These MILs are discussed in more detail in section 4.1.3.

5 This relation and the others mentioned here are defined below.
reflexive, symmetric, and transitive.

The "\( \geq \)" symbol indicates that two versions are compatible in some way. For example, \( M_1 \geq_{\text{slot}} M_2 \) indicates that \( M_1 \) is slot compatible with \( M_2 \). Any relation based on the "\( \geq \)" symbol is reflexive, transitive, and anti-symmetric with respect to the analogous equivalence relation, i.e., it defines a partial ordering on program components of that kind.

Most compatibility relations are indicated by the "\( \geq \)" symbol. When a weaker form of compatibility is also possible, we use the "\( \gtrsim \)" symbol. If "\( \geq_{\text{foo}} \)" and "\( \gtrsim_{\text{foo}} \)" are defined, then it is always the case that \( v_1 \geq_{\text{foo}} v_2 \Rightarrow v_1 \gtrsim_{\text{foo}} v_2 \). Any relation based on the "\( \gtrsim \)" symbol defines a partial ordering on program components of the appropriate kind, i.e., is reflexive, transitive, and anti-symmetric with respect to the analogous equivalence relation.

The "\( \gtrsim \)" symbol is used for relations that denote a weaker form of similarity than compatibility. Any relation based on this symbol is reflexive and transitive, but not necessarily anti-symmetric.

We now consider inter-version relations for each kind of component: resource, module, and subsystem.

**Resource Version Relations**

Relations between resource versions depend on the similarity of the tabs, bodies, and (resource-level) slots. We first define the tab identical relation between resource versions:

**DEFN:** Two resources, \( r_1 \) and \( r_2 \), are said to be tab identical if they have the same name and signature. We write this as \( r_1 =_{\text{tab}} r_2 \).

Resources that are tab identical may be substituted for one another without "breaking" the system structure, although the new resource may have different semantics and a different set of requirements.\(^7\) We also need precise ideas of code and requirements similarity, which we now define:

\(^6\) A relation "\( \geq_{\text{foo}} \)" is said to be anti-symmetric with respect to the equivalence relation "\( =_{\text{foo}} \)" if it satisfies the following: \( \forall v_1, v_2 \cdot v_1 =_{\text{foo}} v_2 \iff v_1 \geq_{\text{foo}} v_2 \land v_2 \geq_{\text{foo}} v_1 \). For example, the resource slot compatible relation ("\( \geq_{\text{slot}} \)") is anti-symmetric with respect to "\( =_{\text{slot}} \)" since resource versions \( r_1 =_{\text{slot}} r_2 \) if and only if \( r_1 \geq_{\text{slot}} r_2 \) and \( r_2 \geq_{\text{slot}} r_1 \).

\(^7\) If additional requirements are introduced by the new resources, they must be resolved.
4. COMPATIBILITY AND OTHER ASPECTS OF CONFORM

DEFN: Two resources, \( r_1 \) and \( r_2 \), are said to be code identical if they have the same body.

We write this as \( r_1 =_{\text{code}} r_2 \).

We do not define the ideas of tab or code compatible at the resource level, as these concepts are programming language specific and many programming languages do not explicitly consider them.

We now define resource version relations based on slot similarity:

DEFN: Two resources, \( r_1 \) and \( r_2 \), are said to be slot identical if they have the same set of requirements. We write this as \( r_1 =_{\text{slot}} r_2 \).

DEFN: Resource \( r_1 \) is said to be slot compatible with resource \( r_2 \) if \( r_1 \)'s requirements are a subset of \( r_2 \)'s. We write this as \( r_1 \geq_{\text{slot}} r_2 \).

Slot compatibility is useful because it implies that if \( r_1 \) is used to replace \( r_2 \) in a module, then no new requirements are added.\(^8\) We note that the following lemma is trivially true:

**Lemma:** \( r_1 =_{\text{slot}} r_2 \iff r_1 \geq_{\text{slot}} r_2 \land r_2 \geq_{\text{slot}} r_1 \)

Thus, the slot compatible relation is anti-symmetric and defines a partial ordering on resources with respect the the "\( =_{\text{slot}} \)" equivalence relation.

With the definitions of resource tab, slot, and code relations, we can define overall resource compatibility relations:

DEFN: Two resources, \( r_1 \) and \( r_2 \), are said to be identical if they are both tab identical and code identical. We write this as \( r_1 = r_2 \).

Formally: \( r_1 = r_2 \iff r_1 =_{\text{tab}} r_2 \land r_1 =_{\text{code}} r_2 \)

We allow for the possibility that the signature of a resource might not be contained within the resource's "code".\(^9\) Therefore, we require that two resources be both code identical and tab identical to be considered identical. We also note that since the resource's tab and code together determine the resource-level slots, the following lemma is trivially true:

---

\(^8\) Note that \( r_1 \geq_{\text{slot}} r_2 \) implies \( \text{reqs}(r_1) \subseteq \text{reqs}(r_2) \). While this might seem counter-intuitive based on the shapes of the symbols, we have consistently used the "\( \geq \)" symbol to mean "compatible". Thus, "resource slot compatible" (\( \geq_{\text{slot}} \)) means "requires the same or fewer resources" whereas "module tab compatible" (\( \geq_{\text{tab}} \)) means "provides the same or more resources".

\(^9\) Whether this is true will depend on the particular programming language.
Figure 4.3: Three versions of a stack pop procedure.

**Lemma:** \( r_1 = r_2 \implies r_1 =_{\text{slot}} r_2 \)

We now define the plug identical and plug compatible relations, which are weaker forms of overall resource compatibility:

**Defn:** Two resources, \( r_1 \) and \( r_2 \), are said to be plug identical if \( r_1 \) is both tab identical and slot identical with \( r_2 \). We write this as \( r_1 =_{\text{plug}} r_2 \).

Formally: \( r_1 =_{\text{plug}} r_2 \iff r_1 =_{\text{tab}} r_2 \land r_1 =_{\text{slot}} r_2 \)

**Defn:** Resource \( r_1 \) is said to be plug compatible with \( r_2 \) if \( r_1 \) is both tab identical and slot compatible with \( r_2 \). We write this as \( r_1 \geq_{\text{plug}} r_2 \).

Formally: \( r_1 \geq_{\text{plug}} r_2 \iff r_1 =_{\text{tab}} r_2 \land r_1 \geq_{\text{slot}} r_2 \)

\( r_1 \geq_{\text{plug}} r_2 \) implies that \( r_1 \) may be freely substituted for \( r_2 \) without worrying about "breaking the compile" or introducing new requirements that must be satisfied. However, plug compatibility does not imply anything about possible semantic differences between the two resource versions (e.g., whether the two resources are code identical). Therefore, while plug compatibility defines a partial ordering on resource versions with respect to the "\( =_{\text{plug}} \)" relation, it does not do so with respect to the overall resource equivalence relation "\( = \)".

Let us now consider an example. Figure 4.3 shows three versions of a stack pop procedure. If we let \( \text{pop}_1 \), \( \text{pop}_2 \), and \( \text{pop}_3 \) denote the three procedure versions from left to right, then we
may observe the following facts:

- The requirements of the three procedure versions are as follows:
  - requirements \((\text{pop}_1)\) = \{top, s\}
  - requirements \((\text{pop}_2)\) = \{top, s, err\}
  - requirements \((\text{pop}_3)\) = \{top, s\}

- \(\text{pop}_1 =_{\text{tab}} \text{pop}_2\) but neither is \text{tab} identical with \(\text{pop}_3\).

- \(\text{pop}_1 \geq_{\text{slot}} \text{pop}_2\) but \(\text{pop}_2 \not\geq_{\text{slot}} \text{pop}_1\) since \(\text{pop}_2\) has an additional requirement of \text{err}.
  Similarly, \(\text{pop}_3 \geq_{\text{slot}} \text{pop}_2\) but \(\text{pop}_2 \not\geq_{\text{slot}} \text{pop}_3\).

- \(\text{pop}_1 =_{\text{slot}} \text{pop}_3\).

- No version is code identical with any other version.

- \(\text{pop}_2 \geq_{\text{plug}} \text{pop}_1\) since \(\text{pop}_1 =_{\text{tab}} \text{pop}_2\) and \(\text{pop}_2 \geq_{\text{slot}} \text{pop}_1\). \(\text{pop}_3\) is not \text{plug} compatible with \(\text{pop}_1\) or \(\text{pop}_2\) since it is not \text{tab} compatible with either.

We note that if we were to change the type of the parameter of \(\text{pop}_3\) from \text{int} to \text{string}, \(\text{pop}_3\) would still not be \text{tab} identical with either \(\text{pop}_1\) or \(\text{pop}_2\) since the names of the respective parameters differ. Modelling relations that allow for variable renaming would greatly complicate an implementation of ConForm; consequently, we have decided not to model such relations. Comments from both researchers and practitioners presented in section 6.2 lead us to believe that this was a reasonable and practical decision.

**Module and Subsystem Version Relations**

Module and subsystem (i.e., unit) relations are based mainly on relations between their subcomponents. Unit-level relations based on \text{tab} information can be defined as follows:

**DEFN:** Two units, \(U_1\) and \(U_2\), are said to be \text{tab} identical if they provide exactly the same set of resource interfaces, i.e., \(U_1\) and \(U_2\) provide the same number of resources, and for each resource \(r_1\) provided by \(U_1\) there is a resource \(r_2\) provided by \(U_2\) such that \(r_1 =_{\text{tab}} r_2\). We write this as \(U_1 =_{\text{tab}} U_2\).
DEFN: Unit $U_1$ is said to be tab compatible with unit $U_2$ if $U_1$'s set of provided resource tabs is a superset of $U_2$'s, i.e., for each resource $r_2$ provided by $U_2$, there is a resource $r_1$ provided by $U_1$ such that $r_1 =_{tab} r_2$. We write this as $U_1 \geq_{tab} U_2$.

The difference between $U_1 =_{tab} U_2$ and $U_1 \geq_{tab} U_2$ is that in the latter case $U_1$ may provide additional resources that are not provided by $U_2$.

Unit-level relations based on slot information (both resource-level and module-level) can be defined as follows:

DEFN: Two units, $U_1$ and $U_2$, are said to be requirements identical if $U_1$'s resource-level requirements are the same as $U_2$'s. We write this as $U_1 =_{req} U_2$.

DEFN: Unit $U_1$ is said to be requirements compatible with unit $U_2$ if $U_1$'s resource-level requirements are a subset of $U_2$'s. We write this as $U_1 \geq_{req} U_2$.

DEFN: Two units, $U_1$ and $U_2$, are said to be module dependency identical if $U_1$'s module dependencies are the same as $U_2$'s. We write this as $U_1 =_{modDep} U_2$.

DEFN: Unit $U_1$ is said to be module dependency compatible with unit $U_2$ if $U_1$'s module dependencies are a subset of $U_2$'s. We write this as $U_1 \geq_{modDep} U_2$.

DEFN: Two units, $U_1$ and $U_2$, are said to be slot identical if they are both requirements identical and module dependency identical. We write this as $U_1 =_{slot} U_2$.

Formally: $U_1 =_{slot} U_2 \iff U_1 =_{req} U_2 \land U_1 =_{modDep} U_2$

DEFN: Unit $U_1$ is said to be slot compatible with unit $U_2$ if $U_1$ is both requirements compatible and module dependency compatible with $U_2$. We write this as $U_1 \geq_{slot} U_2$.

Formally: $U_1 \geq_{slot} U_2 \iff U_1 \geq_{req} U_2 \land U_1 \geq_{modDep} U_2$

We now define unit version relations based on subcomponent and structural similarity. We start with the code identical and resource identical relations.

DEFN: Two modules, $M_1$ and $M_2$, are said to be code identical if they have identical bodies. We write this as $M_1 =_{code} M_2$. 

64
CHAPTER 4.  COMPATIBILITY AND OTHER ASPECTS OF CONFORM

DEFN: Two units, \( U_1 \) and \( U_2 \), are said to be resource identical if they contain the same versions of the same resources. We write this as \( U_1 =_{\text{res}} U_2 \).

We note that \( U_1 =_{\text{res}} U_2 \) does not imply that \( U_1 \geq_{\text{tab}} U_2 \); the "=_{\text{res}}" relation asserts that the two units contain the same set of resources, but it does not specify which are provided to clients via the respective unit interfaces.

We now define resource and structural continuity:

DEFN: Unit \( U_1 \) is said to be resource continuous with unit \( U_2 \) if for each resource version \( r_2 \) contained by \( U_2 \), there is a resource version \( r_1 \) contained by \( U_1 \) that is from the same version family as \( r_2 \). We write this as \( U_1 \preceq_{\text{res}} U_2 \).

DEFN: Two subsystems, \( S_1 \) and \( S_2 \), are said to be structurally identical if they contain the same (top-level) unit component versions. We write this as \( S_1 =_{\text{struct}} S_2 \).

DEFN: Subsystem \( S_1 \) is said to be structurally continuous with subsystem \( S_2 \) if for each unit version \( U_2 \) contained by \( S_2 \), there is a unit version \( U_1 \) contained by \( S_1 \) that is from the same version family as \( U_2 \). We write this as \( S_1 \geq_{\text{struct}} S_2 \).

Resource and structural continuity do not necessarily entail any information about tab, slot, or overall compatibility, but they may be useful to someone trying to gain an understanding of the differences between versions of a module or subsystem.

We note that \( S_1 =_{\text{struct}} S_2 \) does not imply that \( S_1 \geq_{\text{tab}} S_2 \). This relation merely states that \( S_1 \) and \( S_2 \) have the same top-level subcomponents (i.e., module and subsystem versions); it asserts nothing about the interfaces of \( S_1 \) and \( S_2 \).

We now define two relations on subsystems that consider only the modules that are contained anywhere within the subsystems (i.e., as top-level components, or anywhere within top-level subsystems). That is, these relations depend on the flattened views of the subsystems. A flattened view of a subsystem consists of all of its top-level modules plus all of the modules of the flattened view of the top-level subsystems.

DEFN: Two subsystems, \( S_1 \) and \( S_2 \), are said to be module identical if, when flattened, they contain the same module versions. We write this as \( S_1 =_{\text{mod}} S_2 \).
**DEFN:** Subsystem $S_1$ is said to be module continuous with subsystem $S_2$ if for each module version $M_2$ contained within $S_2$, there is a module version $M_1$ contained within $S_1$ that is from the same version family as $M_2$. We write this as $S_1 \preceq_{\text{mod}} S_2$.

Overall unit-level compatibility relations can be defined based on tab and slot relations:

**DEFN:** Two modules, $M_1$ and $M_2$, are said to be identical if they are both tab identical and code identical. We write this as $M_1 = M_2$.

Formally: $M_1 = M_2 \iff M_1 =_{\text{tab}} M_2 \land M_1 =_{\text{code}} M_2$

**DEFN:** Two subsystems, $S_1$ and $S_2$, are said to be identical if they are both tab identical and structurally identical. We write this as $S_1 = S_2$.

Formally: $S_1 = S_2 \iff S_1 =_{\text{tab}} S_2 \land S_1 =_{\text{struct}} S_2$

It should be noted that neither $U_1 =_{\text{res}} U_2$ nor $U_1 =_{\text{struct}} U_2$ implies $U_1 = U_2$, as the tabs of $U_1$ and $U_2$ might be different.

We now define plug compatibility between unit versions: plug compatibility is a weaker form of overall compatibility than the "=" relation.

**DEFN:** Two units $U_1$ and $U_2$ are said to be plug identical if they are both tab identical and slot identical. We write this as $U_1 =_{\text{plug}} U_2$.

Formally: $U_1 =_{\text{plug}} U_2 \iff U_1 =_{\text{tab}} U_2 \land U_1 =_{\text{slot}} U_2$

**DEFN:** Unit $U_1$ is said to be strongly plug compatible with unit $U_2$ if it is both tab identical and slot compatible with $U_2$. We write this as $U_1 \succeq_{\text{plug}} U_2$.

Formally: $U_1 \succeq_{\text{plug}} U_2 \iff U_1 =_{\text{tab}} U_2 \land U_1 \succeq_{\text{slot}} U_2$

Strong plug compatibility asserts that $U_1$ may be replaced by $U_2$ within a system without "breaking the compile". However, this relation does not assert anything about the semantic compatibility of the two unit versions.
DEFN: Unit \( U_1 \) is said to be weakly plug compatible with unit \( U_2 \) if it is both tab compatible and slot compatible with \( U_2 \). We write this as \( U_1 \succeq_{\text{plug}} U_2 \).

Formally: \( U_1 \succeq_{\text{plug}} U_2 \iff U_1 \succeq_{\text{tab}} U_2 \land U_1 \succeq_{\text{slot}} U_2 \)

Weak plug compatibility asserts that \( U_1 \) may be replaced by \( U_2 \) within a system without "breaking the compile" as long as the extra resources provided by \( U_2 \) do not conflict with the names of any other resources within the rest of the system. We note that weak plug compatibility is anti-symmetric with respect to "\( =_{\text{plug}} \)" and thus, like strong plug compatibility, defines a partial ordering on unit versions. Also, we note that weak plug compatibility \( (\succeq_{\text{plug}}) \) is a weaker form of plug compatibility than strong plug compatibility \( (\succeq_{\text{plug}}) \), i.e.,:

Lemma: \( U_1 \succeq_{\text{plug}} U_2 \Rightarrow U_1 \succeq_{\text{plug}} U_2 \)

Like strong plug compatibility, this relation does not assert anything about the semantic compatibility of the two unit versions.

Let us now re-examine the example stack modules presented in Fig. 4.1. If we denote module versions 1.1 and 1.2 as \( \text{stack}_1 \) and \( \text{stack}_2 \) respectively, then we can make the following observations:

\begin{itemize}
  \item \( \text{stack}_2 \succeq_{\text{tab}} \text{stack}_1 \) but \( \text{stack}_1 \nless_{\text{tab}} \text{stack}_2 \), since \( \text{stack}_2 \) provides an extra procedure \( \text{isEmpty} \).
  \item \( \text{stack}_2 =_{\text{slot}} \text{stack}_1 \) since neither module has any unresolved resource- or module-level requirements.
  \item \( \text{stack}_2 \succeq_{\text{plug}} \text{stack}_1 \) since \( \text{stack}_2 \succeq_{\text{tab}} \text{stack}_1 \) and \( \text{stack}_2 =_{\text{slot}} \text{stack}_1 \)
\end{itemize}

Since \( \text{stack}_2 \succeq_{\text{plug}} \text{stack}_1 \), this means that \( \text{stack}_2 \) could be used to replace \( \text{stack}_1 \) in a system without fear of "breaking the compile" as long as it can be determined that \( \text{isEmpty} \) does not conflict with the name of any other resource in the rest of the system.

Other Kinds of Relations

The relations defined in this section are all automatically derivable from the source code of program components. However, these are not the only kinds of relations that can be modelled;
ConForm also supports the modelling of inter-version relations based on user-provided information. The historical or evolved from relation is an example of such a relation; the user indicates the historical parent of each component version as it is checked into the repository, and this information is recorded within the appropriate version family.

A ConForm implementation also allows the definition of relations that are based on user-defined annotations. For example, a relation could be defined that identifies all components that have the keyword `X11--Motif` as an annotation. Relations based on other component characteristics such as time stamp and file size are also possible, but we do not consider them explicitly.

Inter-version relationships are discussed in greater detail in section 5.7.2.

4.1.3 Relation to Previous Work

Several previous MILs have considered possible relationships between component versions. Most such MILs have concentrated on modelling semantic properties of program components, whereas ConForm has concentrated on syntactic and structural properties. We now discuss these MILs in more detail.

Tichy's Intercol [87] was the first system modelling notation to consider relations between versions of the same component. In Intercol, a version family consists of all the versions of a component that implement the same abstract interface. That is, ignoring minor syntactic and programming language differences, all versions of the a component should provide and require the same set of resources. Intercol did not insist on strict syntactic compatibility to allow for minor variations in syntax to be ignored, and because Tichy envisaged that different versions of a component might be written in different programming languages. Using ConForm terminology, any two members of a version family in Intercol must be both tab and slot compatible, albeit at an abstract level. Intercol did not explicitly model any other kinds of relations between component versions.

NuMIL [59] also considered the idea of relations between members of the same version family. Like Intercol, NuMIL considers that all members of a version family should implement the same abstract interface. However, NuMIL also differed from Intercol in several important ways. First, NuMIL considered that the requirements of a program component were an implementa-
CHAPTER 4. COMPATIBILITY AND OTHER ASPECTS OF CONFORM

tion detail and should not be used as a determining factor as to whether components could be members of the same version family; consequently, membership in a version family depended only on the resources provided by each version. Second, in addition to syntactic properties, abstract semantic properties of (procedure) resources were modelled via pre- and post-conditions. Third, the abstract interface of a version family was considered to represent only a minimal set of common features; individual versions could also provide additional resources. Because of this, version family members were not freely substitutable for each other. Therefore, NuMIL introduced the idea of “upward compatibility” as a possible relation between members of the same version family. Roughly speaking, a component is upwardly compatible with another if it is tab and slot compatible, and if each procedure of the first component is semantically compatible with a procedure of the second (i.e., the pre-condition is the same or stronger, and the post-condition is the same or weaker). Upward compatibility was the only inter-version relation considered by NuMIL.

Perry’s Inscape [67] explored several kinds of inter-version relations, mostly based on semantic properties. Like NuMIL, Inscape modelled (procedure) resource semantics via pre- and post-conditions. Additionally, Inscape added the semantic idea of a procedure obligation, which models a condition that must be satisfied eventually. For example, a call to a procedure that opens a file might oblige a later call to another procedure that closes the file. Perry defined several different kinds of inter-version relations that expanded on these semantic ideas.

Some recent research and commercial software configuration management (SCM) systems, such as CAPITL, Jason, and Aide de Camp, allow user-defined inter-version relations to be added. Support for modelling such relations is limited; most of the work must be done by the user by hand. Most such systems do not model semantic properties of components.

Like recent SCM systems, ConForm does not explicitly model semantic aspects of components. Determining semantic equivalence is, in general, an undecidable problem, and industrial experience suggests that modelling semantic properties usually does not reward the significant effort that is required.

ConForm’s treatment of inter-version relations differs from previous MILs and SCM systems. ConForm models resources, modules, and subsystems as uniformly as possible; for example, various kinds of plug-compatibility between components can be defined at the resource, module,
or subsystem level. Most of the other approaches treat modules as the only kind of component worth considering in detail. Also, since the ConForm system understands the structure and interconnections of the systems its models, it can provide automated support for relations based on this information. This kind of support is not present in other systems.

4.2 Additional Constraints

One of the strengths of ConForm is that it supports an abstract or generic view of software components and how they may be combined into systems. The ConForm system modelling notation is programming language independent, and ConForm does not insist on a particular design methodology. However, a ConForm implementation does allow the addition of constraints to its system modelling notation; in this way, system models can be made more finely grained, either to incorporate interconnection assumptions of particular programming languages, or to enforce desired design constraints, which are rules of system design that are usually not enforceable at the programming language level. We now discuss each of these issues.

4.2.1 Modelling Different Programming Languages

As mentioned above, the basic ConForm data model defines resources, modules, and subsystems as abstract programming components, independent of the implementation language. Additionally, ConForm has had the design goal of supporting an intelligent approach to system modelling; therefore, it is advantageous to consider adding interconnection assumptions about the particular programming languages in which systems are written. To this end, ConForm allows the addition of constraints to its basic data model that are programming language specific. That is, ConForm allows the definitions of resources, modules, and subsystems to be extended to more closely model the features and assumptions of particular implementation languages.

Sets of constraints for the C and OOT programming languages have been defined and are formally specified in section 5.5. We now briefly discuss the ConForm model of C programs.
Figure 4.4: A simple C source program. There are three files shown here: at the left is the main program main.c, at the top right is the interface module stack.h, and below it is its implementation module stack.c. Both main.c and stack.c also import the interface stdio.h which is not shown. stdio.h is implemented by the standard C library, libc.a, which is automatically linked in by the compiler.
The ConForm Model of C

The C programming language [43] is one of the most commonly used implementation languages in industry. However, the low level of abstraction that is fundamental to C can make the creation of large systems difficult and error prone. Because of this, many programming conventions have come into general use which, although not part of the language per se, entail a more disciplined and structured style of programming. For example, one such convention is the separation of C programs into conceptual modules. Each conceptual module is broken down into two parts: the interface, which is stored in a file whose name has a .h extension, and the implementation, which is stored in a file whose name has a .c extension.

To make our modelling of C programs simpler, we assume that the convention of interface/implementation separation is followed. The ConForm constraints for the C programming language define three kinds of modules: interfaces, implementations, and abstract source modules. Interface modules correspond to “dot h” files, and implementation modules correspond to “dot c” files. An abstract source module does not correspond to a file of source code, but rather to a pair of files, one an interface and the other its implementation. For example, Fig. 4.4 shows the source code for a simple C program that consists of three files: the main program main.c, the interface file stack.h, and its implementation file stack.c. We consider that stack.h and stack.c comprise an abstract source module named stack.10

C libraries are modelled as specialized subsystems that contain several abstract source modules, whose interfaces they provide to clients (the corresponding module implementations are considered to be opaque). In Fig. 4.4, the standard C library libc.a is represented as a subsystem that implements the abstract modules stdio and several others.

The ConForm model of C considers that the include statement represents one of two different kinds of module-level dependencies:

1. the implements relation between an implementation module (.c file) and a similarly named interface module (.h file), and

10Some interface modules do not require corresponding implementation modules if they define only types and constants. Likewise, an implementation module that contains a main program, such as main.c in Fig. 4.4, does not require a corresponding interface module. For the sake of consistency, we assume that such modules do exist, even if they have no contents. Thus, the file main.c and the empty file main.h comprise the abstract source module main.
CHAPTER 4. COMPATIBILITY AND OTHER ASPECTS OF CONFORM

2. the imports relation between an implementation module and an interface module, or between two interface modules.

The ConForm constraints for C also model interconnection assumptions and common programming conventions. For example, the ConForm definition of a C interface module asserts that such a module may not implement another module, and that it may define functions only as stubs (or "prototypes"). Similarly, the ConForm definition of a C implementation module asserts that an implementation module may implement at most one other module, and that any functions it defines must be fully declared. The ConForm definition of a C subsystem adds constraints to model other interconnection assumptions of the language. For example, no two abstract source modules may define and provide identically-named resources.

Figure 4.4 shows the source code for a simple C program consisting of three files: the main program main.c, the interface file stack.h, and its implementation file stack.c. Additionally, main.c and stack.c import the I/O library interface stdio.h that is implemented by the standard C library libc.a. We consider that there is an abstract source module called main comprised of the implementation file main.c and the (null) interface file main.h. The file main.c includes the interface files stack.h and stdio.h; we consider that the abstract module main imports the abstract modules stack and stdio. stack is comprised of the interface file stack.h and the implementation file stack.c. The abstract source module stdio is implemented by the standard C library libc.a, which we consider to be a subsystem that also implements several other abstract modules. Figure 4.5 shows a visual representation of how this program is modelled visually by ConForm.

4.2.2 Design Constraints

ConForm supports the addition of design constraints, which are rules of system design that are not generally enforceable at the programming language level.\textsuperscript{11} For example, a subsystem is an extra-programming-language abstraction used to group units of source code (and other subsystems) into a meaningful whole. Subsystems can be considered to have interfaces derived from those of their components. One possible design constraint involving subsystems is the

\textsuperscript{11}Mancoridis has made a study of design constraints, which he calls architectural scoping rules [51].
Figure 4.5: The ConForm visual representation of the C program presented in Fig. 4.4. We have omitted resource-level information to draw attention to how ConForm models C files as modules and libraries. The abstract source module main is comprised of the C program file main.c: we consider that the interface file main.h exists but has no contents. main imports the abstract modules stack and stdio. The files stack.h and stack.c comprise the abstract source module stack: stack imports the stdio module. The stdio module is implemented within the C standard library libc.a, which is represented as a subsystem. libc.a also implements other abstract modules, such as assert: this is indicated by the set of module tabs at the right.
"One-level export rule" — If a subsystem $S_1$ contains another subsystem $S_2$ that contains a module $M$, then $S_2$ may export $M$, but $S_1$ may not export $M$. This rule encourages high cohesion within subsystems; it prohibits deeply nested components from being exported "up" multiple levels in a subsystem hierarchy. For example, recall the structure of the subsystem DEVICE_SYS shown in Fig. 3.8. DEVICE_SYS provides modules Tty and DeviceDriver to its clients.\footnote{Non-exported modules Disk and DiskMutex are not available to clients of DEVICE_SYS.} Let us suppose that DEVICE_SYS is a subcomponent of another subsystem, call it S. The "One-level export rule" entails that Tty and DeviceDriver may be used by other subcomponents of S, but S may not export them to its clients.

The ability to define and enforce design constraints within a ConForm implementation is consistent with ConForm's goal of supporting a generic yet configurable approach to system modelling. Several example design constraints are discussed and defined formally in section 5.6.

4.3 Summary

The previous chapter introduced the ConForm notation for versioned system modelling. That chapter showed how ConForm supports both the snapshot and the evolutionary view of systems at three levels of abstraction (resource, module, and subsystem), and provides a visual interpretation of both views. In this chapter, we have considered the modelling of various kinds of relations between versions of the same component, based on various kinds of tab, slot, and body compatibility. Also, we showed how the ConForm generic model of program components can be extended to incorporate interconnection constraints of implementation languages, as well as user-defined design constraints. In the the next chapter, we present a formal definition of the concepts that have been presented in this chapter and the previous one.
Chapter 5

A Formal Model of ConForm

5.1 Introduction

In the previous two chapters, we motivated the need for an approach to system modelling that supports versioning and visual manipulation of system components, both from the "snapshot" and the "evolutionary" points of view. We informally described the data model of ConForm, how versions of components may be modelled, and how constraints may be added to model both programming language interconnection information as well as particular rules of system design that a company might wish to enforce.

In this chapter, we present the formal specification of the ConForm notation for modelling versioned software system components. We begin by describing our motivation for the creation of the formal specification. Then, we give a brief overview of Z, the specification language that we used to specify ConForm. The remainder of the chapter presents the formal specification of ConForm: section 5.4 formally defines the basic ConForm data model of resources, modules, subsystems, and their sub-parts; sections 5.5 and 5.6 show respectively how programming language and design constraints may be added to the formal model; section 5.7 formally defines component versions, version families, and repositories; and section 5.8 defines our calculus for system construction.

We recognize that the reader may be unfamiliar with the specification language we have used, and we concede that he or she may not wish to invest the effort required to understand the formal specification of ConForm in its full detail. To aid reader understanding, we have provided
an overview of the Z specification language in section 5.3, and throughout the specification we have given an ongoing prose explanation of the formalism details. The reader may wish to skim through the central part of this chapter at first reading, returning later to study the details.

We now discuss our motivation and goals for creating a formal specification of ConForm.

5.2 Motivation

We have created a formal specification of the ConForm system modelling notation for several reasons:

1. We wanted to show that the ConForm notation is formalizable; that is, that the basic data model, and ideas of inter-version relationships, programming language assumptions, design constraints, and a calculus of system construction can all be expressed in mathematically precise terms.

2. We wanted to provide a precise definition of the ConForm notation; the specification may be considered to constitute a set of formal documentation of the ConForm system modelling notation.

3. Such a specification can serve as a design document for the creation of an executable ConForm prototype. We have created a prototype implementation, called Proto-ConForm, written in the Object-Oriented Turing programming language [36]. A key consideration was the availability of a package of abstract data structures called Abstur (Abstract Turing) [29, 64]. The Abstur package implements mathematical data types — such as sets, sequences, and maps — which are the underlying units of abstraction in the Z specification language. The availability of such a package helped to minimize the conceptual distance from specification to implementation. As mistakes were recognized and design decisions changed, it was easy to keep the formal definition and prototype implementation up-to-date and mutually consistent.

An additional reason for formally specifying ConForm was that we wished to gain experience with the Z specification language. We wished to test Z's ease of use and evaluate its
CHAPTER 5. A FORMAL MODEL OF CONFORM

general utility in specifying a large, complex software system. We found that Z worked well in
the preliminary stages of specification when the data structures were fairly simple. However,
as the system grew larger and the logical inter-dependencies became more complex, it became
increasingly difficult to model some important concepts, such as recursive structures and polymorphism, using Z. As a result, there are some differences between “standard” Z and the dialect
used here. We discuss this issue in more detail in appendix B.

We have not mechanically checked the final version of the specification presented here.
An early version of the specification, which was written in “standard” Z, was checked by the
research tool ZTC [41]. However, as discussed in appendix B, we eventually decided to adopt a
variant of the Z specification language for which no tools exist (apart from typesetting macros).

5.3 Basics of Z

This section provides a short introduction to Z, a widely-used language for formal specification
[83] that is based on set theory. The reader who is already familiar with Z may wish to skip
to section 5.4. The reader who desires more information may wish to survey the plethora of Z
reference materials that exist [20, 21, 22, 83, 84].

We begin by giving a short overview of modelling data entities in Z. Additional information
on modelling procedures/operations in Z is given in section 5.8.1.

The fundamental entity in a Z specification is the schema. A schema has a name, a
“data” part, and an “assertions” part. The data part describes the various parts that com-
prise the schema’s state, and the assertions list invariants on the state. For example, consider
the ModDeps schema below:\footnote{The definition of ModDeps and the other examples given in this section are taken from the next section; the
definitions are repeated there in their proper context.} its name is ModDeps, it has three data components (modDeps, implements, and imports which are finite sets of IDs), and its (single) assertion defines an invari-
ant that must always be true of any instance of the schema ModDeps, i.e., modDeps is equal to
the union of implements and imports.
A schema may expand another schema by referencing it in the "data part".\(^2\) For example, the schema UnitReq given below extends both the schemas ModDeps and Req. The semantics of schema inclusion is that the data parts of all included schemas are also considered to be part of the new schema, and the assertions of included schemas are implicitly conjoined with the new schema's assertions. Since UnitReq includes schema ModDeps, it has components named modDeps, imports, and implements, and they obey the invariant that \(\text{modDeps} = \text{imports} \cup \text{implements}\).

\[
\text{ModDeps}
\begin{align*}
\text{modDeps}, & \text{ implements, imports : F ID} \\
\text{modDeps} & = \text{implements} \cup \text{imports}
\end{align*}
\]

\[
\text{Req}
\begin{align*}
\text{Entity} & \\
\text{moduleID} & : \text{ID} \\
\text{modKind} & : \text{moduleKind} \\
\text{ModDeps} &
\end{align*}
\]

\[
\text{UnitReq}
\begin{align*}
\text{Req} & \\
\text{moduleID} & : \text{ID} \\
\text{modKind} & : \text{moduleKind} \\
\text{ModDeps} &
\end{align*}
\]

A schema may also define or reference instances of other schemas: for example, ResStub (defined below) contains a finite set of instances of the Req schema. The instances of the named schemas obey their definition, but neither the data part nor the assertions of the referenced schemas are considered to be part of the new schema.

\[
\text{ResStub}
\begin{align*}
\text{ResIntf} & \\
\text{requires} & : \text{F Req}
\end{align*}
\]

In addition to schemas, Z allows "given types", user-constructed types, and axiomatic definitions. Given types are types that are considered to be atomic within the context of a specification; SIGNATURE and CODE below are examples. User-constructed types explicitly list or describe all possible values; CHAR and ID below are examples. Axiomatic definitions

\(^2\)A schema may also be referenced in the assertions part of a schema. This is described in section 5.8.3.
allow for a kind of global data entity declaration together with a possible invariant; the special identifier **NULL** given below is an example.

\[
\text{[\text{SIGNATURE, CODE}]}
\]

\[
\text{CHAR} \ := \ \text{'a'} \ | \ \cdots \ | \ \text{'z'} \ | \ \text{'A'} \ | \ \cdots \ | \ \text{'Z'} \ | \ \text{'0'} \ | \ \cdots \ | \ \text{'9'} \ | \ \cdots \ | \ \text{'}/\ | \ \cdots
\]

\[
\text{ID} \ := \ \text{seq} \ \text{CHAR}
\]

\[
\text{NULL} : \text{ID}
\]

\[
\text{NULL} \ = \ \text{(})
\]

We now present the formal definition of the ConForm notation.

### 5.4 Program Components

In this section we define the basic ConForm data model of program components (i.e., resources, modules, and subsystem). We begin by defining some elementary constructs that are used throughout the formal definition. Then, we define ConForm resources and their sub-parts. Finally, we define ConForm units (i.e., modules and subsystems) and their sub-parts.

#### 5.4.1 Basic Entities and Requirements

In our Z definition of ConForm, we consider resource signatures (i.e., the Z type **SIGNATURE**) and programming language code (i.e., the Z type **CODE**) to be atomic entities, as their details vary between programming languages. Consequently, we specify **SIGNATURE** and **CODE** as "given types" in Z.

\[
\text{[\text{SIGNATURE, CODE}]}
\]

Programming languages often constrain what constitutes a legal identifier (i.e., the type **ID**). We will ignore these considerations and will consider an identifier to be simply a sequence of characters.

\[
\text{CHAR} \ := \ \text{'a'} \ | \ \cdots \ | \ \text{'z'} \ | \ \text{'A'} \ | \ \cdots \ | \ \text{'Z'} \ | \ \text{'0'} \ | \ \cdots \ | \ \text{'9'} \ | \ \cdots \ | \ \text{'}/\ | \ \cdots
\]

\[
\text{ID} \ := \ \text{seq} \ \text{CHAR}
\]
NULL is a special identifier to name entities that are under construction. It is effectively the empty string; formally, it is defined to be a null sequence of characters.

\[
\begin{align*}
\text{NULL} & : ID \\
\text{NULL} & = ()
\end{align*}
\]

Different implementation languages have different kinds of "resources" and "modules". For example, in the C programming language, we consider that there are four kinds of resources (functions, variables, constants, and typedefs) and three kinds of modules (interfaces, implementations, and abstract source modules). The types resourceKind and moduleKind will consist of different values depending on the implementation language being modelled, so we do not specify the possible values here. We introduce these types here as they are used within the definitions of resources, modules, and subsystems.

\[
\begin{align*}
\text{resourceKind} & ::= \cdots (\text{language dependent}) \cdots \\
\text{moduleKind} & ::= \cdots (\text{language dependent}) \cdots 
\end{align*}
\]

An Entity is something that has a name, such as a requirement, a resource signature or implementation, or a unit. Many subsequent schema definitions are based on the Entity schema.

\[
\begin{align*}
\text{Entity} & \\
\text{name} & : ID
\end{align*}
\]

ModDeps are the module-level requirements or dependencies of a unit, i.e., the "red slots" in the visual representation of a ConForm module or subsystem. When we introduced the idea of module-level requirements in section 3.3.2, we stated that they correspond to the explicitly-stated dependencies on other modules that are listed in import, include, inherit, and similar programming language statements. We now group these dependencies into two sub-categories, which we model as sets within the ModDeps schema:

1. implements, which consists of the names of all modules that the current module implements or inherits from, and

---

3 The next section presents examples from the C and Object-Oriented Turing programming languages.
4 ModDeps forms part of the definition of Unit below.
CHAPTER 5. A FORMAL MODEL OF CONFORM

2. imports, which consists of the names of all modules that the current module uses, or imports.

We consider that the implements relation models not only programming languages where a module’s interface is typically separate from its implementation (such as in C and Modula), but that it also models inheritance between classes in an object-oriented language (such as OOT). We consider that imports models all other kinds of “use” dependencies between modules.

\[
\text{ModDeps} \\
\text{modDeps, implements, imports : } \mathbb{F} \text{ ID} \\
\text{modDeps = implements } \cup \text{ imports}
\]

Req models a resource-level requirement of a resource, module, or subsystem, i.e., a “green slot” in the visual representation of a ConForm program component. Often, when trying to determine if a potential resource interconnection is legal, some information about the defining module is also necessary. Thus, we annotate resource-level requirements of modules to include the name and kind of the requiring module, plus the module’s dependencies.

\[
\text{UnitReq} \\
\text{Req} \\
\text{moduleID : ID} \\
\text{modKind : moduleKind} \\
\text{ModDeps}
\]

5.4.2 Resources

We now formally define ConForm resources and their sub-parts.

A ResInf models the interface of a resource, i.e., a “green tab”. It consists of the resource’s name, signature, and resource kind (e.g., variable, procedure, etc.). A UnitResInf extends a resource interface to include the name of the module in which the resource is defined; this annotation makes it easier to reason about when resource-level interconnections are legal.
A resource interface may be further expanded into one of two kinds of resource definitions: a resource stub (such as a C function prototype), or a resource implementation (such as a C function body). In either case, we assume that the resource interface is part of the resource definition, although this is not a strict requirement of some program languages.

A resource stub (i.e., ResStub) consists of a resource interface plus a set of requirements that correspond to the unresolved names within the resource’s signature, such as user-defined types, i.e., a resource stub consists of the resource tab as well as the slots that are derived from the signature. A UnitResStub is further annotated by the name of the module in which the resource stub is defined.

A resource implementation (i.e., ResImpl) consists of a resource interface, the code that implements the resource (i.e., the resource “body”), and a set of requirements that correspond to the unresolved names within the resource definition. A UnitResImpl is further annotated by the name of the module in which the resource implementation is defined.

We now define a resource definition (i.e., Resource) as either a resource stub or a resource implementation. This is the formal definition of a resource in ConForm.
Analogously, a UnitResource is either a UnitResStub or a UnitResImpl, i.e., a Resource annotated with the name of the defining module.\(^5\)

\[
\text{Resource} \equiv \text{ResStub} \cup \text{ResImpl} \\
\text{UnitResource} \equiv \text{UnitResStub} \cup \text{UnitResImpl}
\]

UnitResources models the set of all resources defined within a unit. If the unit is a module, this corresponds to the set of resources defined immediately within the module; if the unit is a subsystem, it corresponds to the set of all resources exported by other (top-level) contained units.

In general, a unit will have a set of resource stubs and implementations. A resource implementation may supply the body for a resource stub; this is modelled by the relation implBindings. When and where a resource binding is legal depends on the underlying implementation language; thus, we give only a weak requirement that if a resource implementation implements a resource stub, then they must share the same interface (i.e., name and signature). Visibility constraints are not considered at this level as conventions vary according to the particular programming language.

It should be noted that most object-oriented languages support \textit{procedural overriding}; that is, it is permissible for more than one resource implementation to implement a resource stub within a system. It should also be noted that some programming languages do not require that all resource stubs be implemented.

\[
\begin{align*}
\text{UnitResources} \\
\text{resDefs} : \mathcal{F} \text{UnitResource} \\
\text{resStubs, unimplResStubs} : \mathcal{F} \text{UnitResStub} \\
\text{resImpls} : \mathcal{F} \text{UnitResImpl} \\
\text{implBindings} : \text{UnitResImpl} \leftrightarrow \text{UnitResStub} \\
\text{resDefs} &= \text{resStubs} \cup \text{resImpls} \\
\text{dom implBindings} &\subseteq \text{resStubs} \\
\text{ran implBindings} &\subseteq \text{resImpls} \\
\text{unimplResStubs} &= \text{resStubs} \setminus \text{dom implBindings} \\
\text{implBindings} &\subseteq \{(r_1, r_s) : \text{resImpls} \times \text{resStubs} \mid r_s.(\theta\text{ResInf}) = r_1.(\theta\text{ResInf})\}
\end{align*}
\]

\(^5\)In the standard reference for the C programming language [43], Kernighan and Ritchie use the term \textit{definition} to mean what we have defined as “resource implementation”, and the term \textit{declaration} to mean what we have defined as “resource stub”.

84
The resource-level requirements of a unit \((i.e., \text{UnitReqs})\) consists of:

- the raw requirements \((i.e., \text{rawReqs})\), which consist of all of the resource-level requirements of all of the contained components (whether resolved or not),

- the resolved requirements \((i.e., \text{resolvedReqs})\), which consist of the resource-level requirements that are resolved within the unit by a contained resource, and

- the unresolved requirements \((i.e., \text{requires})\), which consist of the \text{rawReqs} that are not resolved within the unit, and are thus requirements of the unit.

The set \text{requires} corresponds to the "green slots" in the visual representation of a ConForm module or subsystem.

What it means for a resource requirement to be resolved by a resource will depend on the programming language. Here, we insist only that the name of the resource match the name of the requirement. This condition may be strengthened by programming-language-specific constraints.

\[
\begin{align*}
\text{UnitReqs} & \quad \text{rawReqs, resolvedReqs, requires : } \mathcal{F} \text{ UnitReq} \\
\text{reqBindings} & \quad \text{UnitResource \leftrightarrow UnitReq} \\
\text{requires} & = \text{rawReqs} \setminus \text{resolvedReqs} \\
\text{resolvedReqs} & = \text{ran reqBindings} \\
\text{reqBindings} & \subseteq \{(r, q) : \text{UnitResource} \times \text{UnitReq} \mid q \in \text{rawReqs} \land r.\text{name} = q.\text{name}\}
\end{align*}
\]

5.4.3 Units: Modules and Subsystems

We now formally define ConForm units and their sub-parts.

A ConForm unit \((i.e., \text{Unit})\) is a module or subsystem. Modules contain resources, which they may provide for use by other modules.\(^6\) Subsystems contain modules and other subsystems; a subsystem may also provide a set of resources, but each such resource must correspond to a resource that is provided by one of the (top-level) contained units. That is, a subsystem \(S\) may provide a resource \(R\) only if there is a module \(M\) that defines and provides \(R\) and either \(M\) is a top-level component of \(S\) or \(M\) is contained somewhere within a top-level subsystem subcomponent of \(S\).

\(^6\)Conceptually, we consider that only the interfaces of the resources are made visible.
The schema Unit defines the features of units that are common to modules and subsystem; the schemas Module and Subsystem expand on this schema.

Modules may have stated dependencies on other modules (i.e., ModDeps as defined above). Subsystems may also have module-level dependencies which correspond to the unresolved module-level dependencies of their subcomponents. The two assertions in Unit state that (a) each provided resource interface must correspond to a contained resource, and (b) any resolved resource requirements must be resolved by contained resources.

In terms of our visual model from Chapter 3, the modDeps component of ModDeps models the red (module-level) slots of a unit, the requires component of UnitReqs models the green (resource-level) slots, the resDefs component of UnitResources models the contained resources that comprises the body, and provides models the provided green (resource) tabs.

\[
\begin{align*}
\text{Unit} \\
\text{Entity} \\
\text{ModDeps} \\
\text{UnitReqs} \\
\text{UnitResources} \\
\text{provides} : F \text{UnitResInf} \\
\{ p : \text{provides} \circ p.(\theta \text{ResInf}) \} \subseteq \{ r : \text{resDefs} \circ r.(\theta \text{ResInf}) \} \\
\text{dom reqBindings} \subseteq \text{resDefs}
\end{align*}
\]

A module (i.e., Module) is a unit that defines and contains resources; a subsystem may also be considered to "contain" resources but any such resource must be defined (and provided) by a module contained at some level within the subsystem.

The code of a module corresponds to the (contained) resource definitions, plus "wrapper" information, as well as any appropriate module initialization.\(^7\) As mentioned above, there are several kinds of modules; however, what is considered to be a module will vary according to the programming language.

The assertions of the Module definition state several immediate facts. First, a module may not "depend" on itself (since module dependencies list the other modules that a module depends on). Also, all of the resources, provided resource interfaces, and module requirements are considered to be annotated with the module's name; this extra information is added to

\(^7\)Thus, there is some overlap between the module's code, and the code of the contained resources.
allow for the convenient modelling of constraints on requirement resolution at the Subsystem level. Finally, the resource-level requirements of the module must be a superset of the set of all unresolved resource-level requirements of the contained resources; the other resource-level requirements of the module correspond to requirements of the module's initialization code, if any.

\[
\begin{align*}
\text{Module} & \\
\text{Unit} & \\
\text{kind} & : \text{moduleKind} \\
\text{code} & : \text{seq CODE} \\
\text{name} & \notin \text{modDeps} \\
\forall \text{req} & : \text{rawReqs} \bullet (\text{req.moduleID} = \text{name} \land \text{req.modKind} = \text{kind} \land \text{req.(\theta ModDeps)} = \theta \text{ModDeps}) \\
\forall \text{prov} & : \text{provides} \bullet \text{prov.moduleID} = \text{name} \\
\forall \text{res} & : \text{resDefs} \bullet (\text{res.moduleID} = \text{name} \land \text{res.code in code}) \\
\{ q : \text{Req} \mid (\exists \text{res} : \text{resDefs} \bullet q \in \text{res.requires}) \} & \subseteq \{ uq : \text{UnitReq} \mid uq \in \text{rawReqs} \bullet uq.(\theta \text{Req}) \}
\end{align*}
\]

\[\text{ModuleOrSubsystem} \equiv \text{Module} \cup \text{Subsystem}\]

A subsystem (i.e., Subsystem) contains a set of modules and other subsystems. The set components models the top-level subcomponents of the subsystem. Because subsystems are typically not entities of the underlying programming language, we are often interested in reasoning about the set of all modules contained anywhere within a given subsystem, i.e., the top-level modules, and those transitively contained within any top-level subsystems; allModules models the set of all modules transitively contained anywhere within a subsystem.

A subsystem may provide modules that it contains. A provided module must either be a top-level subcomponent, or be provided by a top-level subsystem. provModules denotes the set of modules provided by the subsystem.

A subsystem may also provide resources belonging to top-level components. The set provides (inherited from the Unit schema definition) models this. However, often we need to be able to reason about the set of all resources provided by any module anywhere within the subsystem; allProvResources models this. Also, for a resource to be provided by a subsystem, the module in which the resource is defined (and originally provided) must also be provided by the subsystem. If a subsystem provides a module, this indicates permission for the subsystem to also provide
any resources provided by the module.

```
Subsystem

Unit
components : F ModuleOrSubsystem
allModules, provModules : F Module
allProvResources : F UnitResource

resDefs = \{ r : UnitResource | (\exists C : component \bullet r \in C.resDefs \land r.(\emptyset UnitResInf) \in C.provides)\}
rawReqs = \bigcup\{ C : components \bullet C.requires\}
allProvResources = resDefs \cup (\bigcup\{ S : components \cap Subsystem \bullet S.allProvResources\})
allModules = (components \cap Module) \cup (\bigcup\{ S : components \cap Subsystem \bullet S.allModules\})
provModules \subseteq ((components \cap Module) \cup (\bigcup\{ S : components \cap Subsystem \bullet S.provModules\}))
\forall p : provides \bullet (\\exists M : provModules \bullet M.name = p.moduleID \land p \in M.provides)
modDeps = \bigcup\{ C : components \bullet C.modDeps\}
\quad \bigcup\{ mID : ID | (\exists S : components \cap Subsystem \bullet mID \in S.provModules)\})
```

Many programming languages separate shared (i.e., provided) resources into stubs and implementations. We consider that either a resource stub or a resource implementation may fulfill a resource requirement. It should be noted, however, that just because a requirement has been fulfilled, it does not imply that the resource has also been implemented. Thus, implBindings and reqBindings denote two very different ideas.

The general condition for a system to be complete is that there must be no unresolved resource requirements or missing modules. It should be noted that some programming languages allow resource stubs to be defined without being implemented. The C language considers the static use of an unimplemented stub to be an error. Object-Oriented Turing checks for unimplemented resource stubs at run-time, consequently it is statically legal to access an unimplemented resource.

```
CompleteSystem

Subsystem
requires = \emptyset
modDeps = \emptyset
```
5.5 Example Programming Language Constraints

As discussed previously, one of the strengths of ConForm is that it takes a generic approach to system modelling. The basic ConForm data model, most of which was formally defined in the previous section, is programming language independent. However, this basic data model can be extended to incorporate structural and interconnection assumptions of common programming languages, such as C. This additional information aids in creating more finely grained system models, allowing more detail about systems written in these languages to be modelled.

In this section, we extend the basic component definitions to model two example programming languages: the C programming language [43], and Object-Oriented Turing (OOT) [36]. We have chosen these languages because they support two very different views of programming. C is a relatively small and simple language that supports only low-level programming abstractions; C is commonly used in industrial applications. OOT is a clean, high-level programming language with strong support for information-hiding and object-oriented programming. Its main use has been in software engineering research and in education. It is our intention to demonstrate that ConForm can be used to model systems written in a low level language such as C, as well as a high level language such as OOT.

We now present the ConForm model of the C and OOT programming languages. We also make some assumptions about the particular programming language features and conventions used; these assumptions are noted in the individual sections.

5.5.1 C Model

The C programming language [43] is one of the most common in industrial use. However, despite a recent ANSI standardization, C contains many features that are either implementation dependent or not well defined. Additionally, the low level of abstraction that is characteristic of C makes writing large programs difficult and error prone unless reasonable defensive programming conventions are followed. These conventions, such as the separation of conceptual or “abstract” modules into interfaces (i.e., .h files) and implementations (i.e., .c files) are well known, but are not enforced by the language. Also, the C language has features whose undisciplined use can make large programs difficult to understand and maintain, such as unrestricted
use of macros (i.e., \#defines), conditional compilation instructions embedded within the code (i.e., \#IFDEFs and \#IFNDEFs), and implicit type cheats (e.g., a function is assumed to return an int value if its prototype is not visible within the scope of its use). We therefore make several assumptions about the use of C.

- No macros (i.e., \#defines) are used, except to define explicit constants.\(^8\)
- No \#IFDEFs or \#IFNDEFs are used.
- Correct and complete type information is used in assignment statements and in function parameters and return values. No implicit "type cheats" are used.
- The common .c/.h file name programming convention is strictly observed:
  - For each program file M.c that contains C resource definitions, there is a file M.h that contains the interfaces of the resources defined in M.c that are provided to clients.\(^9\)
    We consider that the pair of files M.h and M.c comprise an "abstract" source module M; we call M.h the interface module and we call M.c the implementation module.\(^10\)
  - Each implementation module (i.e., .c file) includes its interface module (i.e., .h file). We consider that the implementation module implements the interface module.
  - If an interface or implementation module requires a resource provided by another module, it must include the appropriate .h file. We consider that the module imports the .h file. Note that import relationships between interface modules is allowed.
  - No .c file may include another .c file.

We now extend the ConForm definitions of resources, modules, and subsystems to model the C language.

\(^8\)For the purposes of this discussion, resources defined using the ANSI C const construct are considered to be variables rather than constants.

\(^9\)The syntax of C does not permit the separation of constant and type definitions into interfaces and implementation. Therefore, exported constants and types are actually fully defined within the .h file.

\(^10\)Interface modules containing only type and constant definitions do not require a corresponding implementation module, since these resources are fully defined within the interface module. For consistency, we consider that a corresponding implementation module does exist, but has no contents. Similarly, a .c file that is a main program does not require an interface file, but we consider that a trivial one does exist.
C Resources

We begin by defining the Z type resourceKind, which we left undefined in section 5.4.1. We consider that there are four kinds of resources in C: functions, variables, type definitions (typedefs), and constants.

\[ \text{resourceKind} ::= \text{function} | \text{variable} | \text{typedef} | \text{constant} \]

A C function or variable may be shared between modules, or hidden within a single module. If a function is to be shared between modules, then its interface or prototype is declared in an interface module (i.e., a .h file), and its implementation or body is given within the corresponding implementation module (i.e., .c file). If a variable is to be shared between modules, then it is declared using the C extern construct in an interface module, and its full definition is given within the corresponding implementation module.

If a function or variable is to be hidden within a module, then its definition is given within an implementation module, preceded by the keyword static. No mention of such a function or variable should occur within the corresponding interface module.

Constants in C are somewhat problematic. Prior to the ANSI standardization, constants were not technically part of the language per se. Instead, a symbolic constant could be declared using the #define macro construct, and the C preprocessor simply performed a macro substitution on all such symbolic constants prior to compilation. ANSI C introduced the const construct for defining a variable whose value would not change. Constants so defined are treated in the same way as variables; they may be declared as extern in an interface module if they are to be shared, and as static within an implementation module if not. In fact, a const is a variable in every way except that its value is not allowed to change. Consequently, without loss of generality a const may be considered to be a variable for our purposes.

Unfortunately, however, a const cannot be used in all situations where a constant value is required; for example, the ANSI language definition does not allow a const to be used as the upper bound of an array index.\(^\text{11}\) Therefore, we must allow for the modelling of symbolic constants also; within the context of our discussion of C, the term "constant" will refer to

\(^{11}\) Some implementations of ANSI C, such as gcc, permit variables to be used as the upper bound of an array index, but this is illegal according to the ANSI standard.
a symbolic constant declared using a `#define`. Also, although symbolic constants are not technically part of the language, practically we can consider that these constants are "normal" resources in C.

It should be noted that constant and type definitions in C are atomic, and cannot be separated out into stubs and bodies like variable and function declarations can. Constants and types may be shared between modules (in which case they are declared within an interface module) or they may be hidden within a single implementation module.

C Modules

We consider that there are three kinds of modules in C: interface modules, implementation modules, and abstract source modules.

- A C interface module (i.e., a file whose name ends in `.h`) consists of a name and a set of resource interfaces (i.e., constants, `typedefs`, function prototypes, and `extern` variables).

- A C implementation module (i.e., a file whose name ends in `.c`) consists of a name and a set of resource implementations (i.e., constants, `typedefs`, function bodies, and variable definitions).

- A C abstract source module consists of an interface module and its corresponding implementation module. The interface module lists the resources provided by the abstract module, and the implementation module gives the implementations of the provided resources (and possibly other resources also). An abstract source module does not correspond to a single file of source code; rather, it is an abstraction for a `.h`/`.c` pair of files. For example, we consider that an interface module `stack.h` and an implementation module `stack.c` together comprise the abstract source module `stack`.

We now define the type `moduleKind`, which we left undefined in the previous section, as consisting of three distinct values.

\[
\text{moduleKind} ::= \text{dot}._\text{h} | \text{dot}._\text{c} | \text{abstract}
\]
We define an auxiliary function rootID that returns the root name of a C program file; that is, it removes the .h or .c extension from the file name, returning the name of the abstract source module. For example, rootID of stack.c is stack.

\[
\begin{align*}
\text{rootID} & : ID \rightarrow ID \\
\forall id : ID \bullet \text{ if } "\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\...
modules may define additional resources that are not provided to clients.

\[
\text{ImplementationModule} \\
\text{Module} \\
\text{rootName} : ID \\
\text{kind} = \text{dot}_c \\
\text{".c" suffix name} \\
\text{rootName} = (\text{rootID name}) \\
\text{implements} = \{\text{rootName} \sim \text{".h"}\} \\
\text{provides} = \emptyset
\]

An abstract source module packages a C interface/implementation module pair into a single logical unit; as such, abstract modules simplify discussion of large, multi-component software systems, although they do not correspond directly to files of source code. The name of the abstract module is the same as the root name of its two submodules; for example, the abstract module `stack` is comprised of the interface/implementation module pair `stack.h` and `stack.c`.

An abstract module is considered to import the abstract modules of any interface modules required by either of its submodules; an abstract module may not implement another module. An abstract module is considered to contain all of the resources defined by its submodules; furthermore, an abstract module provides all of the resources defined in its interface submodule to other abstract modules.

Within an abstract module, two resources may have the same name only if one implements the other (e.g., one is a function prototype and the other is a syntactically compatible function definition). We need not explicitly state how intra-module resource implementations and requirements are resolved; the definitions of `implBindings` and `reqBindings` from the section 5.4.3 are sufficient to specify:

- when resource stubs (i.e., of functions and variables) from the interface module may be implemented by resource implementations in the implementation module, and

- when resource-level requirements of the implementation module may be fulfilled by resource definitions (i.e., of constants and types) from the interface module.

It should be noted that the C language does not require that all resource stubs be implemented. C considers only that the static use of an unimplemented resource is an error.
AbstractModule

C_Module
intf : InterfaceModule
impl : ImplementationModule

kind = abstract
name = (rootID intf.name)
name = (rootID impl.name)
imports = {modID : intf.imports ∪ impl.imports • (rootID modID)}

implments = {r : resDefs • r.(θResource)}
ements = {r : impl.resDefs • r.(θResource)}
p : provides • p.(θResIntf)}
\forall r_1, r_2 : resDefs • r_1.name = r_2.name
\Rightarrow (r_1 = r_2 \lor (r_1, r_2) \in implBindings) \lor (r_2, r_1) \in implBindings

C Subsystems

We consider that C subsystems are comprised of abstract modules. Many of the assumptions about the C interconnection model are incorporated into the definition of Subsystem:

- No two modules may have the same name.

- Resource-level requirements are resolved by matching to resources of the appropriate name. Additionally, the module of the requiring resource must import the module of the providing resource. Note that resource requirements are usually resolved by finding a resource stub of the appropriate name; to determine if a resource stub is implemented, we must examine implBindings.

- We assume that all bindings of resource implementations to resource stubs must occur within abstract modules (although this is only a programming convention, and is not enforced by the language). Therefore, within a subsystem implBindings must be empty. That is, no abstract module may implement a resource whose stub is defined within another abstract module.

- If two distinct resources, say r_1 and r_2, have the same name, then exactly one of the following conditions must be true (assuming r_1 is defined in module M_1 and r_2 is defined in module M_2):
1. $r_1$ and $r_2$ are contained within the same abstract module, and one of them implements the other.

2. Neither of them is provided by their defining abstract modules.

3. One of them, say $r_1$, is provided by its defining module, but the other resource ($r_2$) is not, and module $M_2$ must not transitively "depend on" module $M_1$. That is, $M_2$ may not import $M_1$ or any module that (transitively) does.

We consider that a C library is a special kind of subsystem; it corresponds to a set of abstract modules that have been precompiled into binary form. A C library consists of a name, and a set of abstract modules. We assume that all of the contained abstract modules are provided to clients of the library. We consider that the interfaces of the abstract modules are visible to clients of the library, but that the implementation submodules are opaque. By convention, libraries are "stand alone" entities that implement all of their resource interfaces, and have no module- or resource-level requirements.
Within a system, all requirements must (eventually) be resolved but not all resource stubs need necessarily be implemented. C requires only that a resource stub be implemented if there is a static use of the resource.

\[
\forall r_i : \text{unimplResStubs} \land (\exists M : \text{allModules} \land (\exists q : M.\text{rawReqs} \land r_i.\text{modID} \in (q.\text{imports} \land q.\text{name} = r_i.\text{name}))
\]

5.5.2 Object-Oriented Turing Model

While Object-Oriented Turing is a clean, strongly-typed programming language, it does contain features for system programming that may be considered “dirty” and “dangerous”, such as type cheats, manipulation of machine addresses, and extern procedure definitions. We explicitly ignore these aspects of the language. Also, while OOT modules may “depend” on each other by means of the import, inherit, and implement statements, we make the assumption that no module includes another.

**OOT Resources**

We consider that there are four “normal” kinds of OOT resources: procedures, variables, user-defined types, and constants. We consider that there is an additional, special resource kind called moduleItself that is used to represent the containing module as a sort of resource.

\[\text{We use the term "procedure" to denote both OOT procedures and functions.}\]
CHAPTER 5. A FORMAL MODEL OF CONFORM

This is useful in OOT since the name of a module may be a requirement of a resource in another module, and syntactically there is no way to determine if this requirement corresponds to the name of a module or a resource. Each module is therefore considered to define and export a "special" resource that has the same name as the module which is used to resolve resource dependencies on module names. The existence of this special resource is indicated within the definition of Module below.

\[
\text{resourceKind ::= procedure | variable | type | constant | moduleItself}
\]

Only procedure definitions may be separated into stubs and bodies; all other resource declarations are atomic and fully define the resource. This fact is indicated in the definition of Module below.

OOT Modules

There are two kinds of modules in OOT: classes and modules. An OOT class defines a class of potential abstract data structures, which must be declared and instantiated individually as required. An OOT module declaration defines and creates a single-instance of an abstract data structure. To distinguish between our generic notion of a module (i.e., a container of resources) and the specific idea of an OOT module (i.e., a single-instance abstract data structure), we use the term "OOT-module" when referring to the latter.

\[
\text{moduleKind ::= class | module}
\]

OOT permits but does not insist on the strict separation of modules into interface and implementation files. Instead, we consider that an OOT module simply defines a set of resources and provides a subset of them by listing their names in an export statement.

The implement relation is used to model both inheritance between classes and the implement [by] relation between OOT-modules. OOT does not permit multiple inheritance among classes, nor does it allow OOT-module implementations to be split across multiple OOT-modules.\(^{14}\) Also, it is illegal for a module to both implement and import another module.

\(^{14}\)The Modula3 programming language allows the implementation of a module interface to be split across multiple implementation modules [5].
CHAPTER 5. A FORMAL MODEL OF CONFORM

```
Module

::

::

#implements \leq 1
imports \cap \text{implements} = \emptyset
\forall r : \text{resStubs} \implies r.\text{resourceKind} = \text{procedure}
\exists r : \text{resImpls} \implies r.\text{name} = \text{name} \land r.\text{resKind} = \text{moduleItself} \land r.\text{(\text{UnitResInf})} \in \text{provides}
```

OOT Subsystems

Most of the assumptions about the OOT interconnection model are shown within the definition of Subsystem:

- No two modules may have the same name.
- If one module implements another, then they must both be of the same module kind, i.e., they must both be classes or OOT-modules.
- Circular dependencies among modules are not permitted, i.e., there may be no sequence of modules $M_1, \ldots M_N$ such that each $M_i$ depends on $M_{i+1}$ and $M_N$ depends on $M_1$.
- Resource body $r_i$ implements resource stub $r_s$ iff there exist modules $M_i$ and $M_A$ such that
  - $M_i$ contains resource implementation $r_i$, and
  - both $M_i$ and $M_A$ are either top-level modules in the subsystem, or exported by top-level subsystems (i.e., they can "see" each other), and
  - $M_i$ implements $M_A$, and $M_A$ "transitively implements" the module that defines $r_s$; that is, there is a sequence of modules $M_1 \ldots M_N$ contained within the subsystem such that
    1. each $M_{i+1}$ implements (or inherits from) $M_i$, and
    2. $M_i$ and $M_A$ are the first two elements in the sequence, and
    3. $M_N$ contains resource stub $r_s$. 

99
In an object-oriented context, this models (for example) procedure overriding within an inheritance hierarchy as well as implementing procedures defined as deferred. Note that in OOT a resource need not be exported to be overridable.

- Resource \( r \) satisfies resource requirement \( q \) iff there exist modules \( M_q \) and \( M_A \) such that
  - \( q \) is a requirement of module \( M_q \), and
  - both \( M_q \) and \( M_A \) are either top-level modules in the subsystem, or exported by top-level subsystems (i.e., they can "see" each other), and
  - exactly one of the following conditions is true:
    1. \( r \) is a provided resource of \( M_A \) and \( M_q \) imports \( M_A \).
    2. \( M_q \) "transitively implements" \( M_A \) (see above).
    3. \( M_q \) imports \( M_A \), and \( M_A \) "transitively implements" a module that defines and exports \( r \).
    4. \( M_q \) implements \( M_A \) which "transitively implements" a module that imports the module that defines and exports \( r \).
No additional constraints on CompleteSystem are necessary. The default assumptions that all requirements and module dependencies be resolved is sufficient. Unlike C, OOT considers only the run-time use of an unimplemented resource (i.e., procedure) to be an error; such a use is statically legal in OOT.

5.6 Example Design Constraints

As mentioned in the previous chapter, ConForm supports the addition of design constraints, which are rules of system design that are not enforceable at the programming language level. The ability to define and enforce such constraints is a useful feature of ConForm; it allows different theories of system structuring to be imposed on the construction of new systems as well as verified against existing systems. Being able to add design constraints within a ConForm...
implementation is consistent with ConForm's goal of supporting a generic yet configurable approach to system modelling. However, we will not discuss in detail the relative merits of different design constraints in this dissertation; it is not the focus of our work, and such analyses have been done by others [51, 57].

Before considering particular design constraints, it is useful to re-iterate the default ConForm rules for the exporting of modules and resources by subsystems:

- A subsystem may export any module that it contains as a top-level component.
- A subsystem may export any module that is in turn exported by another subsystem that is a top-level component of the first subsystem.
- If a subsystem exports a module, then it may export any of the resources exported by that module. However, the subsystem need not export any or all of the module’s resources.

We now briefly consider several example design constraints.

1. **Global export rule** — Subsystems must export all of their components and resources.

   ```plaintext
   Subsystem
   
   ...
   
   ...
   
   provesModules = allModules
   provides = \{ r : resources \cdot r.(\emptyUnitResIntf) \}
   ```

   This rule is a weak design constraint; it implies that subsystems are used only for grouping modules, and do not add any visibility restrictions on program source components. Neither modules nor resources may be selectively exported from or hidden within a subsystem; all modules and resources are exported.

2. **Whole export rule** — Subsystems must export all of the (exported) resources of any exported module.
By default, a subsystem $S$ may export any resource that is exported by one of its top-level components, as long as the defining module, say $M$, is also exported by $S$. Thus, $S$ may export only a single resource of $M$, effectively hiding from the clients of $S$ all of the other resources exported by $M$.

The "whole export rule" rule asserts that if $S$ exports a module $M$, then it must also export all of the resources exported by $M$. No finer granularity of exporting is permitted: this rule allows the selective hiding/exporting of modules, but not of individual resources.

3. **One-level export rule** — Modules may be exported only by their enclosing subsystem.

This rule encourages high cohesion within subsystems; it prohibits deeply nested components from being exported "up" multiple levels in a subsystem hierarchy. That is, suppose a module $M$ is contained by a subsystem $S$ which is in turn contained by a subsystem $S'$. The default scoping rule for subsystems allows $S'$ to export $M$ as long as it has also been exported by $S$. The "one-level export rule" allows $S$ to export $M$ but prohibits $S'$ from exporting $M$. This means that all uses of $M$ and resources provided by $M$ must occur within $S$ and $S'$; no other use of $M$ is permitted.
5.7 Versions of Program Components

In this section, we give the formal definition of component versions, version families, and repositories. We also describe various kinds of relations that are possible between versions of the same component.

5.7.1 Component Versions

We begin with the definition of STRING, which is a given type that is meant to model character strings. STRINGSs are used to model user-defined annotations of versions.

\[
\text{[STRING]} 
\]

A Version consists of an name that uniquely identifies each version within a version family (vID), a set of keywords, and a prose description. Additionally, we include the name of the containing version family (vflD) with each version.

\[
\begin{align*}
\text{Version} & : \text{ID} \\
\text{vID} & : \text{ID} \\
\text{keywords} & : \text{F STRING} \\
\text{description} & : \text{STRING}
\end{align*}
\]

Resources, modules, and subsystems may be versioned. A resource version consists simply of a Resource extended with the attributes of a Version.

\[
\begin{align*}
\text{Resource Version} & : \text{F Resource} \\
\text{Resource} & : \text{F Version}
\end{align*}
\]

A UnitVersion is a module or subsystem version. AllVersions models the set of all possible versions of program components; it is used in defining VersionFamily below.

\[
\text{UnitVersion} \equiv \text{Module Version} \cup \text{Subsystem Version} \\
\text{AllVersions} \equiv \text{Resource Version} \cup \text{Unit Version}
\]

A ModuleVersion consists of a Module plus a Version plus a component map (rvMap) of the
CHAPTER 5. A FORMAL MODEL OF CONFORM

contained resources. resVersions corresponds to the set of resources in the underlying Module, augmented with version information. rvMap maps each contained Resource to the appropriate resource version.

\[
\begin{align*}
\text{ModuleVersion} & \quad \text{Module} \\
& \quad \text{Version} \\
& \quad \text{resVersions} : F \text{ Resource Version} \\
& \quad \text{rvMap} : \text{Resource} \leftrightarrow \text{Resource Version} \\
\text{dom rvMap} = \text{resDefs} \\
\text{ran rvMap} = \text{resVersions} \\
\forall r : \text{resDefs} \circ (\text{rvMap} r). (\theta \text{Resource}) = r
\end{align*}
\]

A SubsystemVersion consists of a Subsystem plus a Version plus a component map (cvMap) of the contained units. compVersions corresponds to the set of unit components in the underlying Subsystem, augmented with version information. cvMap maps each contained unit component to the appropriate unit version. Additionally, allModVersions corresponds to the underlying subsystem's allModules, with each module augmented with appropriate version information (and thus by the definition of ModuleVersion, each resource contained in each module is also augmented with appropriate version information).

\[
\begin{align*}
\text{SubsystemVersion} & \quad \text{Subsystem} \\
& \quad \text{Version} \\
& \quad \text{compVersions} : F \text{ Unit Version} \\
& \quad \text{cvMap} : \text{ModuleOrSubsystem} \leftrightarrow \text{Unit Version} \\
& \quad \text{allModVersions} : F \text{ Module Version} \\
\text{dom cvMap} = \text{components} \\
\text{ran cvMap} = \text{compVersions} \\
\forall C : \text{components} \circ (C \in \text{Module} \Rightarrow (\text{cvMap} C).\text{Module} = C) \\
& \quad \land (C \in \text{Subsystem} \Rightarrow (\text{cvMap} C).\text{Subsystem} = C)) \\
\text{allModVersions} = (\text{compVersions} \cap \text{Module Version}) \cup \\
& \quad (\bigcup \{ su : \text{compVersion} \cap \text{SubsystemVersion} \circ s.\text{allModVersions} \}) \\
\{mv : \text{allModVersions} \circ \text{mv}.(\theta \text{Module})\} = \text{allModules}
\end{align*}
\]
5.7.2 Inter-Version Relations

We now consider the various kinds of relationships that may exist between versions of the same component within a version family. We define the relations separately by component kind, but all three kinds of components (i.e., resources, modules, and subsystems) define relations based on the compatibility of component tabs, slots, and bodies, as well as overall compatibility relations. Most of these relations were introduced informally in section 4.1.2; others are described immediately below. We now define these relations precisely, in terms of the formal model of ConForm program components presented earlier in this chapter.

Additional Relations

When two components do not meet a strict definition of compatibility but it is felt that they are related in some way, it is possible to indicate this by a similarity relation. For example, if two module versions are identical except for a single resource signature that differs slightly between the versions, then the resource versions may be explicitly tagged as being rTabSimilar within their resource version family. All similarity relations are decided by the user; they have no formal semantics.

Apart from compatibility relations, all version families maintain a history of member versions as a special relation called evolvedFrom: as each new version is added to a version family, the user must specify an existing family member that the new version is considered to have evolved from. The evolvedFrom relations therefore comprises a tree whose root is the first member of the version family.

Additional inter-version relations, such as those involving user-defined keywords, may be defined in an implementation of ConForm, either a priori or dynamically during execution of a ConForm session.

We now consider the definition of resource, module, and subsystem inter-version relations.

Resource Version Relations

Two resource versions, say $r_{v1}$ and $r_{v2}$, are said to be:

- $r$TabIdentical iff they have the same name and signature.
- rSlotIdentical iff rv1’s set of requirements is the same as rv2’s.
- rSlotCompatible iff rv1’s requirements is a subset of rv2’s, i.e., rv1 has no requirements that rv2 doesn’t also have.
- rCodeIdentical iff they have the same code.
- rIdentical iff they have the same name, signature, and code.
- rPlugIdentical iff are tab identical and slot identical.
- rPlugCompatible iff are tab identical and slot compatible. This means that rv1 may be substituted for rv2 without introducing new requirements that must be fulfilled.

\[
\begin{align*}
&\text{Resource Version Relations} \\
&r\text{TabIdentical} \subseteq \cdots \subseteq r\text{SlotIdentical} \subseteq r\text{SlotCompatible} \subseteq r\text{SlotSimilar} \\
&\quad \subseteq r\text{CodeIdentical} \subseteq r\text{CodeSimilar} \subseteq r\text{Identical} \subseteq r\text{PlugIdentical} \subseteq r\text{PlugCompatible} \\
&\quad \subseteq r\text{SimilarTo} \subseteq r\text{EvolvedFrom} \\
&r\text{AllRelations} = r\text{TabIdentical} \cup \cdots \cup r\text{SimilarTo} \cup r\text{EvolvedFrom} \\
&r\text{TabIdentical} \subseteq r\text{TabSimilar} \\
&r\text{SlotIdentical} \subseteq r\text{SlotCompatible} \subseteq r\text{SlotSimilar} \\
&r\text{CodeIdentical} \subseteq r\text{CodeSimilar} \\
&r\text{Identical} \subseteq r\text{PlugIdentical} \subseteq r\text{PlugCompatible} \subseteq r\text{SimilarTo} \\
\forall rv_1, rv_2 : \text{Resource Version} \bullet \\
&\quad (rv_1 r\text{TabIdentical} rv_2 \iff (rv_1.\text{name} = rv_2.\text{name} \land rv_1.\text{signature} = rv_2.\text{signature}) \\
&\quad \land rv_1 r\text{SlotIdentical} rv_2 \iff rv_1.\text{requires} = rv_2.\text{requires} \\
&\quad \land rv_1 r\text{SlotCompatible} rv_2 \iff rv_1.\text{requires} \subseteq rv_2.\text{requires} \\
&\quad \land rv_1 r\text{CodeIdentical} rv_2 \iff rv_1.\text{code} = rv_2.\text{code} \\
&\quad \land rv_1 r\text{Identical} rv_2 \iff (rv_1 r\text{TabIdentical} rv_2 \land rv_1 r\text{CodeIdentical} rv_2) \\
&\quad \land rv_1 r\text{PlugIdentical} rv_2 \iff (rv_1 r\text{TabIdentical} rv_2 \land rv_1 r\text{SlotIdentical} rv_2) \\
&\quad \land rv_1 r\text{PlugCompatible} rv_2 \iff (rv_1 r\text{TabIdentical} rv_2 \land rv_1 r\text{SlotCompatible} rv_2) \\
\end{align*}
\]

Unit Version Relations

In this section, we define possible relations between versions of units. Both module and subsystem versions may engage in unit version relations. Relations that are specific only to modules or subsystems are clearly indicated as such.

We first define unit version relations that are based on tab similarity. Two unit versions, say uw_1 and uw_2, are said to be:
- \( u_{\text{TabIdentical}} \) iff they provide the same set of resource interfaces, i.e., \( u_{v_1} \) and \( u_{v_2} \) provide the same number of resources and for each resource \( rv_1 \) provided by \( u_{v_1} \), there is a resource \( rv_2 \) provided by \( u_{v_2} \) such that \( rv_1 \) is tab identical to \( rv_2 \).

- \( u_{\text{TabCompatible}} \) iff the set of resource interfaces provided by \( u_{v_1} \) is a superset of that provided by \( u_{v_2} \). That is, \( u_{v_1} \) provides at least the set of resource interfaces provided by \( u_{v_2} \) and may provide additional ones also.

\[
\begin{align*}
\text{Unit Tab Relations} & \quad \text{Resource Version Relations} \\
\text{\( _{u\text{TabIdentical}} \), \( _{u\text{TabCompatible}} \), \( _{u\text{TabSimilar}} \) : Unit Version } & \leftrightarrow \text{ Unit Version}
\end{align*}
\]

\[
\begin{align*}
\forall u_{v_1}, u_{v_2} : \text{Unit Version} \Rightarrow \ ( u_{v_1} & \text{\( _{u\text{TabIdentical}} \) } u_{v_2} \leftrightarrow \\
( \text{\#}(u_{v_1}.\text{provides}) & = \text{\#}(u_{v_2}.\text{provides}) \\
\land (\forall r_{v_1} : u_{v_1}\text{.res Versions} \mid r_{v_1}.(\theta \text{ResIntf}) \in u_{v_1}\text{.provides} \Rightarrow \\
(\exists r_{v_2} : u_{v_2}\text{.res Versions} \mid r_{v_2}.(\theta \text{ResIntf}) \in u_{v_2}\text{.provides} \Rightarrow \\
r_{v_1} \text{\( _{u\text{TabIdentical}} \) } r_{v_2})))) \\
\land u_{v_1} & \text{\( _{u\text{TabCompatible}} \) } u_{v_2} \leftrightarrow \\
(\text{\#}(u_{v_1}.\text{provides}) & \geq \text{\#}(u_{v_2}.\text{provides}) \\
\land (\forall r_{v_1} : u_{v_1}\text{.res Versions} \mid r_{v_1}.(\theta \text{ResIntf}) \in u_{v_1}\text{.provides} \Rightarrow \\
(\exists r_{v_2} : u_{v_2}\text{.res Versions} \mid r_{v_2}.(\theta \text{ResIntf}) \in u_{v_2}\text{.provides} \Rightarrow \\
r_{v_1} \text{\( _{u\text{TabIdentical}} \) } r_{v_2})))) 
\end{align*}
\]

We now define unit version relations that are based on slot compatibility. Two unit versions, say \( u_{v_1} \) and \( u_{v_2} \), are said to be:

- \( u_{\text{ReqIdentical}} \) iff \( u_{v_1} \)’s set of (resource-level) requirements is the same as \( u_{v_2} \)’s.

- \( u_{\text{ReqCompatible}} \) iff \( u_{v_1} \)’s requirements is a subset of \( u_{v_2} \)’s.

- \( u_{\text{ModDepIdentical}} \) iff \( u_{v_1} \)’s set of module-level dependencies is the same as \( u_{v_2} \)’s.

- \( u_{\text{ModDepCompatible}} \) iff \( u_{v_1} \)’s set of module-level dependencies is a subset of \( u_{v_2} \)’s.

- \( u_{\text{SlotIdentical}} \) iff \( u_{v_1} \) is both \( u_{\text{ReqIdentical}} \) and \( u_{\text{ModDepIdentical}} \) with \( u_{v_2} \).

- \( u_{\text{SlotCompatible}} \) iff \( u_{v_1} \) is both \( u_{\text{ReqCompatible}} \) and \( u_{\text{ModDepCompatible}} \) with \( u_{v_2} \).
\[
\begin{align*}
\text{UnitSlotRelations} & \\
\text{uRegIdentical, uReqCompatible, uRegSimilar,} & \\
\text{uModDepIdentical, uModDepCompatible, uModDepSimilar,} & \\
\text{uSlotIdentical, uSlotCompatible, uSlotSimilar : UnitVersion} & \leftrightarrow \text{UnitVersion}
\end{align*}
\]

\[
\begin{align*}
\text{uRegIdentical} & \subseteq \text{uRegCompatible} \subseteq \text{uRegSimilar} \\
\text{uModDepIdentical} & \subseteq \text{uModDepCompatible} \subseteq \text{uModDepSimilar} \\
\text{uSlotIdentical} & \subseteq \text{uSlotCompatible} \subseteq \text{uSlotSimilar}
\end{align*}
\]

\[
\forall w_1, w_2 : \text{UnitVersion} \bullet \\
\quad ( w_1 \text{ uRegIdentical } w_2 \Leftrightarrow \\
\quad \{ r : w_1.\text{requires } r.\text{name} \} = \{ r : w_2.\text{requires } r.\text{name} \} ) \\
\quad \wedge w_1 \text{ uRegCompatible } w_2 \Leftrightarrow \\
\quad \{ r : w_1.\text{requires } r.\text{name} \} \subseteq \{ r : w_2.\text{requires } r.\text{name} \} \\
\quad \wedge w_1 \text{ uModDepIdentical } w_2 \Leftrightarrow \\
\quad ( w_1.\text{imports } = w_2.\text{imports} \wedge w_1.\text{implements } = \text{w}_2.\text{implements} ) \\
\quad \wedge w_1 \text{ uModDepCompatible } w_2 \Leftrightarrow \\
\quad ( w_1.\text{imports } \subseteq w_2.\text{imports} \wedge w_1.\text{implements } \subseteq w_2.\text{implements} ) \\
\quad \wedge w_1 \text{ uSlotIdentical } w_2 \Leftrightarrow \\
\quad ( w_1 \text{ uRegIdentical } w_2 \wedge w_1 \text{ uModDepIdentical } w_2 ) \\
\quad \wedge w_1 \text{ uSlotCompatible } w_2 \Leftrightarrow \\
\quad ( w_1 \text{ uRegCompatible } w_2 \wedge w_1 \text{ uModDepCompatible } w_2 )
\]

Two module versions, say \( m v_1 \) and \( m v_2 \), are said to be \( \text{mCodeIdentical} \) iff they have exactly the same code.

\[
\begin{align*}
\text{ModuleCodeRelations} & \\
\text{mCodeIdentical, mCodeSimilar : ModuleVersion} & \leftrightarrow \text{ModuleVersion}
\end{align*}
\]

\[
\begin{align*}
\text{mCodeIdentical} & \subseteq \text{mCodeSimilar} \\
\forall m v_1, m v_2 : \text{ModuleVersion} \bullet \\
\quad m v_1 \text{ mCodeIdentical } m v_2 \Leftrightarrow m v_1.\text{code } = m v_2.\text{code}
\end{align*}
\]

We now define unit version relations based on the similarity of the contained resources. In the case of two module versions, we consider all of the top-level resources defined within the modules. In the case of two subsystem versions, we consider all of the resources exported by top-level contained units. In both cases, we do not consider which of the resources are exported and which are not. That is, it is possible for two units to be \( \text{uResourceIdentical} \) but have different export lists. We also note that resource continuity asserts nothing about tab or slot compatibility of the resources; we include it as it may provide useful information to someone trying to understand the structural differences between unit versions.
Two unit versions, say \( u_{v1} \) and \( u_{v2} \), are said to be:

- \( u_{\text{ResourceIdentical}} \) iff they contain the exactly the same set of resource versions.
- \( u_{\text{ResourceContinuous}} \) iff for each resource version \( r_{v2} \) contained in \( u_{v2} \), unit \( u_{v1} \) contains a resource version \( r_{v1} \) from the same resource family as \( r_{v2} \).

\[
\begin{align*}
\text{UnitResourceRelations} & \subseteq u_{\text{ResourceIdentical}} \subseteq u_{\text{ResourceContinuous}} \subseteq u_{\text{ResourceSimilar}} \\
\forall u_{v1}, u_{v2} : \text{UnitVersion} \bullet & \ ( u_{v1} \ u_{\text{ResourceIdentical}} u_{v2} \Leftrightarrow \ u_{v1}.\text{resVersions} = u_{v2}.\text{resVersions} ) \\
& \wedge u_{v1} u_{\text{ResourceContinuous}} u_{v2} \Leftrightarrow \ \forall r_{v2} : u_{v2}.\text{resDefs} \bullet ( \exists r_{v1} : u_{v1}.\text{resDefs} \bullet r_{v1}.\text{vfid} = r_{v2}.\text{vfid} )
\end{align*}
\]

We now define structural similarity relations between versions of subsystems. Two subsystem versions, say \( s_{v1} \) and \( s_{v2} \), are said to be:

- \( s_{\text{StructIdentical}} \) iff they contain the same versions of the same (top-level) unit components.
  It should be noted that \( s_{v1} \) and \( s_{v2} \) need not provide the same set of modules and resources.
- \( s_{\text{StructContinuous}} \) iff for each (top-level) component \( u_{v2} \) contained in \( s_{v2} \), \( s_{v1} \) contains a unit \( u_{v1} \) from the same unit version family as \( u_{v2} \).
- \( s_{\text{ModuleIdentical}} \) iff they contain the same versions of the same modules. This is a weaker assertion than \( s_{\text{StructIdentical}} \), as the subsystem structuring can be quite different.
- \( s_{\text{ModuleContinuous}} \) iff for each module \( m_{v2} \) contained at any level within \( s_{v2} \), \( s_{v1} \) contains a module \( m_{v1} \) from the same module version family as \( m_{v2} \).
CHAPTER 5. A FORMAL MODEL OF CONFORM

SubsystemStructureRelations

\[ \_\_sStructIdentical\_\_ , \_sStructContinuous\_\_ , \_sStructSimilar\_\_ , \_sModuleIdentical\_\_ , \_sModuleContinuous\_\_ , \_sModuleSimilar\_ \]: Subsystem Version ↔ Subsystem Version

\[ sStructIdentical \subseteq sStructContinuous \subseteq sStructSimilar \]
\[ sStructIdentical \subseteq sModuleIdentical \subseteq sModuleContinuous \subseteq sModuleSimilar \]
\[ \forall sV_1, sV_2 : \text{Subsystem Version} \Rightarrow \]
\[ ( sV_1 sStructIdentical sV_2 \Leftrightarrow \]
\[ sV_1 . compVersions = sV_2 . compVersions \]
\[ \land sV_1 sStructContinuous sV_2 \Leftrightarrow \]
\[ \forall cV_2 : sV_2 . compVersions \bullet (\exists_1 cV_1 : sV_1 . compVersions \bullet cV_1 . vflD = cV_2 . vflD) \]
\[ \land sV_1 sModuleIdentical sV_2 \Leftrightarrow \]
\[ sV_1 . allModVersions = sV_2 . allModVersions \]
\[ \land sV_1 sModuleContinuous sV_2 \Leftrightarrow \]
\[ \forall mV_2 : sV_2 . allModVersions \Rightarrow \]
\[ (\exists_1 mV_1 : sV_1 . allModVersions \bullet mV_1 . vflD = mV_2 . vflD) \]

We now define overall compatibility relations between unit versions. Two unit versions, say \( uv_1 \) and \( uv_2 \), are said to be:

- \text{uIdentical} if they are tab identical and either code identical (if they are modules) or structurally identical (if they are subsystems). Note that being code/structural identical automatically implies slot compatibility as the requirements and module dependencies are determined by the code/components.

- \text{uPlugIdentical} if they are tab and slot identical. This means that they provide the same components, and that they have identical resource- and module-level requirements; therefore, either unit version may be freely substituted for the other without fear of "breaking the compile", although the semantics are not guaranteed to be the same.

- \text{uStronglyPlugCompatible} if \( uv_1 \) is tab identical and slot compatible with \( uv_2 \). This means that they provide the same components, and that \( uv_1 \) requires no more than \( uv_2 \); therefore, \( uv_1 \) may be freely substituted for \( uv_2 \) without fear of "breaking the compile".

- \text{uWeaklyPlugCompatible} if \( uv_1 \) is strongly tab compatible and slot compatible with \( uv_2 \). This is slightly weaker than strong plug compatibility as \( uv_1 \) may provide additional components not provided by \( uv_2 \); if \( uv_1 \) is to be substituted for \( uv_2 \), then it must be
ascertained that these “extras” do not conflict with the names of other components within
the system [59].

\[
\begin{align*}
\text{UnitCompatibilityRelations} \quad & \quad \text{UnitTabRelations} \\
\text{UnitSlotRelations} \quad & \quad \text{UnitResourceRelations} \\
\text{UnitCodeRelations} \quad & \quad \text{UnitCompatibilityRelations} \\
\text{mEvolvedFrom} \quad & \quad \text{mAllRelations} : \text{Module Version} \leftrightarrow \text{Module Version} \\
\text{mAllRelations} = \text{uTabIdentical} \cup \cdots \cup \text{uSimilarTo} \cup \text{mEvolvedFrom} \\
\text{dom} \; \text{mAllRelations} \cup \text{ran} \; \text{mAllRelations} \subseteq \text{Module Version}
\end{align*}
\]

ModuleVersionRelations and SubsystemVersionRelations combine the appropriate tab, slot,
code, and structural compatibility relations. Additionally, we incorporate the historical or
evolvedFrom relation. Other relations are also possible, especially those based on user-defined
attributes.
A version family (or VF) groups all of the versions of a component into a single container. A version family has a name (typically, it is the same as the name of all of its members) plus a set of component versions. Version families are grouped into repositories: resource VFs are grouped into a resource repository, and the module and subsystem VFs of a software system are grouped into a system repository (see below). parentID is the name of the containing or parent repository.

A resource version family is a VF whose elements are all resource versions. Resource version relations are modelled. Additionally, the evolvedFrom relation defines a tree with rootRV as its root; we omit the specification of this detail.
![Diagram](image.png)

A *module version family* is a VF whose elements are all module versions. Module version relations are modelled. A module VF also contains a resource repository (see below) that contains all of the versions of all of the resources contained in all of the module versions organized into resource version families. We assume that all versions of the module are of the same module kind. Additionally, the mEvolvedFrom relation defines a tree with rootMV as its root.

![Diagram](image.png)

A *subsystem version family* is a VF whose elements are all subsystem versions. Subsystem version relations are modelled. Additionally, the sEvolvedFrom relation defines a tree with rootSV as its root.
CHAPTER 5. A FORMAL MODEL OF CONFORM

\[ \text{Unit Version Family} \equiv \text{Module Version Family} \cup \text{Subsystem Version Family} \]
\[ \text{All Version Families} \equiv \text{Resource Version Family} \cup \text{Unit Version Family} \]

5.7.4 Repositories

A repository contains version families: a resource repository contains resource VFs and a system repository contains module and subsystem VFs. A repository has a name: typically, the name of a resource repository is the same as the name of the module that contains it, and the name of a system repository corresponds to the name of the particular software system whose components the unit VFs contain.

\[
\text{Repository} \\
\text{name, parentID : ID} \\
\text{VFs : } \mathcal{F} \text{ All Version Families} \\
\text{vfIDs : } \mathcal{F} \text{ ID} \\
\text{idVF : ID } \rightarrow \text{ All Version Families} \\
\forall \text{vf : VFs } \bullet \text{vf.parentID = name} \\
\forall \text{vfID : vfIDs } \bullet \text{(idVF vfID).name = vfID} \\
\text{dom idVF = vfIDs} \\
\text{ran idVF = VFs} \\
\forall \text{vf}_1, \text{vf}_2 : \text{VFs } \bullet \text{vf}_1.\text{name = vf}_2.\text{name } \Rightarrow \text{vf}_1 = \text{vf}_2
\]

A resource repository contains all of the resource version families of a particular module VF. We consider that the resource repository is part of and contained within the module VF. The resource repository attribute parentID contains the name of the containing module VF.

\[
\text{ResourceRepository} \\
\text{Repository} \\
\text{VFs } \subseteq \text{Resource Version Family}
\]

A system repository contains all of the module and subsystem VFs of a particular software system. The component versions of different software systems are stored in different system repositories. System repositories are grouped into a software repository (see below); parentID is the name of the containing software repository.

115
VFs $\subseteq (\text{Module VersionFamily} \cup \text{Subsystem VersionFamily})$

$\forall \text{suf} : \text{VFs} \cap \text{Subsystem VersionFamily} \bullet (\forall \text{sv} : \text{suf}. \text{versions} \bullet$

$(\forall \text{cv} : \text{sv}. \text{comp Versions} \bullet (\text{cv}. \text{vfID} \in \text{vfIDs} \land \text{cv} \in (\text{idVF} \text{ cv}. \text{vfID}))))$

A collection of system repositories comprises a *software repository*.

5.8 CalSyCon — A Calculus of System Construction

We now consider the specification of a formal calculus for system construction, which we call CalSyCon. This calculus specifies the operations that may be used to construct new subsystems from a repository of program components; these operations also define the basic functionality required for a visual system configuration tool.

We have formally specified CalSyCon as a set of Z "operations" (i.e., procedures) on the schemas (i.e., data types) that we defined earlier in this chapter. However, before proceeding with the definition, we first explain some of the conventions used in modelling operations in Z.

5.8.1 Modelling Operations in Z

A procedure or "operation" (to use the standard Z terminology) is specified as a schema, similar to the way in which data types are specified. Each operation schema has a name, a data part, and an assertions part. The data part describes the assumed state and parameters of the operation. The assertions describe the semantics of the operation in terms of its effects on the
entities listed in the data part. By assumption, an operation may "affect" only those data items listed in the data part.

By convention, data components whose names end with a question mark denote input or "read-only" parameters of the operation. Similarly, components whose names end with an exclamation mark indicate output, or "write-only" parameters of the operation. For example, in the operation schema `CreateAndAddSystemRepos` below, `name?` is an input parameter and `sysRepos!` is an output parameter. "Read-write" parameters are indicated by the use of a single quote, or "prime": an unprimed reference in an assertion denotes the pre-operation value of the data item, and a primed reference denotes the post-operation value. For example, in the operation schema `CreateAndAddSystemRepos` below, `swr` and `swr'` are meant to denote the pre- and post-operation values of the same entity. It should be noted that these naming rules are only conventions; they are not "built in" to the definition of the Z language in any way. Z does not actually model time or changes of state per se; rather, it considers, for example, primed and unprimed references to denote distinct entities whose values happen to share interesting and useful properties.

Most operations involve input and output parameters, as well as primed and unprimed data items. However, some operations — especially those that involve initializing a structure of some kind — contain primed data items but not the corresponding unprimed ones. `CreateSoftwareRepos` is an example of such an operation. In this case, we consider simply that the initial or unprimed values of such data items are uninteresting.

Because primed and unprimed data items usually occur in pairs, there is a special shorthand notation, $\Delta S$, for including a primed and an unprimed schema called $S$. Typically, this is used to indicate situations where parts of a schema $S$ may be changed by an operation. No assumptions about the relation between the primed and unprimed components are assured, beyond the fact that they must both satisfy $S$'s schema assertions. In particular, it is often the case that an operation will "change" a small number of the components of the named schema, leaving the rest (possibly a large set of components) unchanged. However, it is not enough to simply constrain the values of the changed components; one must also state formally "... and nothing else changed." Typically, this involves explicitly stating for each unchanged component $c$ of $S$ that $c' = c$. If the schema contains many unchanged parts, this can make
the specification cumbersome and difficult to read. This situation, commonly known as the frame problem [3, 64], has no clean, general-purpose solution in standard Z. We have adopted an informal notational solution: if a $\Delta S$ occurs in the data part of an operation schema, then a diagonal ellipsis (i.e., `.`) at the end of the schema indicates that any parts of the schema $S$ that are not mentioned within the assertions part will not be changed.\footnote{It should be noted that this is not a standard Z convention.}

5.8.2 Creating and Adding Repositories and Version Families

The first operation in our calculus concerns the creation of a software repository. A (non-$\texttt{NULL}$) name is provided, and a new software repository is created of that name with an initially empty set of system repositories.

```
CreateSoftwareRepo
sur' : SoftwareRepository
name? : ID

name? \neq \texttt{NULL}
sur'.name = name?
sur'.sysRepositories = $\emptyset$
```

Given an existing software repository and a name, a new system repository of that name is created and added to the software repository. The system repository is initialized to be empty. The name of the new system repository must not be $\texttt{NULL}$ and must not be the name of another system repository already within the software repository.

```
CreateAndAddSystemRepo
sur, sur' : SoftwareRepository
sysRepos! : SystemRepository
name? : ID

name? \neq \texttt{NULL}
name? \notin sur.sysReposIDs
sur'.sysRepositories = sur.sysRepositories $\cup$ \{sysRepos!\}
sysRepos!.name = name?
sysRepos!.VFs = $\emptyset$
sysRepos!.parentID = sur.name
sur'.name = sur.name
```
Given an existing resource repository and a resource version, a new resource version family is created for the resource version and the resource VF is added to the resource repository. (Note that a resource repository is an explicit sub-part of each module version family; consequently, resource repositories are created implicitly when a module VF is created.) The name of the new resource VF will be indicated by the vflD field of the version; typically, this will also be the name of the component (i.e., the name field), although this is not a strict requirement. The final two assertions state that “nothing else changed”.

The first version added to a resource, module, or subsystem version family is considered to be the root version. Each subsequent version of this component will be considered to have evolved from a version already in the VF; consequently, the evolvedFrom relation forms a tree whose root is this root version.

```plaintext
CreateAndAddRVF
rr, rr' : ResourceRepository
rvf! : Resource Version Family
rv? : Resource Version

rv?.name ≠ NULL
rv?.vflD ∉ rr..vflDs
rr'.VFs = rr.VFs ∪ {rvf!}
rvf!.name = rv?.vflD
rvf!.versions = {rv?}
rvf!.rootRV = rv?
rvf!.rAllRelations = ø
rr'.name = rr.name
rr'.parentID = rr.parentID
```

Given a system repository and a unit (i.e., module or subsystem) version, an appropriate unit version family is created and added to the system repository. As above, a unit VF of the indicated name must not already exist, and the unit version is added as the root version. If the unit is a module, then a new resource repository (mrvf!.rr) is implicitly created and explicitly initialized.
5.8.3 Checking in New Versions

Given a resource repository and a resource version, the resource version is checked into the correct resource VF within the resource repository.

If the resource VF does not yet exist, then it is created and added to the resource repository; this is specified by a reference to the previous schema CreateAndAddRVF. The formal meaning of referencing another schema within the assertions part of a schema is to unfold and merge the data and assertions of the referenced schema into the current context. Note that this occurs only if the “guard” condition \( v?.vfID \in sr.ufIDs \) is met; otherwise, the data and assertions of the referenced schema are not incorporated into the current schema.

If there is an appropriately-named resource VF within the resource repository, then (assuming there is not already a version of that vID in the version family) the version is simply added to the VF. The evolvedFrom relation is updated. Within the resource VF, additional assertions ensure that the various similarity relations do not change, except possibly for the addition of instances involving the new version. This is necessary because, unlike the other relations within a VF, similarity relations and have no formal semantics; two versions are “similar” simply if the user says so.
CHAPTER 5. A FORMAL MODEL OF CONFORM

---

CheckInResourceVersion

∀rr,rr' : ResourceRepository
rur : ResourceVersion

∀rr?.name ≠ NULL
∀rr?.vID ≠ NULL

∀rr?.vID ∉ rr.vfIDs ⇒ (∃rf ! : rr'.VF s • CreateAndAddRVF)

∀rr?.vID ∈ rr.vfIDs ⇒ (∃rf : rr'.VF s; rf' : rr'.VF s • rf?.name = rr?.vID
∧ (∃rur1 : rur.versions • rur1.vID = rr?.vID ∧ rf' = rf)
∨ (rr?.vID ∉ rf.vIDs
∧ rr?.name ∉ rf.vIDs
∧ rf'.versions = rur.versions ∪ {rr?})
∧ ∃rur1 : rur.versions • rf'.rEvolvedFrom = rur.rEvolvedFrom ∪ {(rr, rr1)}
∧ rf'.rTabSimilar ∩ (rur.versions × rf.versions) = rf'.rTabSimilar
∧ rf'.rSlotSimilar ∩ (rur.versions × rf.versions) = rf'.rSlotSimilar
∧ rf'.rCodeSimilar ∩ (rur.versions × rf.versions) = rf'.rCodeSimilar
∧ rf'.rSimilarTo ∩ (rur.versions × rf.versions) = rf'.rSimilarTo
∧ rf'.parentID = rur.parentID
∧ rf'.rootRV = rur.rootRV
∧ rf'.name = rur.name))

---

Given a system repository and a module version, the module version is checked into the correct module VF within the system repository. The details are analogous to CheckInResourceVersion, with the additional requirement that all of the module's contained resource versions be checked into the module VF's resource repository.
Given a system repository and a subsystem version, the subsystem version is checked into the correct subsystem VF within the system repository. The details are analogous to `CheckInModuleVersion`, except that the subsystem's contained unit versions are also checked into the appropriate unit VFs within the system repository.
5.8.4 Constructing a New Subsystem

Having described the steps necessary to create repositories and version families, we now consider how to configure new subsystems from existing components. Note that we are not concerned with the creation of modules and resources, as we consider this to be a programmer’s activity. We do, however, provide operations for creating resource/module versions from existing resources/modules, as well as for setting the version identifier and version family name. CreateNewRV and CreateNewMV respectively create resource and module versions from existing resources and modules. The version identifiers and version family identifiers are initially NULL; these may be changed by subsequent operations given below. Note that the second constraint in CreateNewMV is actually redundant: the definition of ModuleVersion ensures that appropriate resource versions exist for each resource in the underlying Module. This assertion is listed for the sake of clarity.
CHAPTER 5. A FORMAL MODEL OF CONFORM

CreateNewSV creates a blank subsystem version. The name, version identifier, and version family identifier are initially set to NULL; they must be reset before the subsystem version can be added to the repository.

CloneNewSV creates a new subsystem version that is identical to another subsystem, except that the name and version identifier are set to NULL. The original or “master” subsystem version may be taken from a repository or may be a partially-constructed subsystem itself.
CHAPTER 5. A FORMAL MODEL OF CONFORM

The name of a new subsystem is initially NULL, which is not a legal name for a subsystem within a repository (a pre-condition to CheckInSubsystemVersion is that the name not be NULL). ChangeSVname allows the name of a subsystem under construction to be changed. A subsystem name may be changed any number of times while the subsystem is under construction. Resource and module names may not be changed, as this affects the underlying program and is thus a programming activity.

No other attribute of the subsystem is altered by ChangeSVname; this is the first schema to use the diagonal ellipsis to indicate that "nothing else changed".

\[
\begin{align*}
\text{ChangeSVname} & \\
\Delta \text{SubsystemVersion} & \\
\text{name}?: \text{ID} & \\
\text{name}?: \neq \text{NULL} & \\
\text{name}' = \text{name} & \\
\text{vID}' = \text{vID} & \\
\text{vfID}' = \text{vfID} & \\
\text{implements}' = \text{implements} & \\
\cdots & \quad \text{"\ldots and nothing else changed."}
\end{align*}
\]

The following six operations allow for resource, module, and subsystem versions to change their version identifier and version family. Apart from being non-NULL, the choice of a new name is not constrained. However, version identifiers must be unique within a version family; this constraint is part of CheckIn...Version.

\[
\begin{align*}
\text{ChangeRVvID} & \\
\Delta \text{ResourceVersion} & \\
\text{vID}?: \text{ID} & \\
\text{vID}?: \neq \text{NULL} & \\
\text{vID}' = \text{vID} & \\
\text{vfID}' = \text{vfID} & \\
\text{name}' = \text{name} & \\
\text{vID}' = \text{vID} & \\
\text{vfID}' = \text{vfID} & \\
\text{implements}' = \text{implements} & \\
\cdots
\end{align*}
\]

\[
\begin{align*}
\text{ChangeRVvfID} & \\
\Delta \text{ResourceVersion} & \\
\text{vfID}?: \text{ID} & \\
\text{vfID}?: \neq \text{NULL} & \\
\text{vfID}' = \text{vfID} & \\
\text{name}' = \text{name} & \\
\text{vID}' = \text{vID} & \\
\text{vfID}' = \text{vfID} & \\
\text{implements}' = \text{implements} & \\
\cdots
\end{align*}
\]
Descriptive keywords may be added to or removed from a component version. We omit the specification of analogous operations to add, delete, and change component version descriptions.
A new subsystem version created by `CreateNewSV` initially has no components. The operation `AddModuleVersionComponent` allows a module version to be added as a top-level component of the subsystem. The name of the module must be distinct from all other modules and top-level subsystems. The module version need not belong to a module VF within the system repository; when the new subsystem is completed and is checked in to the system repository, all of its components will also be added to appropriate unit version families implicitly.

Module versions may be added to or deleted from the top-level of a subsystem under construction. `DeleteModuleVersionComponent` deletes a top-level module version from the subsys-
tem. The module's provided resources are deleted from the subsystem's set of resources and provided resource interfaces. If the module had been exported by the subsystem, then it is removed from the module export list. Some of these constraints are redundant, as they are implicitly true by the definition of SubsystemVersion; these assertions are listed here for the sake of clarity.

\[\text{DeleteModuleVersionComponent}\]
\[
\Delta \text{SubsystemVersion} \\
mv? : \text{Module Version} \\

mv? \in \text{compVersions} \\
\text{compVersions}' = \text{compVersions} \setminus \{mv?\} \\
\text{resources}' = \text{resources} \setminus \{r : \text{mv?\.resources} | r.(\theta \text{UnitResIntf}) \in \text{mv?\.provides}\} \\
\text{provides}' = \text{provides} \setminus \text{mv?\.provides} \\
\text{provModules}' = \text{provModules} \setminus \{mv?\} \\
\theta \text{Version}' = \theta \text{Version}
\]

AddSubsystemVersionComponent allows a subsystem version to be added as a top-level component of the subsystem. The name of the subsystem must be distinct from all modules and top-level subsystems.

\[\text{AddSubsystemVersionComponent}\]
\[
\Delta \text{SubsystemVersion} \\
sv? : \text{Subsystem Version} \\

\exists cv : \text{compVersions} \cup \text{allModVersions} \bullet cv\text{.name} = sv?\text{.name} \\
\text{compVersions}' = \text{compVersions} \cup \{sv?\} \\
\theta \text{Version}' = \theta \text{Version} \\
\text{provides}' = \text{provides} \\
\text{provModules}' = \text{provModules}
\]

DeleteSubsystemVersionComponent deletes a top-level subsystem version from the subsystem under construction. The provided resources of the deleted subsystem are removed from the (parent) subsystem's set of resources and provided resource interfaces. If any of the deleted subsystem's modules had been provided by the parent subsystem, then these module names are removed from the list of provided modules. Some of these constraints are redundant, as they are implicitly true by the definition of SubsystemVersion; these assertions are listed here for the
sake of clarity.

\[\text{DeleteSubsystemVersionComponent}\]
\[\Delta \text{Subsystem Version}\]
\(sv? : \text{Subsystem Version}\)

\(sv? \in \text{comp Versions}\)
\(\text{comp Versions}' = \text{comp Versions} \setminus \{mv?\}\)
\(\text{provides}' = \text{provides} \setminus \text{sv?}.\text{provides}\)
\(\text{provModules}' = \text{provModules} \setminus \text{sv?.provModules}\)
\(\theta \text{Version}' = \theta \text{Version}\)

A top-level module or a module provided by a top-level subsystem may be provided or hidden by the subsystem under construction. If the module is hidden, then none of its resources may be provided by the subsystem.

\[\text{ProvideModule}\]
\[\Delta \text{Subsystem Version}\]
\(m? : \text{Module}\)

\((m? \in \text{components} \lor (\exists ss : \text{components} \cap \text{Subsystem} \bullet m? \in ss.\text{provModules}))\)
\(m? \not\in \text{provModules}\)
\(\text{provModules}' = \text{provModules} \cup \{m?\}\)
\(\theta \text{Version}' = \theta \text{Version}\)
\(\text{comp Versions}' = \text{comp Versions}\)
\(\text{provides}' = \text{provides}\)

\[\text{HideModule}\]
\[\Delta \text{Subsystem Version}\]
\(m? : \text{Module}\)

\(m? \in \text{provModules}\)
\(\text{provModules}' = \text{provModules} \setminus \{m?\}\)
\(\text{provides}' = \text{provides} \setminus \text{mv?}.\text{provides}\)
\(\theta \text{Version}' = \theta \text{Version}\)
\(\text{comp Versions}' = \text{comp Versions}\)
\(\text{provides}' = \text{provides}\)

A resource that is contained and exported by a top-level component may be provided by the subsystem. The module in which the resource is defined must also be provided by the subsystem;
this means that the containing module must be a top-level component, or be exported by a top-level subsystem.

\[
\text{ProvideResource}
\]
\[
\Delta \text{Subsystem Version}
\]
\[
p? : \text{UnitResIntf}
\]
\[
\exists c : \text{components} \bullet p? \in c.provides
\]
\[
p? \notin \text{provides}
\]
\[
\text{provides'} = \text{provides} \cup \{p?\}
\]
\[
\exists m : \text{allModules} \bullet (p? \in m.provides)
\]
\[
\land (\ (m \in \text{proModules} \land \text{proModules'} = \text{proModules})
\]
\]\
\[
\lor (m \notin \text{proModules} \land \text{proModules'} = \text{proModules} \cup \{m\} )
\]
\]
\[
\theta \text{Version'} = \theta \text{Version}
\]
\[
\text{compVersions'} = \text{compVersions}
\]

A resource that is exported by the subsystem may be de-exported or hidden.

\[
\text{HideResource}
\]
\[
\Delta \text{Subsystem Version}
\]
\[
p? : \text{UnitResIntf}
\]
\[
p? \in \text{provides}
\]
\[
\text{provides'} = \text{provides} \setminus \{p?\}
\]
\[
\theta \text{Version'} = \theta \text{Version}
\]
\[
\text{compVersions'} = \text{compVersions}
\]
\[
\text{provides'} = \text{provides}
\]
\[
\text{proModules'} = \text{proModules}
\]

Sometime it is useful to consider a "flattened" view of a subsystem. FlattenSelf takes a subsystem and removes all of the internal subsystems, but retains all of the modules. The result is a subsystem consisting only of modules. The set of provided resources and modules remain the same, so the flattening has no observable effects to clients of the subsystem.

\[
\text{FlattenSelf}
\]
\[
\Delta \text{Subsystem Version}
\]
\[
\text{compVersions'} = \text{allModVersions}
\]
\[
\theta \text{Version'} = \theta \text{Version}
\]
\[
\text{provides'} = \text{provides}
\]
\[
\text{proModules'} = \text{proModules}
\]
FlattenSubsystemComponent is similar to FlattenSelf, except that it is a top-level component subsystem that is flattened, rather than the parent subsystem. The component subsystem is removed from the parent subsystem, but all of its (transitively) contained modules are added to the parent subsystem as top-level components.

\[
\begin{align*}
\text{FlattenSubsystemComponent} \\
\Delta \text{Subsystem Version} \\
sv? : \text{Subsystem Version} \\
sv? \in \text{comp Versions} \\
\text{comp Versions'} = \text{comp Versions} \cup sv?.allModVersions \setminus \{sv?\} \\
\theta \text{ Version'} = \theta \text{ Version} \\
\text{provides'} = \text{provides} \\
\text{prouModules'} = \text{prouModules}
\end{align*}
\]

5.9 Summary

In this chapter, we presented the formal definition of the ConForm notation for system modelling. We formally specified the ideas of ConForm program components (resources, modules, and subsystem), as well as program component versions. We defined a large set of possible relations between versions of the same component. We formally specified the additional constraints required of the basic model to model programs written in the C and Object-Oriented Turing programming languages, and we defined several possible system design constraints. Finally, we presented a calculus of system construction that describes the basic operations that an implementation might have.

In the next chapter, we discuss various implementation issues. We describe a prototype ConForm implementation called Proto-ConForm, and we discuss validation of the ConForm approach.
Chapter 6

Implementation and Evaluation

In previous chapters, we introduced the ConForm framework for visual system configuration. In Chapter 3, we motivated the need for such a framework, and discussed the ConForm notation for versioned system modelling. In Chapter 4, we discussed other aspects of ConForm, such as modelling relations between component versions and the addition of constraints to the basic ConForm data model. In Chapter 5, we presented a formal definition of ConForm.

In this chapter, we discuss implementation and evaluation considerations. First, we describe a simple process by which a software system may be stored in and manipulated by a ConForm implementation. Then, we describe how the ConForm framework for visual system configuration may be specialized, both abstractly and concretely, and we consider the requirements of an implementation of ConForm. Next, we discuss a prototype implementation of ConForm, called Proto-ConForm. Finally, we briefly describe the results of some informal interviews that were held with several researchers and industrial practitioners to discuss the practicality of ConForm.

6.1 Implementation Considerations

6.1.1 A Typical Process

In Chapter 3, we introduced ConForm's system modelling notation, and in Chapter 5 we formally defined it. We now present a simple process that describes how an existing software system might be stored in a ConForm implementation. We assume that the system source code is complete and consistent, and that there are multiple versions of the system and its
CHAPTER 6. IMPLEMENTATION AND EVALUATION

components that we wish to include in our repository. We also assume that the ConForm implementation includes a parser for the programming language of the example system.

The first step is to group the system's modules into subsystems so that the entire system may be viewed as a tree-like hierarchy of subsystem “snapshots”. This subsystem structuring must be performed for each current or historically significant version of the full system. Note, however, that the subsystem decomposition need not be identical for every version of the system. We do not explicitly address “goodness” criteria for subsystem structuring, such as coupling and cohesion, as these topics have been well studied by others [74, 82, 57].

The second step is to group all of the versions of each unit (i.e., module or subsystem) into version families. That is, create a version family for each module and subsystem that appears in at least one version of the full system. These unit version families comprise the system repository for this software system. This step can be carried out automatically by a ConForm implementation in conjunction with the next step.

The third step is to use program understanding techniques to extract structural, syntactic, and resource flow information from each version of each of the source modules. Once extracted, this information can then be added to the description of the individual module versions within the appropriate module version families.

Having extracted this information about each module, it is then possible to determine structural, syntactic, and resource flow information about each subsystem version (based on that of its subcomponents). Explicit subsystem interfaces can be created, and both the structure and component interconnection information can be added to the description of the subsystem versions within the appropriate subsystem version families.

The final step is to use the information extracted above to define various relationships between versions of the same component, such as flavours of compatibility and similarity. This can be done automatically for all three kinds of components: resource, module, and subsystem. Other inter-version relations may also be added as desired.

Once these steps have been completed and the system repository has been set up, it is possible to add new versions of components with little effort. As a new component version is added, it is passed through the ConForm program understanding tool to extract its structural, syntactic, and resource flow information. New versions of modules can be checked into the
repository individually, or whole subsystems can be added at once; in either case, the sub-
components are also processed and added into the appropriate version family. Additionally,
new version families are created automatically as needed for entirely new components.

6.1.2 Specializing the ConForm Framework

ConForm is a framework for visual system configuration. It supports a generic view of system
modelling that is independent of programming language, and it does not insist on a particular
design methodology or subsystem structuring scheme. However, this framework may be special-
ized to model the interconnection assumptions of individual programming languages such as C,
as well as user-defined design constraints which specify particular rules of system structuring.
This specialization of ConForm may be both abstract and concrete:

- Abstract specialization is accomplished by adding constraints to the formal model. For
  example, sets of rules to model the C and OOT programming languages were presented
  in sections 4.2.1 and 5.5, and design constraints were considered in sections 4.2.2 and 5.6.

- Concrete specialization may be accomplished by the tailoring of a ConForm implementa-
tion to check that design constraints and programming-language-specific constraints are
  obeyed.

We note that concrete specialization is not simply a matter of creating and integrating
a specialized parser that understands the particular programming language and subsystem
description language. Any parser that comprises part of a ConForm implementation must
be capable of extracting the facts about program components that are modelled by the basic
ConForm system modelling notation. Concrete specialization means that the ConForm system
must enforce additional constraints not found in the basic model, such as design constraints
and programming language constraints. That is, concrete specialization implements abstract
specialization.

We now discuss the tools that comprise an implementation of ConForm.

1 Although legal, we consider it unlikely that users will wish to add new versions of resources by hand. It is
more likely that new resource versions will be added implicitly within their defining modules.
6.1.3 Constituent Tools

An implementation of ConForm consists of several tools (Fig. 6.1):

1. a specialized parser,

2. a mechanism to implement a repository of program components,

3. a graphical browser for the repository, and

4. a graphical implementation of the calculus of system construction defined in section 5.8.

The parser in a ConForm implementation must be capable of understanding programming language modules as well as subsystems written in a subsystem description language such as SIL [51]. It must be able to extract the properties of program components (i.e., subsystems, modules, and resources) that are modelled by the ConForm system modelling notation. For example, such a tool must be able to parse the source code of a system to determine which resources are provided by each module and subsystem, and what the resource- and module-level requirements of each component are. The results of parsing the system are then passed to the repository mechanism, which stores the information about the various program components. If additional programming-language-specific or design constraints are to be enforced, then the parser must be tailored to implement them; as yet, we have not addressed the problem of seamlessly integrating such constraints at run-time.

The repository mechanism must be able to manage the various levels of program component version families and sub-repositories as described in section 3.4.2 and illustrated in Fig. 3.10. In addition to storing the component versions, the repository mechanism must be able to support the definition and querying of relations between component versions.

The graphical repository browser must support the convenient browsing of the various levels of the ConForm repository, including the visual querying of inter-version relations and the modelling of differences between component versions. It should also include appropriate visual elision mechanisms and support multiple views of component VFs and repositories.

The graphical implementation of CalSyCon must support the creation of new components from existing ones within the ConForm repository. It should be integrated within the overall
Figure 6.1: The architecture of a ConForm implementation. The blue arrows indicate run-time dependencies between the major components. The purple arrows indicate when a subsystem is exported by its parent subsystem. The black arrows denote system configuration-time interactions: black arrows are not part of the ConForm notation and are added as annotations to the diagram.
implementation so that it is possible to phrase queries of the repository from within the CalSy-Con tool, and so that it is possible to invoke the parser to verify the well-formedness of newly constructed systems.

The system architecture of a ConForm implementation is shown in Fig. 6.1.\textsuperscript{2} We have drawn the system architecture using a ConForm-style approach: the major components are represented as subsystems, their run-time dependencies then are represented as (blue) resolution arrows, and (purple) propagation arrows indicate when subsystems are exported by their containing subsystem for use by clients. Additionally, we have added black arrows to indicate possible configuration-time interactions: the system administrator may define programming language constraints or design constraints that are implemented within the parser. We note that these black arrows are not part of the basic ConForm visual notation.

In the next section, we discuss how our prototype implementation of ConForm, called Proto-ConForm, satisfies each of these requirements.

6.1.4 Proto-ConForm

A prototype implementation of ConForm, called Proto-ConForm, has been developed. Proto-ConForm includes a special-purpose parser that supports the Object-Oriented Turing programming language. The ConForm repository has been implemented by using the Unix file system to structure the various levels of repositories and version families, and by using text files to store the program components and their various attributes. The parser and repository mechanism are implemented in Object-Oriented Turing (OOT) using about 7000 lines of source code. The visual browsing and system configuration facilities have been implemented separately in OOT using the OOT graphical user interface library GUILT [55]; this comprises about 8000 lines of source code, not counting the GUILT source.

OOT was chosen as the implementation language for several reasons: first, OOT is a clean, strongly-typed language with good support for high-level abstractions; second, it is the language with which the author is most familiar; and third, because of the existence of a set of OOT abstract data structures, called Abstur [29], which allowed the conceptual distance from

\textsuperscript{2}The interactions between the four basic components define an example of a repository architectural style [81, 82].
CHAPTER 6. IMPLEMENTATION AND EVALUATION

specification to implementation to be kept relatively small [64].

Figure 6.2 shows an example use of the Proto-ConForm design.³ The window at the top left shows the System Repository Browser for MiniTunis. This particular view of the system repository shows a collection of four module VFs (i.e., Tty, DeviceDriver, Disk, and DiskMutex) and one subsystem VF (i.e., DEVICE_SYS).

This view shows the module VFs within the subsystem VF. Although in section 3.4.2 we indicated that all of the module and subsystem VFs of a system are considered to exist in a flat namespace at the "top-level" of the system repository, we can also create and store different structural snapshot views of the system to be used as navigational aids.

The Disk module VF has been selected within the system repository browser; its contents are displayed below in the Version Family Browser. The VF browser shows all of the different versions of the Disk module. Additionally, inter-version relations can be indicated; here, a blue arrow drawn from one version to another indicates that the first is strongly plug compatible with the second.

The right-most window is the Configuration Build Window. Here, a new version of the DEVICE_SYS subsystem is being constructed from both new and old components. Version 3.0 of DEVICE_SYS consists of new versions of the Tty and DeviceDriver modules; these module versions have been parsed by ConForm, but have not yet been added to their respective module VFs. DeviceDriver still requires a version of the Disk module, as indicated by its highlighted module-level slot. Since the ConForm system understands⁴ both the versions of Disk within the repository as well as the new versions of Tty and DeviceDriver in the configuration build window, ConForm can provide automated help in reasoning about possible compatible components from within the system repository. That is, the ConForm system can aid the user in "browsing the version space" to determine if any existing versions of Disk are compatible with the new versions of Tty and DeviceDriver, and if not, what changes must be made.

The ConForm system is capable of verifying that a newly constructed system is well formed, i.e., that it satisfies the basic ConForm rules for system construction given in earlier chapters, the programming-language-specific rules for the particular implementation language, and that it

³Not all of this design has been implemented.
⁴ConForm understands these components because it has parsed them.
Figure 6.2: An example Proto-ConForm session. The top left window shows the system repository browser, which is focused on the DEVICE_SYS subsystem version family (VF). Within this subsystem VF, we can see four module VFs; the Disk module VF has been selected, and its contents are shown immediately below in the version family browser window. This window displays all of the different versions of Disk, and can also indicate various kinds of inter-version relations. On the right is the configuration build window, which shows a new version of DEVICE_SYS under construction. This new version is not yet complete because its DeviceDriver subcomponent has an unfulfilled module-level requirement, as indicated by the highlighted slot labelled Disk.
satisfies any design constraints that have been imposed. Once a new system has been verified to be well formed, it can be checked into the system repository: the new version of the DEVICE_SYS subsystem is checked into the DEVICE_SYS subsystem VF, and the new subcomponents are also automatically added to the appropriate module VFs.

Prototypes of all four of the constituent tools of ConForm have been implemented within Proto-ConForm. However, we have focused our investigations on experimenting with the practicality of various ideas and design strategies, rather than on creating a production-quality tool. Consequently, the current implementation of Proto-ConForm does not support all of the functionality described above; for example, the graphical interface implemented by Proto-ConForm supports an older design of the ConForm system modelling notation than that shown in Fig. 6.2, and the Proto-ConForm repository does not yet support the querying of inter-version relations.

6.1.5 Summary of Implementation Considerations

In this section, we have discussed various implementation considerations of ConForm. We described a typical process for using ConForm, we discussed the system architecture of a ConForm implementation, and we described a prototype implementation called Proto-ConForm.

In the next section, we present an evaluation of the practicality of ConForm based on interviews that were held with several researchers and industrial practitioners.

6.2 Informal Evaluation

We have conducted informal interviews with three researchers and three industrial practitioners so as to receive some preliminary feedback about the overall practicality of ConForm. These interviews were conducted by the author of the dissertation, and lasted between one and two hours each. The interview subjects were given an overview of ConForm, were shown demos of Proto-ConForm, and then were asked some questions about their perceived strengths and weaknesses of the ConForm approach. We note that these interviews were not intended to be a scientific study of user interface practicality, nor do they provide benchmarks of the use of a ConForm implementation for real-world applications. Rather, these interviews were an attempt to evaluate the potential for practical application of the ConForm approach to system
The interview subjects were asked questions related to several areas of concern: motivation, visual aspects, formal aspects, and practicality. All of respondents agreed that there was a real need for practical solutions to the scenario described as “Alice’s Problem” in section 3.1. All of the respondents liked the basic visual representation of the ConForm system modelling notation; however, some expressed concern that it might not “scale up” to larger systems. All of the respondents considered the existence of a formal definition of ConForm to be re-assuring. The respondents were unsure as to the practicality of ConForm in an industrial setting, since no industrial-strength tools existed nor had any large-scale experiments been performed; however, all of the practitioners said that they would be interested in experimenting with an industrial-strength version of ConForm.

6.3 Summary

In this chapter, we examined various implementation and evaluation considerations of ConForm. We discussed the requirements of an implementation of ConForm, and described a prototype implementation called Proto-ConForm. We also summarized the results of some interviews that were held with researchers and software practitioners that discussed the practicality of ConForm. In the next chapter, we illustrate the use of the ConForm approach to system modelling by describing an extended example in which a software developer uses an implementation of ConForm to reconfigure an existing software system.
Chapter 7

The Practical Utility of ConForm

In chapter 3, we motivated the need for a framework for visual system configuration that models component interconnection information. We first described a short fictional industrial scenario involving a newly-hired developer who had been asked to reconfigure a complicated software system, and we examined the likely problems such a developer would have to confront. In the rest of this dissertation, we have presented the ConForm notation for versioned system modelling: we presented its visual model in section 3.3; we discussed modelling differences and similarities between program component version in chapter 4; we gave its formal definition in chapter 5; and in chapter 6, we described a typical process for its use and discussed implementation considerations.

In this chapter we show, by means of a typical software maintenance task, how ConForm can aid in the reconfiguration of a software system. In sections 7.1 and 7.2, we give an overview of the example system and we outline the task to be performed.\(^1\) In section 7.3, we detail the necessary pre-conditions for the use of a ConForm implementation, and in section 7.4 we demonstrate how the key features of ConForm can aid in this maintenance task:

- \textit{convenient access} to the different versions of the system's components is provided by the ConForm repository, which organizes the component versions into version families;

- \textit{understanding of program components} is aided by automated program understanding utilities that extract structural, interface, and interconnection information about the compo-

\(^1\)We do so by expanding upon the scenario presented in section 3.1.
components as they are checked into the repository;

- **understanding of system snapshots** is aided by visualization and navigation tools that comprehend the extracted structural, interface, and interconnection information:

- **understanding of component version histories** is aided by visualization and navigation tools that facilitate the browsing of version families and model inter-version relations:
  and

- **visual system reconfiguration** is aided by a tool that is integrated with the other tools described above, and can perform well-formedness checking of newly constructed systems.

By tracing through the steps that a developer might follow in using a ConForm tool, we will show concretely how these key features of ConForm can aid in reconfiguring an existing software system.

### 7.1 Scenario Background

As mentioned above, the running example we will use in the chapter extends the scenario introduced in section 3.1: NanoSoft Ltd. is a software development company that markets and maintains a small operating system called MiniTunis. MiniTunis has existed for several years: it has undergone extensive evolution, including the addition of many new features, parallel development paths, significant changes to the system design and architecture, and countless bug-fixes. This means that the system design has changed over time, and that each component exists in multiple versions; the structure, syntax, and semantics are not constant across the version space of the system and its components.

The device drivers of the most recent version of MiniTunis, version 95.0, have been found to be unreliable, and NanoSoft therefore requires that a significant reconfiguration be performed. The plan is to extract the device drivers from the previous version of MiniTunis, version 3.1, and adapt them for use with the rest of the version 95.0 system, making whatever changes are necessary to ensure full compatibility.

Figure 7.1 shows a ConForm visual representation of the structure of MiniTunis version 95.0. This particular view shows all of the modules and subsystems, but little interface or intercon-
Figure 7.1: A ConForm snapshot view of MiniTunis version 95.0, highlighting the DEVICE_SYS subsystem. The purple arrows indicate that the Tty and DeviceDriver modules are exported by the DEVICE_SYS subsystem. No other interface or interconnection information is shown in this view.
CHAPTER 7. THE PRACTICAL UTILITY OF CONFORM

nection information. The modules are represented as small boxes with red tabs and no visible subcomponents, and the subsystems are represented as large grey boxes with blue tabs that contain other boxes (the DEVICE_SYS subsystem has been highlighted in this view and is shown in green). The purple tag in the top right hand corner of each module or subsystem indicates its version identifier. The DEVICE_SYS subsystem of MiniTunis v95.0, which is to be replaced, consists of a main driver module (DeviceDriver), individual drivers for several kinds of devices (Tty, Disk, Audio, and Cd_rom), and an auxiliary module used by the Disk device driver (DiskMutex). The purple arrows drawn from Tty and DeviceDriver to the tab of DEVICE_SYS denote that these two modules are exported by the subsystem. However, no other interface or interconnection information is shown in this view.

Figure 7.2 shows ConForm visual representation of the structure of version 3.1 of the DEVICE_SYS subsystem. By comparing it with Fig. 7.1, we can see immediately that the major differences between this version and version 95.0 are that support for cassette drives has been dropped from version 95.0, and two new kinds of devices — audio and CD-ROM — have been added. As in version 95.0, the Tty and DeviceDriver modules are exported for use by clients of the DEVICE_SYS subsystem. This view of DEVICE_SYS version 3.1 also shows the module-level interconnections and requirements of the subsystem and its modules. However, as we shall see, there is a lot of information about the differences between these two subsystem versions that we have not yet examined.

7.2 Problems in the Maintenance Task

Let us suppose that NanoSoft has entrusted this task of reconfiguration to a recently hired developer named Alice. Alice’s assignment — to replace the newer device drivers with the previous versions — is not straightforward. Simply trying to “plug-and-play” the older version 3.1 drivers into MiniTunis v95.0 is likely to cause a number of problems:

Missing features — The older components may lack features provided by the new versions that are required by other modules of MiniTunis v95.0. For example, DEVICE_SYS ver-

---

2We note that no unit (i.e., module or subsystem) from version 3.1 was re-used without at least some changes; consequently, all units in the new system bear the version identifier "95.0".

3We respectfully remind the reader that both Alice and NanoSoft are inventions of the author.
Figure 7.2: A ConForm snapshot view of version 3.1 of the MiniTunis DEVICE_SYS subsystem. This older version supports cassette devices, but not audio or CD-ROM devices as in version 95.0. This view shows the module-level interconnections and requirements of the subsystem and its modules.
CHAPTER 7. THE PRACTICAL UTILITY OF CONFORM

Version 95.0 provides drivers for two kinds of devices not supported by version 3.1: audio devices and CD-ROM drives. These requirements cannot be ignored: either support for them must be added to the version 3.1 drivers, or the rest of MiniTunis v95.0 must be adjusted so that they are no longer required.

**Changed requirements** — The older components may have additional requirements, either at the resource or module level, that are not present in the newer versions. For example, the version 3.1 device drivers require a constant named `cassetteMajor`, which is defined within and provided by version 3.1 of the System module. However, this constant is not defined in version 95.0 of System, since cassette devices are no longer supported. Again, such requirements cannot be ignored; they must either be supported somehow or removed from the version 3.1 drivers.

**Legacy features** — The older components may have "legacy" features not supported in the current system. For example, version 3.1 supports cassette drives but version 95.0 does not. While the mere presence of these "extras" might not cause an immediate problem, the proliferation of needless and outdated features is bad engineering practice. Alice may choose to remove support from cassette drives by editing the code of the version 3.1 drivers, or may choose simply to leave the support in the code. In the latter case, she must make sure future developers understand that support for cassette drives is not part of the current implementation.

**Interface incompatibilities** — The interfaces of the modules and provided routines in the old subsystem may be incompatible with the current ones. If this is so, then Alice will need help identifying the incompatibilities and resolving them.

**Changed semantics** — The semantics of the provided routines may have changed considerably between versions 3.1 and 95.0. In this case, she will need help understanding how and why they have changed; she will need to find the other places within MiniTunis v95.0 where these routines have been used to determine if changes are required.

Care must be taken to address these issues during the reconfiguration effort. As we shall show, appropriate automation can help to draw attention to these problems and ease their
CHAPTER 7. THE PRACTICAL UTILITY OF CONFORM

solution. We next consider how an implementation of ConForm would be tailored for use in such an industrial setting. Then, we show the precise steps a developer such as Alice might follow in performing the system reconfiguration, and how the key features of ConForm can simplify the tasks.

7.3 Tailoring ConForm to MiniTunis

Let us suppose that NanoSoft Ltd. has sought to ease the job of the MiniTunis system maintainers and developers by using an implementation of ConForm, similar to the Proto-ConForm tool discussed in section 6.1.4. However, before an implementation of ConForm can be employed to reconfigure a software system, it must first be hand-tailored for use within the particular industrial setting by specifying appropriate design constraints, constructing a special-purpose ConForm parser, and entering the software system components into the ConForm repository.

The first step in customizing an implementation of ConForm for industrial use is the specification of design constraints appropriate to MiniTunis and NanoSoft’s software design philosophy. For example, the MiniTunis project manager decided that the “whole export rule” from section 5.6 was to be adopted; this constraint states that if a subsystem exports a module, then it must also export all of the (provided) resources of that module to clients of the subsystem. These design constraints may be specified informally (say, as prose) or formally (say, as shown in section 5.6); they comprise a partial set of requirements in building the ConForm parser (described below), and also serve as informational aids to developers browsing the MiniTunis system repository.

Next, NanoSoft’s in-house tools division created a special-purpose ConForm parser for their implementation language, Object-Oriented Turing (OOT). This parser is capable of extracting structural, syntactic, and interconnection information about systems written in the OOT language, and can also verify that all such systems obey the specified design constraints.

Finally, as each new version of MiniTunis has been created, NanoSoft has followed a process as outlined in section 6.1.1, which we now detail.

4Within this chapter, we shall assume the availability of a fully functional implementation of ConForm; we note that the Proto-ConForm prototype described in section 6.1.4 is only a partial implementation.

5Design constraints may be applied globally to all systems stored in the repository, or may be specific to particular systems or subsystems.
7.3.1 Creating A Hierarchic Structure

For each version of the system, a MiniTunis developer has arranged the modules into a hierarchy of subsystems, so that the entire system may be viewed as a tree-like hierarchy of subsystem "snapshots".\(^6\)

For example, as we have seen, Fig. 7.1 shows the system structure of the most recent version of MiniTunis, version 95.0, and Fig. 7.2 shows the system structure of version 3.1 of the DEVICE_SYS subsystem. Additionally, Fig. 7.3 shows the DEVICE_SYS subsystem from MiniTunis v95.0, written in a simple textual subsystem description language. This description lists the modules contained by DEVICE_SYS, and indicates the module and resource exported for use by clients of the subsystem. Note that the all qualifier in the resource export list indicates that all of the resources export by the module are also exported by the subsystem; we can see that this subsystem obeys the "whole export rule" mentioned above.

As discussed below, we can use this kind of simple description as a basis to extract a much more informative and detailed view of the subsystem and its interconnections. Little structural or interconnection information about the components need be modelled at this stage; for example, the particular resource exports of the subsystem are not indicated in any detail. However, we note that this kind of simple structural description is common to industrial approaches to system modelling, such as make [25].

\(^6\)We assume that this organization into subsystems is performed by a human. Other systems, such as Rigi [57], have addressed the issue of automatically creating subsystem views from source code.
7.3.2 Automatically Analyzing Modules

Automated program understanding techniques have been used to derive the precise interconnections between modules in the system. Structural, syntactic, and resource flow information for each module has been extracted automatically from the source code.

For example, Fig. 7.4 shows the source code for the DeviceDriver module of MiniTunis v95.0. DeviceDriver is a “wrapper” module that intercepts calls to the various system devices and reroutes them to the appropriate driver modules. A ConForm textual description of this module is shown in Fig. 7.5. The ConForm system has processed the module to determine the top-level resources, the provided resources, and the module-level and resource-level requirements. This description shows that the basic structure of the module consists of six resource definitions, all of which are exported by the module. There are five module-level dependencies (the module System which contains various global constant and type definitions, plus the four device drivers Disk, Tty, Audio, and Cd_rom), and 30 resource-level requirements. The module-level dependencies are determined by examining the module import list. Extracting the resource-level requirements involves a full-scale parsing of the resource definitions to determine which resources are used but not defined within the module.

We note that Fig. 7.5 represents only a partial list of the characteristics of the DeviceDriver module that were extracted by ConForm from the source code. For example, for the sake of brevity we have shown neither the full source code (which would be stored within the repository) nor the intra-module resource interconnections: LastAddress, Open, Read, Write, and Close each call Valid, and thus Valid fulfills a resource-level requirement of the other five resources.

7.3.3 Automatically Analyzing Subsystems

Concrete subsystem interfaces were created, and precise details of the subsystems-level interconnections were extracted.

Figure 7.6 shows a ConForm description of version 95.0 of the DEVICE_SYS subsystem shown in Fig. 7.1 and Fig. 7.3. This description was generated by processing the subsystem description from Fig. 7.3 and then examining the contained modules (or the ConForm descriptions of the modules, if they have already been processed and entered into the repository). In this way,
CHAPTER 7. THE PRACTICAL UTILITY OF CONFORM

Figure 7.4: Source code for version 95.0 of the DeviceDriver module. This module is basically a “wrapper” that intercepts calls to the various system devices from the rest of MiniTunis and reroutes them to the appropriate driver modules.
CHAPTER 7. THE PRACTICAL UTILITY OF CONFORM

Figure 7.5: ConForm description of the source module from Fig. 7.4 (version 95.0 of the DeviceDriver module).

Figure 7.6: ConForm description of the DEVICE_SYS subsystem from MiniTunis v95.0 from Fig. 7.3.
the ConForm system determined which resources are exported by top-level modules, and which may in turn be exported by the DEVICE_SYS subsystem. ConForm also extracted the resource-level and module-level requirements of the contained units; it determined which requirements may be satisfied by other contained components and which are requirements of the subsystem itself. We have omitted some detail in this description for the sake of brevity: only some of the top-level resources are listed, and intra-subsystem interconnections are not given (Fig 7.2 shows the module-level intra-subsystem interconnections for a different version of DEVICE_SYS).

7.3.4 Creating Version Families

Within the ConForm repository, appropriate module and subsystem version families were created as needed to store the component version descriptions (including the extracted interconnection information).

That is, a module or subsystem version family has been created for each module or subsystem that exists in any version of MiniTunis, including Cassette, Audio, and Cdrom which exist in only some versions of the full system. As each version of each unit was checked into the repository, its ConForm description was added into the appropriate unit version family. Thus, the DeviceDriver version family contains full ConForm descriptions for all of the different versions of this module.

7.3.5 Deriving Relations Between Versions

As each new component version was added to the appropriate version family, the structural, syntactic, and interconnection information extracted by the ConForm system was used to define relations between versions of the same component. These relations, as discussed in section 4.1.2, are mostly based on ideas of structural and interconnection compatibility and similarity, and were derived automatically from the source code.\(^7\) For example, two modules are said to be plug identical if they provide and require the same resources.

Figure 7.7 shows the DeviceDriver module version family; all of the various versions of this module are represented here. Two inter-version relations are shown: a blue arrow indicates that

\(^7\)We remind the reader that we do not address automated modelling of semantic relations in this dissertation.
Figure 7.7: ConForm visual representation of the DeviceDriver module version family for Mini-Tunis. This shows all of the different versions over time. A blue arrow indicates that one version evolved from another. and a red arrow indicates which of these pairs are also plug identical.

one version evolved from another and a red arrow indicates which of these evolutionary pairs are also plug identical. We can see a high correspondence between plug identity and versions whose version identifiers have the same first digit: this should not be surprising, as externally-visible changes (*i.e.*, those that break plug identity, such as altering the signature of an exported procedure) tend to occur mainly at major breakpoints in development.

Figure 7.8 shows how two versions of the same module can be compared at a more detailed level using the ConForm STD notation. Here, version 95.0 of the DeviceDriver module is evolved from relation does not depend on interconnection or structural information. Instead, this relation is determined by tracking the evolution of system components through the ConForm reconfiguration tool in a manner similar to RCS and other SCM tools. A typical behaviour in creating a new component version is to "clone" an existing version, make appropriate changes, and check in the new version. By default, the original version will be recorded as the "evolutionary parent" of the new version within the version family. However, at check-in the developer is free to reset the evolutionary parent to a different version as seems appropriate.

The C-STD notation was introduced in section 4.1.1 and is described fully in appendix A.
being compared to its evolutionary parent, version 3.1. As with the rest of the version 95.0 DEVICE_SYS subsystem, version 95.0 of DeviceDriver supports audio and CD-ROM devices. but does not support cassette drives; consequently, both the module-level and resource-level requirements are slightly different from those of version 3.1, and the two module versions are not plug identical. However, version 95.0 is both tab identical and resource continuous with respect to version 3.1: for each resource defined and exported by version 3.1 of DeviceDriver, there is an analogous resource with an identical signature (but possibly different implementation) that is defined and exported by version 95.0.

### 7.4 Using ConForm to Reconfigure the Target System

Having designed, implemented, and integrated a ConForm parser, and having followed the steps described above on the MiniTunis source code, NanoSoft has laid the groundwork for the use of ConForm in reconfiguring MiniTunis. We note that while the effort required to setup...
ConForm for use with MiniTunis is non-trivial, it is amortized across the entire maintenance cost of MiniTunis and other systems written in the Object-Oriented Turing language. We now consider how Alice might use this ConForm system in her task of replacing the faulty device drivers of MiniTunis v95.0.

7.4.1 Visually Navigating System Snapshots

Alice's first step in reconfiguring MiniTunis using ConForm is to try to gain an understanding of the current version of the system. She does this by finding the system repository for MiniTunis, and querying the ConForm snapshot representations of MiniTunis v95.0 for structural, dependency, and interconnection information using the various graphical browsers of ConForm. Since Alice's main task is to replace the DEVICE_SYS subsystem, she concentrates on learning about the role that the subsystem and each of its components plays relative to the rest of the system. By querying the ConForm repository, she can determine which other components use each of the device driver modules, as well as which of the drivers' exported entities are used and how.

For example, Fig. 7.9 shows the ConForm visual representation of version 95.0 of the DEVICE_SYS subsystem. This view shows the modules and the module-level interconnections of the subsystem as well as the exported resources of each module; the resource-level interconnections within DeviceDriver are also indicated. By inspecting the subsystem interface, Alice notes that only the Device and Tty modules are exported and thus visible to the rest of the system. Because she knows that the ConForm system has examined the entire MiniTunis v95.0 system source code and has verified that the subsystem interfaces are legal and respected by the rest of the system, Alice can be confident that she has an accurate picture of how and where this subsystem may be used. After querying the rest of the system, she discovers that the DeviceDriver module is used only within the FILE_SYS subsystem (the modules FileIO, Inode, and InodeGlobals each call the main DeviceDriver routines), and that the Tty module is used only within the USER_SYS subsystem (the Control module calls Tty.Close during system shutdowns).

Alice now examines the ConForm representation of version 95.0 of DeviceDriver (Fig. 7.5). As Fig. 7.9 also shows, all six of the module's resources are exported, and every resource requires
Figure 7.9: ConForm visual representation of version 95.0 of the DEVICE_SYS subsystem. This view shows the modules, the module-level interconnections, and the resources exported by the modules. Additionally, the resource-level interconnections within the DeviceDriver module are shown.
at least one other resource from each of the required modules. Additionally, she observes that there is relatively little interaction between the module's resources: there are no top-level variables (and hence no state to consider), and the only intra-module resource interconnections involve the `Valid` function which is called by each of the other five routines as a simple "sanity check". After some inspection, she realizes that the reason for this low cohesion within the module is that `DeviceDriver` is actually little more than a wrapper module that reroutes device I/O requests from the rest of the system to the appropriate driver module. It is therefore not surprising that there is no real module state, or that each resource requires resources from each of the individual driver modules. Alice is hopeful that few, if any, changes will have to be made to the `DeviceDriver` module.

Next, Alice looks at the individual device driver modules of MiniTunis v95.0. She first notes that the driver modules have the same interface as `DeviceDriver`: they each define and export six routines with the same names and signatures as those exported by `DeviceDriver`. Again, this is not surprising given that device requests by the rest of the system are handled uniformly via a call to `DeviceDriver`, which then reroutes the call. She notes that the similar syntax of the routines across the different modules might have been better exploited if the system had been written in an object-oriented style and had taken advantage of polymorphism.

Alice pays particular attention to the `Disk` and `Tty` driver modules, which her boss has told her appear to be the source of the reliability problems. She was informed that these modules have been completely rewritten and heavily optimized since the version 3.1 of MiniTunis. In examining the source code, she notes that while the interfaces of the exported routines are straightforward (as mentioned above, they have the same names and syntax as the other drivers' routines), the implementation details are complicated and difficult to follow.

We note that the amount of interconnection and structural information within even such a small and well-designed system as the MiniTunis `DEVICE_SYS` subsystem (Fig. 7.9) can quickly become overwhelming to the viewer.\(^\text{10}\) This is why a ConForm implementation supports a variety of representations and mechanisms for understanding software systems: convenient visual navigation and elision of system snapshots (and version histories), graphical querying, and

\(^{10}\text{A large, poorly-structured system with high coupling between components and low cohesion within components will be difficult to comprehend regardless of the modelling approach used.}\)
concise structured textual representations of system views. These basic features of ConForm, together with user experience, greatly aids in the task of understanding and reconfiguring software systems.

7.4.2 Visually Navigating Component Version Histories

Once Alice has navigated through the various MiniTunis v95.0 snapshots and has made numerous queries so as to understand the system's structure and interdependencies, she is ready to compare it to previous versions of the system. Alice now uses ConForm to browse the version histories of MiniTunis and its components, especially the DEVICE_SYS subsystem. She examines ConForm artifacts such as Fig. 7.10, which shows an abbreviated ConForm-STD summary of the differences between versions 95.0 and 3.1 of DEVICE_SYS. In this way, she is able to learn about the differences between versions 95.0 and 3.1 of this subsystem (some which have been discussed above):

- Driver modules for audio and CD-ROM devices have been added to version 95.0, while the Cassette driver module has been deleted.

- Version 95.0 of DEVICE_SYS has the same module-level dependencies as version 3.1, but has added two new resource-level requirements (audioMajor and cdromMajor) and eliminated one resource-level requirement (cassetteMajor).

- The main driver module, DeviceDriver, has changed only superficially by deleting calls to the cassette driver routines and adding calls to the CD-ROM and audio device driver routines (Fig. 7.8 shows this information).

- The Disk and Tty driver modules have the same interfaces as in version 3.1 (i.e., version 95.0 of each module is tab identical to version 3.1), but the implementations have changed significantly.

- The auxiliary module DiskMutex, which is used only by Disk to manage mutex accesses to the disks, has been altered in minor ways: the names of the two exported routines have been changed from start and finish to Start and Finish respectively, and the initialization code has been optimized slightly. Consequently, DiskMutex version 95.0 is
not tab compatible with version 3.1. However, Alice’s boss has assured her that the new version is reliable and more efficient than version 3.1, and that it should be used in the new version of DEVICE_SYS.

By browsing the version histories of the DEVICE_SYS subsystem and its components, Alice has learned about the particular differences between versions 95.0 and 3.1. She was able to navigate visually through the various version families, and pose queries based on the various inter-version relations that are modelled by ConForm. She was also able to examine differences between component versions using a structured textual description notation, C-STD. With this understanding of the system in hand, she is ready to start the task of reconfiguring MiniTunis.
7.4.3 Visual Reconfiguration and Well-Formedness Checking

Alice now starts to make the required changes using the ConForm visual reconfiguration tool and its well-formedness verifier. Since she was asked to adapt the version 3.1 drivers for use in MiniTunis v95.0, her first action is to “check out” a copy of version 3.1 of the DEVICE_SYS subsystem, which she renames as version 95.1. When she eventually “checks in” the updated subsystem, the ConForm tool will record that version 95.1 actually evolved from version 3.1 rather than 95.0 (unless, of course, she explicitly asks to reset the evolutionary parent to another version).

In checking out a copy of version 3.1 of the DEVICE_SYS subsystem, Alice is given a copy of each of the version 3.1 modules. She first deletes the copy of the Cassette driver module, since support for those devices was dropped from MiniTunis v95.0. Next, she browses the version families of Cd_rom and Audio to find copies of the latest production-quality versions of these modules; she notes the existence of earlier, unreleased versions of the CD-ROM driver, as well as a recent beta-quality version that she has been told to ignore. She then “instantiates” copies of version 95.0 of the CD-ROM and audio drivers into her newly-created version 95.1 of DEVICE_SYS; this scenario is illustrated in Fig. 7.11.

Alice next invokes the ConForm well-formedness verifier on her new subsystem so that she will have an idea of what problems need to be fixed. As it now stands, version 95.1 of DEVICE_SYS contains several inconsistencies that will prevent successful recompilation of the system. For example, version 3.1 of DeviceDriver contains references to the cassette drivers, but not to the CD-ROM or audio drivers (which are expected by the rest of MiniTunis v95.0). A ConForm implementation would not allow such a subsystem to be “checked in”; all newly-constructed components (and their subcomponents) must be verified for well-formedness before they may be added to the repository.

To fix these problems with the DeviceDriver module, Alice could simply “pop-up” a text edit window for this module, make the appropriate changes, and check in the altered module as a new version. However, she recalls that DeviceDriver is really no more than a simple wrapper module; consequently, she deletes version 3.1 from her workspace and instantiates a copy of version 95.0 in its place. Alice makes sure that the DeviceDriver and Tty modules are

161
Figure 7.11: Visual reconfiguration of MiniTunis. At this point, Alice has checked out a copy of version 3.1 of the DEVICE_SYS subsystem, and renamed as version 95.1. She has deleted the Cassette device driver, and added copies of version 95.0 of Cd_rom and Audio. Alice has yet to integrate these disparate components into a coherent whole, but she can use the ConForm tool to help her locate and understand the inconsistencies.
exported from the new subsystem version, and then asks the ConForm tool to verify that this new configuration is well formed. As no new module versions have been introduced so far, the ConForm system is able to use its pre-existing descriptions of the module versions in verifying the new subsystem and in generating a description of it: all of the internal resource-level and module-level interconnections are found to be consistent, the resource export list is generated, and the subsystem's resource-level and module-level requirements are extracted.

Before checking in version 95.1 into the DEVICE_SYS subsystem version family, Alice must ensure that she has included the most recent version of the DiskMutex module, as her boss asked her to do. She therefore deletes version 3.1 of this module and instantiates a copy of version 95.0. However, when she asks the ConForm system to verify this new configuration, she is informed that version 3.1 of Disk has two resource-level requirements that are not fulfilled by the rest of the subsystem: `DiskMutex.start` and `DiskMutex.finish`. Alice recalls that in version 95.0 of the DiskMutex module the initialization code and the names of the two exported routines have been changed. She therefore decides to create a new version of the Disk module, based on version 3.1, that will be compatible with version 95.0 of DiskMutex. She asks the ConForm system to "pop up" a text edit window of Disk version 3.1; she changes all occurrences of `DiskMutex.start` to `DiskMutex.Start`, and `DiskMutex.finish` to `DiskMutex.Finish`. She has now created a new version of the Disk module based on version 3.1.

Once she has finished editing the new module version, the ConForm system automatically asks her for a version identifier for it. Like most version control systems, ConForm does not allow repository members to be altered per se; instead versions may be copied, edited, and then re-entered in the repository as new versions. Alice chooses a version identifier, 95.1, that does not conflict with other versions of Disk and is consistent with the version identifier of the containing subsystem. Note, however, that Alice will not check in this version of the Disk module into the repository until she has built and tested both the new DEVICE_SYS subsystem as well as a full version of MiniTunis.

Alice asks the ConForm system to verify her subsystem once more. ConForm must now perform an analysis on the new version of the Disk module to extract its structural, syntactic, and interconnection information; then, it constructs a ConForm description of the full subsystem. Once again, ConForm verifies that Alice has a well-formed subsystem. Figure 7.12 shows
CHAPTER 7. THE PRACTICAL UTILITY OF CONFORM

Figure 7.12: ConForm visual representation of version 95.1 of the DEVICE_SYS subsystem.

```plaintext
subsystem DEVICE_SYS version 95.1
  comparedTo version 95.0
  Relations:
    evolvedFrom, plugIdentical, tableIdentical, slotIdentical,
    moduleContinuous, resourceContinuous

Units:
  mutates
    Disk  III [modifications deleted]
    Try   III [modifications deleted]

Resources:
  mutates: (10)
    module Disk
      procedure Open (disk : MinorDeviceId)
        141c41
          DiskMuxer.start (disk)
        ...
        143c143
          DiskMuxer.start (disk)
        ...
          DiskMuxer.finish (disk)
        ...
          DiskMuxer.finish (disk)
    : III [reset deleted]

end DEVICE_SYS version 96.1
```

Figure 7.13: ConForm structured textual difference (CSTD) representation of the differences between versions 95.1 and 95.0 of the DEVICE_SYS subsystem.
structure of the final version, and Fig. 7.13 shows the differences between it and version 95.0. We can see, for example, that version 95.1 is tab identical with version 95.0, which means the subsystems are freely substitutable for one another without the possibility of breaking the compile.\textsuperscript{11}

7.4.4 Checking In the New Versions

Having used the ConForm visual reconfiguration tool to aid in the construction of new subsystem and module versions, and having verified the well-formedness of the new components, Alice can now begin the “check in” process. First, she constructs an executable version of the subsystem and performs a standardized regression test suite, using a driver program and test data stored elsewhere within the ConForm repository.\textsuperscript{12} Satisfied with the testing results, Alice is ready to create a new version of the entire MiniTunis system.

Alice next “checks out” a copy of the full version of MiniTunis v95.0, and replaces the version 95.0 DEVICE_SYS subsystem with the version 95.1 that just she has created. This action requires that this new version of MiniTunis be given a new version number; she again chooses 95.1. Alice now builds and tests the full system, version 95.1 of MiniTunis.

Once she (and/or her manager) is content that the reconfiguration task has been completed satisfactorily, Alice checks this new version of MiniTunis into the ConForm repository. Since the ConForm tool has been “watching” Alice’s actions from the beginning, it understands which components have been altered and which have not. Each of the new component versions is added to an appropriate version family, appropriate interconnection information is extracted, new subsystem interfaces are derived as needed, and inter-version relations are modelled as described above. The reconfiguration of MiniTunis is now complete, and the ConForm system has provided important help at many steps along the way.

\textsuperscript{11}Of course, nothing is guaranteed about the relative semantic compatibility.

\textsuperscript{12}It is standard industrial practice to perform system builds and run regression test suites before committing new component versions to a shared source code repository. We note that we have not explicitly addressed system building or testing in this dissertation; we tacitly assume that test suites and build plans have been created and stored elsewhere within the repository.
7.5 Summary

In this chapter, we have sought to illustrate the practicality of the ConForm approach to system modelling by presenting an extended example of the use of a ConForm implementation in reconfiguring an existing software system. We described the necessary pre-conditions for the use of ConForm — the availability of a special-purpose parser, the specification of design constraints, etc. — and we detailed how a ConForm tool can assist a developer in reconfiguring a software system. In particular, we showed how convenient access to the different versions of the various components of a system is provided by the ConForm repository; we showed how the modelling of structural, syntactic, and interconnection information about program components is aided by ConForm's automated program understanding utilities; we showed how comprehension of static system snapshots and component version histories is aided by ConForm's visualization and navigation tools; and we saw how system reconfiguration can be assisted by ConForm's visual reconfiguration tool and well-formedness verifier. By showing how these typical maintenance tasks can be aided by a ConForm implementation, we have illustrated the utility of the ConForm approach to system modelling. In the next chapter, we summarize the research contributions of ConForm, and discuss future research directions.
Chapter 8

Conclusions

In this dissertation, we have presented the ConForm framework for visual system configuration. In the first chapter, we introduced the problems of configuration management and system modelling. In the second chapter, we discussed previous research that had addressed these problems. Next, we presented “Alice's Problem” and motivated the need for a framework for visual system configuration that models component interconnection information; we introduced the ConForm system modelling notation and showed how it models both the snapshot and evolutionary views of software systems. In the fourth chapter, we examined other features of ConForm, such as the modelling of the similarities and differences between program component versions, and the addition of design constraints and programming language constraints. We then presented the formal definition of the ConForm notation, and we discussed implementation and evaluation considerations. Finally, in the previous chapter we gave an an extended example of how ConForm could be used to help re-engineer and reconfigure a software system.

In this chapter, we summarize the research contributions of the ideas presented in this dissertation. We also consider potential extensions to our work, and we discuss possibilities for future research.

8.1 Research Contributions

We consider the primary research contributions of this dissertation to be the following:
CHAPTER 8. CONCLUSIONS

1. The ConForm framework for visual system configuration — The main goal of our research has been to develop a framework for system configuration that improves on existing approaches to system modelling. The ConForm framework is notable because:

- it addresses both the snapshot and the evolutionary views of software systems and their components;
- its system modelling notation models architectural information, such as system structure and component interconnections;
- it supports visual representation and manipulation of system models; and
- it is specializable, both in an abstract and a concrete sense, to model programming language assumptions and user-defined design constraints.

2. Formal definition of the ConForm system modelling notation — We have formally defined the system modelling notation using the Z formal specification language. The formal definition gives ConForm a sound theoretical basis which is uncommon in SCM system modelling notations. The formal definition serves as a mathematically precise description of the basic concepts. It also provides a platform for the abstract specialization of ConForm by the addition of program language assumptions and user-defined design constraints to the formal definition.

Additionally, we consider the following to be secondary research contributions of this dissertation:

1. CalSyCon, the Calculus of System Construction — We have defined a calculus of system construction, called CalSyCon, based on the formal definition of ConForm. This calculus specifies the operations that may be used to construct new subsystems from a repository of program components; these operations define the basic functionality required for a visual system configuration tool.

2. Design of system architecture for an environment that implements the ConForm framework — This system architecture described in section 6.1.3 clearly delineates how an implementation of ConForm is composed of a programming-language-specific
CHAPTER 8. CONCLUSIONS

parser, a repository of program components, and a graphical user interface (which in turn consists of a visual repository browser and a visual system configuration tool). This architecture describes the interactions between the architecture components and users and it defines a blueprint for someone wishing to create or reconfigure an implementation of ConForm.

The next section describes ways in which the work described in this dissertation may be extended.

8.2 Future Work

The research of ConForm can be extended in the following ways:

1. Completion of implementation of Proto-ConForm — The Proto-ConForm prototype implementation of ConForm is incomplete. While the repository and parser components are almost fully functional, the graphical user interface components (the visual repository browser and the visual implementation of CalSyCon) of Proto-ConForm implement an older design of the ConForm system modelling notation.

2. Creation of an industrial-strength implementation — We intend to investigate the practicality of creating an industrial-strength implementation of ConForm. This would require:

   - the creation of an industrial-strength repository mechanism. The Proto-ConForm approach of using the Unix file system as a repository will not scale up for the modelling of large, industrial systems. We will investigate the tailoring of an existing database system.
   
   - the implementation of a ConForm parser that supports an industrial programming language, such as C or C++.

3. Industrial experimentation — Once an industrial-strength implementation has been achieved, experimentation with "real-world" systems may be undertaken. We may then further evaluate the ConForm framework:
CHAPTER 8. CONCLUSIONS

- Do the basic ideas of ConForm "scale up" to large, industrial systems?
- Do the visualisms "scale up"?
- Are the inter-version relations defined in section 4.1.2 useful? What other useful relations should be modelled?
- What do the practitioners think of ConForm?

The next section describes possible directions for future research.

8.3 Research Directions

We consider that the work presented in this dissertation may be applicable to other areas of software engineering research:

1. **Software architecture descriptions** — In this dissertation, we have for the most part ignored the area of software architecture [17, 56, 69, 80, 81, 82]. Software architecture research attempts to characterize the high-level ways in which large software systems are structured and communicate. However, software architecture is a new research field, and many perceive that its research results are still preliminary. Allowing the definition of higher-level dependencies between program components beyond the level of "provides/requires" or "implements/is implemented by" is one way of integrating software architecture ideas into ConForm. As yet, however, it is not clear what these relations should be.

2. **Other SCM issues** — In our research, we have sought to address issues that concern the system modelling sub-area of software configuration management (SCM). We have not explicitly considered other areas of SCM, such as system building and process support. We consider some issues, such as process support, to be beyond the scope of our investigations. However, we intend to investigate system building and how system build recipes may be constructed from ConForm system models. We note that other researchers have investigated this area [47, 1].
8.4 Closing Remarks

Our work has addressed serious problems that exist within industrial software development settings. We are confident that we have contributed to clarifying some of the problems and have suggested simple and useful abstractions for viewing software systems, the interactions between their components, and the relations between different component versions. We hope that the framework we have proposed may aid future software developers in understanding how systems are composed and how they evolve over time. In particular, we hope that we have suggested ways in which automation and visualization may aid the industrial programmer in dealing with the complexity of large, versioned systems.
Bibliography


BIBLIOGRAPHY


[80] Mary Shaw, "Larger Scale Systems Require Higher-Level Abstractions", in [40].


Appendix A

CSTD — ConForm Structured Textual Differencing

In this appendix, we define the C-STD notation (ConForm Structured Textual Differencing); C-STD is the ConForm notation for indicating the differences between two versions of the same component.

The grammar for the C-STD notation is given in Fig. A.1. While the full description of C-STD might appear somewhat detailed for a notation that purports to give a concise view of the differences between two component versions, it should be noted that this grammar lists all of the possible ways in which two component versions may differ. Often, it will be the case that two versions differ very little and the C-STD description of their differences will be short. Additionally, C-STD descriptions give a structured view of the differences, which a simple textual diff, such as the output from the RCS tool rcsdiff, does not.

We now present the C-STD language in detail.

A.1 The C-STD Notation

A C-STD description models the differences between two versions of a program component (i.e., a resource, module, or subsystem). For the purposes of this discussion, we shall call the first component version the primary component and the other the secondary component.

The first two lines of a C-STD description identify the primary and secondary components.


<table>
<thead>
<tr>
<th>compKind</th>
<th>compID</th>
<th>version</th>
<th>uID</th>
</tr>
</thead>
<tbody>
<tr>
<td>comparedTo</td>
<td>version</td>
<td>vID</td>
<td></td>
</tr>
</tbody>
</table>

Relations:
- `relationID`, ...

Exports:
- Resource exports:
  - adds `resourceID`, ...
  - deletes `resourceID`, ...
- Module exports:
  - adds `moduleID`, ...
  - deletes `moduleID`, ...

Requirements:
- Module imports:
  - adds `moduleID`, ...
  - deletes `moduleID`, ...
- Module implements:
  - adds `moduleID`, ...
  - deletes `moduleID`, ...

Resources:
- adds `resourceID`, ...
- deletes `resourceID`, ...

Units:
- adds `moduleDef`, ...
- deletes `moduleID`, ...
- mutates `moduleDef`, ...

Resources:
- adds `resourceDef`, ...
- deletes `resourceName`, ...
- mutates `resourceDef`, ...

end compID version vID

Figure A.1: Grammar for the C-STD notation. C-STD describes the differences between two component versions in a structured way.
The first line lists the component kind (i.e., resource, module, or subsystem), the version family name, and the version identifier of the primary component. The second line gives the version identifier of the secondary component; the component kind and version family name of the secondary component must be the same as that of the primary component, and so it is not necessary to list them.

A C-STD description may contain up to five main clauses. The first clause lists the relations that are known to exist between the primary and secondary component versions. In the example given in section 4.1 (Fig. 4.2), we can see that stack module version 1.2 is tab compatible, slot identical, and weakly plug compatible with version 1.1.

The remaining four main clauses describe how the two component versions differ in terms of their interfaces, requirements, and subcomponents. Additionally, each main clause may have several subclauses. Not all of these clauses need appear in a C-STD description if the two components are identical in that respect, or if the clause is not appropriate for that component kind. For example, if two component versions have the identical interfaces, then it is not necessary to list the Interface main clause. Also, a C-STD comparison of a pair of resources or modules will not have a Units main clause, as only subsystems may contain unit subcomponents.

The Exports clause in a C-STD description indicates how the sets of provided subcomponents differ. It has two subclauses: the Resource exports which describes how the set of provided resources differ, and the Module exports which describes how the set of provided modules differ. The Module exports is appropriate only in the case of comparing two subsystems. The adds section of the Resource exports subclause lists the names of the resources provided by the primary component but not by the secondary, and the deletes sections lists the names of the resources provided by the secondary but not by the primary. The adds and deletes section of the moduleExportList similarly describe the differences in the lists of provided modules.

The Requirements clause describes how the module-level and resource-level slots of the primary and secondary components differ. The Module imports and Module implements describe respectively how the list of modules that are required to fulfill imports and implements relations; these subclauses may only appear in a C-STD comparison of two module or subsystem versions. The Resources subclause describes how the resource-level requirements differ.

The Units clause describes how two subsystems differ in terms of their contained units. If
the primary component contains units not found in the secondary, these are listed in the adds subclause. If the secondary component contains units not found in the primary, these are listed in the deletes subclause. Finally, if the primary contains a different version of a unit contained in the secondary, this is listed in the mutates section.

The Resources clause describes how two modules or subsystems differ in terms of their contained resources. If the primary component contains resources not found in the secondary, these are listed in the adds subclause. If the secondary component contains resources not found in the primary, these are listed in the deletes subclause. Finally, if the primary contains a different version of a resource contained in the secondary, this is listed in the mutates section.
Appendix B

Use of the Z Notation

The formal definition of ConForm given in chapter 5 is written in the Z specification language. However, there are some differences between “standard” Z, and the dialect used here. We now discuss these differences.

The Z Reference Manual [83] is considered to be the standard reference for the Z specification language. However, the shortcomings of “standard” Z as a specification language for large-scale systems are well known [84, 22] and many researchers have developed dialects of Z to suit particular research purposes; for example, a collection of object-oriented extensions to Z has been published [84].

Preliminary specification of the ConForm data model was done using standard Z. However, we were unhappy with the somewhat clumsy way in which standard Z handles some issues, such as recursive structures and polymorphism. Consequently, several of the object-oriented extensions to Z mentioned above were examined, and it was decided to adopt some of the ideas of the Object-Z specification language.

The Object-Z specification language [22, 84] is a dialect of Z that differs from standard Z in several ways, including the introduction of class schemas and schema inheritance, as well as changing some of the fundamental assumptions about Z. Although we found many of the new ideas introduced by Object-Z to be useful, we chose not to adopt Object-Z as the specification language for ConForm. There were two main reasons for this decision. First, even at the time of writing this appendix, the design of Object-Z is still a “moving target”. Private correspondence with the Object-Z developers revealed that the several important changes have occurred to the
APPENDIX B. USE OF THE Z NOTATION

language definition since the publication of the original technical report [22]. Second, it was not clear that many of the extra features introduced by Object-Z would be of immediate use in formally specifying the design of ConForm. In particular, we did not use the Object-Z idea of class schemas and schema inheritance, as we found that standard Z ideas of schemas and schema extension were sufficiently rich to model ConForm.

In the end, we decided that the appropriate course of action would be to stay within the standard Z framework as much as possible, but to employ the few features and assumptions of Object-Z that we found particularly useful. There were two significant features of Object-Z that we adopted:

1. **Forward referencing** — The Object-Z language permits forward referencing in schema definitions, as long as all constructs are finitary. For example, in section 5.4.3, the definition of a Subsystem includes a finite set of ModuleOrSubsystems, which are in turn defined by

   \[ \text{ModuleOrSubsystem} \equiv \text{Module} \cup \text{Subsystem} \]

   Forward referencing is discussed in detail elsewhere [21]. This assumption allowed the convenient representation of recursive structures of complicated schemas. Standard Z does permit a limited kind of recursive structuring, but it is much more restrictive than Object-Z's approach.

2. **Polymorphic core** — In Object-Z, if A and B define schema types, then \( C \equiv A \cup B \), denotes the set of all possible instances of either schema. Object-Z defines the *polymorphic core* of such a schema C to be the set of attributes common to A and B. In Object-Z, it is permissible to reference a "generic" element of the class union C, as long as all referenced attributes are in the polymorphic core. This idea is roughly analogous to *dynamic binding*, an idea used by some object-oriented programming languages such as Smalltalk.

   This issue is discussed in detail elsewhere [20]. The use of polymorphic cores allowed for much greater polymorphism within the specification, which in turn made the specification much more concise than it would have been in standard Z. Standard Z does not handle polymorphism very well; it uses an approach analogous to union types found in programming languages like C.

185
These two assumptions greatly improved the readability and conciseness of the specification. Having adopted these assumptions, we found it much easier to express many of design ideas directly, rather than having to fight with the standard \( Z \) syntax that required extra levels of indirection to state fairly straightforward assertions. When we implemented the prototype system Proto-ConForm, we used the specification as a guide. We feel that the improved readability made for a faster implementation, and it was easier to spot where mistakes had been made.

However, one problem in deviating from standard \( Z \) was that we were unable to use any tools to check the specification. While a variety of tools exist to support the development and checking of specifications written in standard \( Z \), as yet no similar tools exist to support Object-\( Z \). In the early stages of specification, when we were still using standard \( Z \), we were able to type check the specification using the freely-available research tool ZTC (Z Type Checker) [41]. ZTC did find several typographical and syntactic errors in the early specification; however, these errors were low-level and not very serious. Consequently, when we adopted the assumptions of Object-\( Z \) mentioned above, we did not consider the lack of supporting tools to be a significant problem.
Appendix C

The Role of Formal Specification

The design of the ConForm notation for system modelling and its subsequent prototype implementation was done hand-in-hand with its formal specification using a dialect of the Z specification language [83]. We consider that the use of a formal notation was invaluable in the development of ConForm, and we now describe some of the benefits we perceived.

1. Emphasis on simple, mathematical principles — The use of a formal, mathematical specification language to develop and express the design of ConForm entailed that the eventual notation had a high-level, abstract mathematical flavour to it. We consider that this is the appropriate level of abstraction for a system modelling notation. We feel that a major shortcoming of existing approaches, such as Make, has been a concentration on modelling low-level programming language features and peculiarities. We feel that this attention to low-level detail detracts from modelling the high-level nature of the more important tasks at hand, viz. how software systems are decomposed into subsystems, interactions between high-level components, representing subsystem interfaces, and modelling component evolution.

The use of Z encouraged the mindset that program components are, fundamentally, simple and abstract entities that may be combined in a straightforward manner to produce larger-scale components. The use of schemas in Z allowed “essential elements” to be specified independently, and then combined as appropriate to define larger-scale entities using the Z schema calculus. While we showed in sections 4.2.1 and 5.5 that it is possible to
APPENDIX C. THE ROLE OF FORMAL SPECIFICATION

model low-level programming language "peculiarities", our ideas of resources, modules, and subsystems were not dependent on a particular language. Thus, for example, we defined a C language module as a specialization of an abstract module; we did not take the bottom-up approach of defining an abstract module based on the peculiarities of C's data structuring or interconnection model.

2. Concise definition of complex data structures — Another advantage of using a formal specification language to develop ConForm was that it allowed for complex data entities to be modelled concisely. In Z, schemas describe data entities in terms of their structural as well as their semantic properties (i.e., data invariants). As schemas are combined to form larger-scale entities using the schema calculus, the data invariants of the components implicitly form part of the specification of the new structure. This use of modularity and abstraction allowed lower-level components to be used as building blocks in composing higher-level components. The specification of the higher-level components need only state how its lower-level components are interrelated; a lot of semantic information is incorporated implicitly and need not be repeated.

3. Ease of verification and construction of prototype — Finally, as mentioned in section 5.1, we have used the Abstur package of OOT abstract data structures to construct a prototype implementation of ConForm. The main reason for this is that Abstur implements a set of abstract mathematical data structures — such as sets, sequences, and maps — that closely correspond to the fundamental entities of the Z specification language. Using the Abstur package, it was relatively straightforward to create a simple implementation of the Z specification of ConForm. This meant that different design strategies could be tried out quickly and easily; also, since the conceptual distance from specification to prototype was small, it was easy to keep the formal definition and the prototype implementation up-to-date and mutually consistent.

For example, the preliminary design of ConForm used sets of entity names to model collections of components (together with several global name maps to identify the corresponding objects). This proved to be impractical to implement, and also seemed to be a weak design. After researching alternative ways of modelling object containment
and "ownership" in Z and Object-Z [84, 22, 21], a more object-oriented approach was employed; this proved to be a much more satisfactory design. In both cases, the fast turnaround from design to prototype implementation allowed for quick experimentation and evaluation.

For these reasons, we feel that formal specification can play an important role in the design of high-level abstractions, such as the ConForm system modelling notation.