Prototyping a Visual Formalism for System Modelling

Michael W. Godfrey, Richard C. Holt, and Spiros Mancoridis

Department of Computer Science
University of Toronto
10 King's College Road
Toronto, Ontario M5S 1A4
CANADA
e-mail: {migod, holt, spiros}@turing.toronto.edu

Abstract. Formal, visual approaches to system modelling are a promising research sub-area of configuration management. A visual notation for configuring software systems, called ConForm [5, 6], has been designed, formally specified in the Z language, and a prototype is currently being implemented. This paper outlines the design of ConForm, and details experience gained in transforming the formal specification into a prototype.

1 Introduction

Configuration management plays a key role in the engineering of large software systems; it comprises several sub-areas, including version control, access control, system building, and system modelling. *Version control* involves creating an identification scheme for different versions of a system and its components, and ensuring that the scheme is applied when new versions are created. *Access control* involves restricting the access privileges of developers to certain components of the system, as well as dealing with concurrency control issues such as file locking and merging files that have been worked on in parallel. *System building* is the activity of assembling system components into an executable program to run on some target computer system. Finally, *system modelling* involves specifying how versions of atomic source code elements are interrelated and composed into large software systems.

Configuration management systems, as surveyed by Feiler [3] and Dart [1], have successfully attacked many of the problems of associated with version control, access control, and system building. Most of these systems, however, do not feature system modelling notations that can express configurations conveniently and intuitively.

System modelling concepts have also appeared within research on module interconnection languages (MILs) and, more recently, software architecture descriptions. In their foundational paper on MILs, DeRemer and Kron [2] use the term *programming-in-the-large* to denote the activities involved in specifying how versions of software entities are composed into large software systems; they use a MIL as their system modelling notation. Garlan and Shaw's definition of software architecture [4] includes "element composition", an idea analogous to system modelling. Schwanke et al. [16] consider a software architecture description to be the set of allowed connections between software system components, while Perry and Wolf [14] have a similar (but more elaborate) model.

Our research emphasizes the system modelling sub-area of configuration management. Most system modelling notations are textual, yet diagrammatic representations of system compositions seem to be more intuitive. In a recent paper, Harel [8] addresses the need for visual and graphical notations for the specification of large software systems.[1] Also, notations for configuration management are rarely formally defined. A formal definition is essential to guarantee that a notation is unambiguous and free from unpleasant surprises. Our notation for system modelling, called ConForm (*Con*figuration *Form*alism), is visual and formally defined. The formal definition of the semantics of the ConForm notation [6] are written using the Z formal specification language [17].

The next section gives a brief overview of ConForm, together with a short example. Then, the development goals of ConForm are discussed. Next, experiences gained in designing, specifying and prototyping ConForm are detailed. Finally, the current status of ConForm is discussed, and some observations about its development are made.

2 ConForm — A Visual Formalism for System Modelling

ConForm is a visual notation for formally specifying the composition of software systems. ConForm is implemented as a repository of program components (interfaces, module and configuration definitions), a repository browser, and a configuration editor.

ConForm supports system modelling; that is, it helps the user build new configurations of systems from existing components. ConForm entities have both a graphical and a textual representation; ConForm can operate using either representation.

ConForm augments an existing programming language and/or software development environment. Although ConForm is not tied to a particular language or environment, there are several assumptions about the underlying platform:

- The programming language must have a unit of abstraction at the level of a module or class. For simplicity, these components are henceforth referred to as *module definitions*, or just *modules*. A module is represented visually as a box with a thin border.
- Each module *implements* one or more named *interfaces*, which are represented visually by *tabs* drawn on the top of the module box. Main program modules are assumed to implement the special interface `main`.

[1] Although Harel uses the term "system modelling", his application area is limited to reactive systems.

– Each module specifies its required imports by listing the interfaces that provide the required functionality. This is represented visually by *slots* drawn on the bottom of the module box.
– Interface and module definitions are considered to be *atomic*; that is, ConForm does not model programming language constructs of any finer granularity. Additionally, we have made the simplifying assumption that interface, module and configuration names are unique.
– A ConForm *configuration* is a composite entity, made up of instances of modules and other configurations, plus their interrelationships. It is represented visually as a box with a thick border. A configuration may also have tabs and slots; these correspond to the tabs and unresolved slots of its components. A configuration may "export" the tab of any of its components by making it a tab of the configuration. A configuration **must** make all unresolved slots of its components into slots of the configuration.

The ConForm repository (Fig. 1) is partitioned into two parts: a repository of program interfaces, and a repository of program *units* (module and configuration definitions). Within the interface repository, interfaces are represented as ellipses. Relationships between interfaces are allowed; for example, an arrow drawn from one interface to another might indicate that the first extends the second.

In the program unit repository, units are represented as boxes decorated with tabs and slots. Modules have thin borders and configurations have thick borders. Each unit has one or more tabs that indicate which interfaces the unit implements. The slots of a unit indicate which interfaces it requires implementations of.

Modules are considered to be atomic and, consequently, have no visible components. Configurations are composite entities, comprising instances of other units. A configuration definition is *complete* if each of its tabs is bound to the tab of a contained instance and its slots correspond exactly to the set of unresolved slots of the contained instances.

Once a new configuration definition has been completed using the configuration editor, it can be added to the unit repository. Instances of it may then be created and bound into other new configurations. A program unit — module or configuration — may be run as a main program if it implements the interface *main* and has no slots. All other units may be used to construct new configurations.

Figure 1 shows a configuration called FastCalc under construction. It contains an instance of the module Spreadsheet and an instance of the configuration FastStack; the internals of FastStack are also visible (they have been hidden in the *Unit Browser*). FastCalc is not yet complete as the hashDefs slot of Spreadsheet has not been resolved. This slot can be resolved in two ways: either it can be bound to a slot of FastCalc, or an instance of hashTable can be created within FastCalc and its tab bound to the slot. Once the slot is resolved, FastCalc can be checked into the unit repository as a new configuration definition.

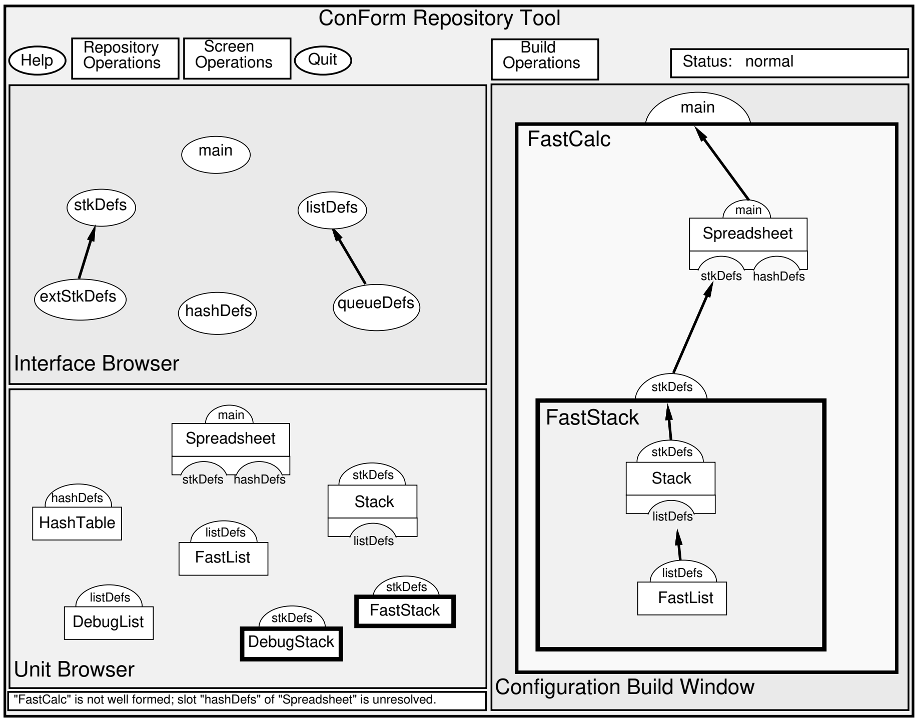

Fig. 1. An example ConForm repository. The interface repository contains entries for all valid interfaces; interfaces are represented visually as ellipses. The unit repository contains both modules (boxes with thin borders) and configurations (boxes with thick borders — the internal details have been hidden). The tabs/slots indicate the interfaces that the units provide/require. New configurations are created in the *Configuration Build Window*; once verified, they can be checked into the unit repository. Note that the configuration under construction, FastCalc, is incomplete, as indicated by the message in the bottom left corner of the screen.

3 Development Goals of ConForm

The primary purpose for designing ConForm was to explore accessible, visual approaches to system modelling. Certain other aspects of configuration management, such as versioning and access control, were not addressed explicitly — these issues have been successfully resolved by other configuration management systems.

Another goal was to investigate the use of formal specification in developing a GUI-based application. Most well-known specification notations, such as Z [17], are inherently textual. It was uncertain if such a specification language would be practical in developing a prototype of a graphical program such as an implementation of ConForm.[2]

[2] Subsequently, Z was used to specify the functionality of the back-end, but was not

4 Design and Specification of ConForm

To validate the design of ConForm, we undertook to formally specify it and to prototype it. Once a preliminary design for ConForm had been mapped out, it remained to decide on an appropriate implementation strategy. We chose to separate out the underlying data structures and operations from the graphical interface as much as possible. First, the textual "back-end" of ConForm would be specified and prototyped. Then, the design of the graphical interface would be updated and prototyped. Finally, the two parts would be merged into a single prototype program.

The back-end of ConForm was fairly complicated: the constraints on the interrelationships between kinds of data objects were complex, and the operations depended on these constraints for their correctness. The back-end was specified using the Z specification language [6]. The results were mixed: the basics of Z were very intuitive and easily learned, but the support for complicated data structuring techniques, especially object-oriented design, was not good.[3] Furthermore, the developers did not have access to any Z tools, and this lack of early feedback resulted in errors in the specification that were not detected until the implementation began.

Even with the above problems, the exercise of translating a precise but informal design into a formal specification was invaluable. Many inconsistencies, redundancies and other design flaws were identified and resolved early in the development cycle.

4.1 Metamorphic Development of ConForm

Transforming the Z specification of the ConForm back-end into a prototype was unexpectedly easy. The Z specification was translated into the Abstur dialect of the Object-Oriented Turing (OOT) language.[4] Abstur (*Abs*tract *Tur*ing) is a superset of OOT that includes abstract data structures, such as sets and sequences, in addition to traditional programming language data structures. The intended use of Abstur is to facilitate the building of prototypes (or executable specifications) from specifications written in model-based notations such as Z. Furthermore, since Abstur is a superset of OOT, a full implementation can be constructed from the prototype by gradually replacing inefficient abstract components with efficient concrete ones. We have referred to this approach elsewhere as *metamorphic* programming [13]: a system under construction undergoes a continuous metamorphosis across development stages.

Using Abstur as the prototyping language had many advantages. First, since Abstur's data structures include sets, functions, and the other data types that are fundamental to Z, the translation of the Z data structures into Abstur was

used to specify the graphical aspects.

[3] A book on object-oriented extensions to Z has recently been published [18].

[4] OOT is an object-oriented programming language similar to Modula and C++ [9, 10].

almost mechanical. Also, since Abstur supports the object-oriented paradigm, it was possible to incorporate aspects of the original design that had not been part of the Z specification, as they had been difficult to express in Z. Finally, test runs of the prototype revealed several errors of intent in the specification. Thus, the use of a prototyping language that includes abstract data structures allowed for the quick detection of errors and early feedback about design decisions.

Once the back-end had been prototyped, the design of the GUI-based graphical front-end for ConForm proved to be fairly straightforward. However, in addition to well-formedness rules, the GUI-level operations also involved graphical constraints; determining these constraints was fairly simple, but their implementation was detailed and tedious work. It is interesting to note that experience with the GUI-level representation also led to much greater insight into the nature of ConForm, and this resulted in some redesign of the back-end.

5 Current Status

Work is still ongoing in developing the graphical front-end of ConForm. However, we feel confident in making several observations based on what we have learned so far. First, we feel strongly that visual formalisms for system modelling are a very promising area of research. As evidence, we note that even after having designed and prototyped the back-end of ConForm, subsequent experience with the visual-level design led to a deeper understanding of the problem and gave insight into the underlying data structures, prompting some fundamental redesign.

We also observed that textual formal specification languages such as Z can be useful in the development of applications with a graphical interface. Although we did not use Z to specify the graphical aspects of ConForm, we found Z to be very useful when specifying the back-end. We also feel that when transforming an object-oriented design into an object-oriented program, it is best to use a specification language that also gives support for the object-oriented paradigm.

Finally, we were pleased with the use of Abstur as the prototyping language. Using a prototyping language that is close in spirit to the specification language and is also notationally related to the final implementation language greatly lessened the "semantic gaps" [13] that developers often encounter when moving between development stages. The quick feedback allowed for the detection of errors and design flaws early in the development cycle, and it was easier to make the required changes in the specification and the prototype.

References

1. Susan A. Dart, "The Past, Present and Future of Configuration Management", Technical Report CMU/SEI-92TR-8, Software Engineering Institute, Carnegie-Mellon University, July 1992.
2. Frank DeRemer and Hans H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small", *IEEE Trans. on Software Engineering*, vol. SE-2, no. 2, June 1976.

3. Peter H. Feiler, "Configuration Management Models in Commercial Environments", Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie-Mellon University, March 1991.
4. David Garlan and Mary Shaw, "An Introduction to Software Architecture", in *Advances in Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, vol. 1, 1993.
5. Michael W. Godfrey, "Visual Formalisms for Configuration Management", *Proc. of IBM CASCON '93*.
6. Michael W. Godfrey, "A Formal Notation for Visualizing Configurations of Software Systems", in preparation.
7. David Harel, "On Visual Formalisms", *Communications of the ACM*, vol. 31, no. 5, May 1988.
8. David Harel, "Biting the Silver Bullet", *IEEE Computer*, vol. 25, no. 1, January 1992, pp 8-20.
9. Richard C. Holt, *Turing Reference Manual*, Third Edition, Holt Software Associates, Toronto, 1993.
10. Spiros Mancoridis, Richard C. Holt and Michael W. Godfrey, "Tools for Software Engineering Education", *Proc. of the ICSE-16 Workshop on Software Engineering Education*, Sorrento, Italy, published as Imperial College Dept. of Computing Technical Report 94/6, June 1994.
11. James G. Mitchell, William Maybury and Richard Sweet, *Mesa Language Manual*, Version 5.0, Technical Report CSL-79-3, Xerox-PARC, April 1979.
12. Marc Moorcroft, "GUILT — A GUI Library for Turing", private circulation, 1993.
13. David A. Penny, Richard C. Holt and Michael W. Godfrey, "Formal Specification in Metamorphic Programming", in [15].
14. Dewayne E. Perry and Alexander L. Wolf, "Foundations for the Study of Software Architectures", *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, October 1992.
15. S. Prehn and W. J. Toetenel (eds.), *VDM '91: Formal Software Development Methods — Proc. of the 4th International Symposium of VDM Europe*, Springer-Verlag Lecture Notes in Computer Science no. 551, October 1991.
16. Robert W. Schwanke, R. Z. Altucher and M. A. Platoff, "Discovering, Visualizing, and Controlling Software Structure", *Proc. of the Fifth International Workshop on Software Specification and Design*, Pittsburgh, USA, 19–20 May 1989.
17. J. M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall International, 1989.
18. Susan Stepney, Rosalind Barden and David Cooper (eds.), *Object Orientation in Z*, Springer-Verlag, Berlin, 1992.

This article was processed using the LaTeX macro package with LLNCS style