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 [God93], 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 modelling, and system building. *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 modelling* involves specifying how versions of atomic source code elements are interrelated and composed into large software systems. Finally, *system building* is the activity of assembling system components into an executable program to run on some target computer system.

Configuration management systems, as surveyed by Feiler [Fei91] and Dart [Dar92], have successfully attacked many of the problems 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 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 [DK76]; they use a MIL as their system modelling notation. Garlan and Shaw's definition of software architecture includes "element composition", an idea analogous to system modelling [GS93]. Schwanke et al. consider a software architecture description to be the set of allowed connections between software system components [SAP89], while Perry and Wolf have a similar (but more elaborate) model [PW92].

Our research emphasizes the system modelling sub-area of configuration management. In particular, we are interested notations that are visual and formally defined. Most system modelling notations are textual, yet anecdotal evidence suggests that software engineers often find diagrammatic representations of system compositions to be more intuitive.[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; furthermore, the availability of a formal specification can aid in constructing a more reliable implementation, as the specification can serve as a formal contract between designer and implementor.

Our notation for system modelling, called ConForm (*Con*figuration *Form*alism), is both visual and formally defined. The formal definition of the semantics of ConForm is written using the Z formal specification language [Spi92]; an abridged version of the specification is presented in the appendix.

The next section of this paper 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 (interface, 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

[1] In a recent paper, Harel addressed the need for visual notations for the specification of large software systems [Har92]. Although Harel uses the term "system modelling", his application area is limited to reactive systems.

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*; this is represented visually by *tabs* drawn on the top of the module box. Main program modules are assumed to implement the special interface `main`. The details of what kinds of entities may constitute an interface, or what it means for a module to implement an interface will depend on the implementation language being modelled.
- 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.[2]

Modules are considered to be atomic and, consequently, have no visible subcomponents. 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

[2] A configuration representing a large system may have many tabs and/or slots and contain many sub-components. Because of this, the browsing mechanism allows for both elision and navigation through the containment hierarchy.

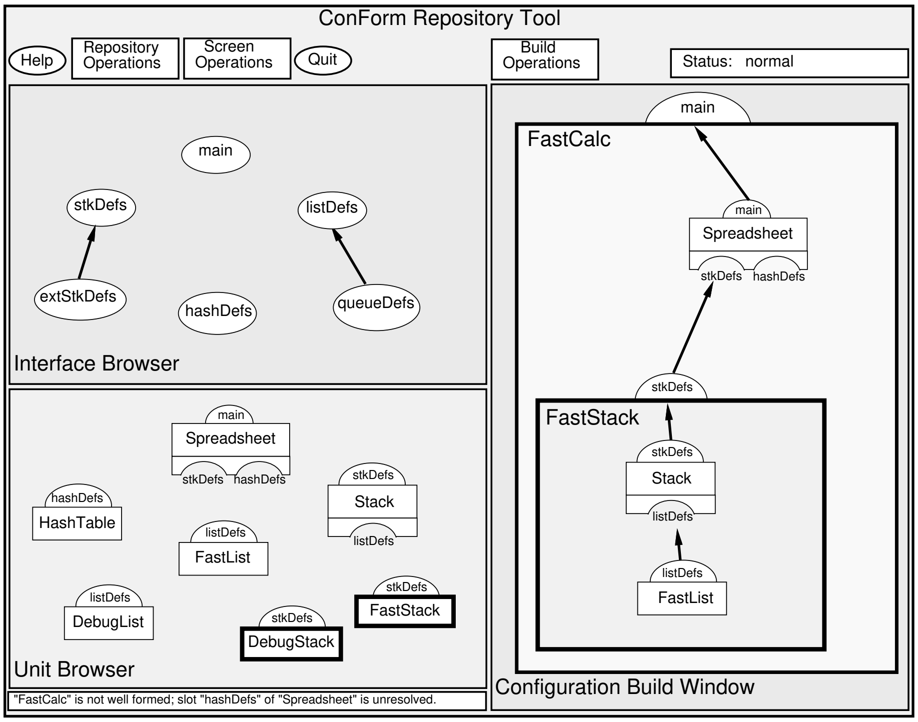

Fig. 1. An example ConForm repository. The interface repository contains entries for all valid interfaces; interfaces are represented visually as ellipses. The unit repository contains both modules (boxes with thin borders) and configurations (boxes with thick borders — the internal details have been 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.

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.

3 Development Goals of ConForm

The primary purpose for designing ConForm was to explore 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 [Spi92], are inherently textual. It was uncertain how such a specification language might be practical in developing a prototype of a graphical program such as an implementation of ConForm.

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. The results were mixed: while the basics of Z were intuitive and easily learned, and Z worked well in modelling the basic mechanisms, 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.

However, even with the above problems we found the exercise of translating a precise but informal design into a formal specification to be invaluable. Many inconsistencies, redundancies and other design flaws were identified and resolved early in the development cycle. We were satisfied that formally specifying the back-end had been worthwhile.

Despite this success, we were less certain as to the benefits of formally specifying the graphical front-end using Z. The graphical interface was to be built using a new library, GUILT [Moo93]. There were two major concerns. First, GUILT was written in a highly object-oriented style, and we were mindful of our problems with using Z to model object-oriented features of the back-end. Second, a formal specification of the front-end would effectively have had to

[3] While some object-oriented concepts — such as encapsulation, extension and genericity — are easily modelled in Z, other concepts — such as polymorphism — are not.

include most of the details of GUILT; we felt that this would make the specification unwieldy without adding much interesting detail. For these reasons, it was decided to forgo formally specifying the front-end.

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 [PHG91]: a system under construction undergoes a continuous metamorphosis across development stages.[5]

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 straightforward. 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.

---

[4] OOT is an object-oriented programming language similar to Modula3 and C++ [Hol93, MHG94].

[5] Metamorphic programming is similar to the idea of refinement/reification familiar to users of Z and similar specification languages [Jon90, Mor90]. However, refinement/reification emphasizes the formal mapping of an abstract representation of a program component to a more concrete one. We use the term metamorphic programming to emphasize that essentially we are engaged in *programming* an evolving system: the intent is to make it easy to move back and forth between abstract and concrete versions of components while staying within a common framework. Also, the link between successive versions of components need not be formal.

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 noted that although textual specification languages such as Z have been used by others to specify graphical programs [Bow92], we did not consider it worthwhile to formally specify the graphical interface portion of our system. We did, however, find it very helpful to formally specify the (textual) back-end of ConForm. 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" [PHG91] 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

[Bow92] Jonathan P. Bowen. "X: Why Z?". *Computer Graphics Forum*, II(4), October 1992.

[Dar92] Susan A. Dart. "The Past, Present and Future of Configuration Management". Technical Report CMU/SEI-92TR-8, Carnegie-Mellon University, July 1992.

[DK76] Frank DeRemer and Hans H. Kron. "Programming-in-the-Large Versus Programming-in-the-Small". *IEEE Trans. on Software Engineering*, SE-2(2), June 1976.

[Fei91] Peter H. Feiler. "Configuration Management Models in Commercial Environments". Technical Report CMU/SEI-91-TR-7, Carnegie-Mellon University, March 1991.

[God93] Michael W. Godfrey. "Visual Formalisms for Configuration Management". In *Proc. of CASCON '93*, Toronto, October 1993.

[GS93] David Garlan and Mary Shaw. "An Introduction to Software Architecture". *Advances in Software Engineering and Knowledge Engineering*, 1, 1993.

[Har92] David Harel. "Biting the Silver Bullet". *IEEE Computer*, 25(1), January 1992.

[Hol93] Richard C. Holt. *Turing Reference Manual.* Holt Software Associates, Toronto, 1993.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International, second edition, 1990.

[MHG94] Spiros Mancoridis, Richard C. Holt, and Michael W. Godfrey. "Tools for Software Engineering Education". In *Proc. of the ICSE-16 Workshop on Software Engineering Education*, Sorrento, Italy, April 1994. Proceedings published as Technical Report 94/6, Dept. of Computing, Imperial College, London, June 1994.

[Moo93] Marc Moorcroft. "GUILT — A GUI Library for Turing". Private circulation, 1993.

[Mor90] Carroll Morgan. *Programming from Specifications*. Prentice Hall International, 1990.

[PHG91] David A. Penny, Richard C. Holt, and Michael W. Godfrey. "Formal Specification in Metamorphic Programming". In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal Software Development Methods — Proc. of the 4th International Symposium of VDM Europe*, October 1991. Proceedings published as Springer-Verlag Lecture Notes in Computer Science no. 551.

[PW92] Dewayne E. Perry and Alexander L. Wolf. "Foundations for the Study of Software Architectures". *ACM SIGSOFT Software Engineering Notes*, 17(4), October 1992.

[SAP89] Robert W. Schwanke, R. Z. Altucher, and M. A. Platoff. "Discovering, Visualizing, and Controlling Software Structure". In *Proc. of the Fifth International Workshop on Software Specification and Design*, Pittsburgh, May 1989.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, second edition, 1992.

A A Formal Model of ConForm

A formal specification of the back-end of ConForm was written using the Z specification language [Spi92]. An abridged version of the specification is now presented, along with prose annotations.

A.1 ConForm Fundamentals

We begin the specification with two given types: ID, the set of legal identifiers, and BODY, the set of atomic programming-language-level entities.

$$[ID, BODY]$$

There are two special identifiers: main and null. The identifier main denotes a predefined interface that is considered to be implemented by all main programs. The identifier null denotes that a tab or slot of a configuration under construction has not yet been bound to a component.

$$\begin{array}{|l} main, null : ID \\ \hline main \neq null \end{array}$$

The namespace of a ConForm repository is called ID. IntfIDs is the set of interface names, UnitIDs is the set of unit names, ModuleIDs is the set of module names, and ConfigIDs is the set of configuration names. Since a unit is either a module or a configuration, UnitIDs the union of ModuleIDs and ConfigIDs. Module and configuration names are unique. No interface or unit may be named null, as null has a special meaning.

$$
\begin{array}{|l}
\hline
\textit{IDs} \\
\textit{IntfIDs}, \textit{UnitIDs}, \textit{ModuleIDs}, \textit{ConfigIDs} : \mathbb{P}\ \textit{ID} \\
\hline
\textit{UnitIDs} = \textit{ModuleIDs} \cup \textit{ConfigIDs} \\
\textit{ModuleIDs} \cap \textit{ConfigIDs} = \varnothing \\
\textit{main} \in \textit{IntfIDs} \\
\textit{null} \notin \textit{IntfIDs} \cup \textit{UnitIDs} \\
\hline
\end{array}
$$

A.2 ConForm Entities

There are two kinds of basic entities in ConForm: interfaces and units. An interface is considered to be atomic, consisting simply of a name and a body.

$$
\begin{array}{|l}
\hline
\textit{Interface} \\
\textit{name} : \textit{ID} \\
\textit{body} : \textit{BODY} \\
\hline
\end{array}
$$

Modules and configurations indicate the functionality they provide/require via tabs and slots. Each tab/slot has an identifier and an associated kind. The tab/slot kind must be the name of an existing interface, except for tabs/slots of a configuration under construction which may temporarily have the kind null.

$$
\begin{array}{|l}
\hline
\textit{Tab} \\
\textit{IDs} \\
\textit{tabID}, \textit{tabKind} : \textit{ID} \\
\hline
\textit{tabKind} = \textit{null} \\
\quad \lor\ \textit{tabKind} \in \textit{IntfIDs} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{Slot} \\
\textit{IDs} \\
\textit{slotID}, \textit{slotKind} : \textit{ID} \\
\hline
(\textit{slotKind} = \textit{null} \\
\quad \lor\ \textit{slotKind} \in \textit{IntfIDs}) \\
\textit{slotKind} \neq \textit{main} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{TabsAndSlots} \\
\textit{tabIDs}, \textit{slotIDs} : \mathbb{P}\ \textit{ID} \\
\textit{idTab} : \textit{ID} \twoheadrightarrow\!\!\!\!\!\!\shortmid\ \textit{Tab} \\
\textit{idSlot} : \textit{ID} \twoheadrightarrow\!\!\!\!\!\!\shortmid\ \textit{Slot} \\
\hline
\mathrm{dom}\ \textit{idTab} = \textit{tabIDs} \\
\mathrm{dom}\ \textit{idSlot} = \textit{slotIDs} \\
\hline
\end{array}
$$

A UnitDef consists of those features common to both modules and configurations: a name, a set of tabs and a set of slots. A module (ModDef) is a unit that contains a programming language body.

$$
\begin{array}{|l}
\hline
\textit{UnitDef} \\
\textit{name} : \textit{ID} \\
\textit{TabsAndSlots} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{ModDef} \\
\textit{UnitDef} \\
\textit{body} : \textit{BODY} \\
\hline
\forall\, t : \textit{tabIDs} \bullet (\textit{idTab}\ t).\textit{tabKind} \neq \textit{null} \\
\forall\, s : \textit{slotIDs} \bullet (\textit{idSlot}\ s).\textit{slotKind} \neq \textit{null} \\
\hline
\end{array}
$$

A configuration contains instances of modules and other configurations. Each instance, or *configuration component*, inherits all the attributes of its definer, which must be an existing unit definition. These attributes are not explicitly included in the schema.

$$\begin{array}{|l}\multicolumn{1}{l}{\_\,\mathit{Instance}\,\_\_\_\_\_\_} \\ \mathit{IDs} \\ \mathit{compID}, \mathit{definer} : \mathit{ID} \\ \hline \mathit{definer} \in \mathit{UnitIDs} \\ \hline \end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{\_\,\mathit{ConfComponents}\,\_\_\_\_\_\_} \\ \mathit{compIDs} : \mathbb{P}\, \mathit{ID} \\ \mathit{idComp} : \mathit{ID} \pfun \mathit{Instance} \\ \hline \operatorname{dom} \mathit{idComp} = \mathit{compIDs} \\ \hline \end{array}$$

A configuration contains three different kinds of bindings:

- A TabToTabBinding binds a component's tab to a tab of the configuration. confTabID identifies the configuration tab, tabCompID identifies the component, and compTabID identifies the component's tab.
- A SlotToSlotBinding binds a component's slot to a slot of the configuration. confSlotID identifies the configuration slot, slotCompID identifies the component, and compSlotID identifies the component's slot.
- A TabToSlotBinding binds a component's tab to a slot of another component. tabCompID and slotCompID identify the components, tabID identifies the tab, slotID identifies the slot. A component may not bind one of its tabs to one of its slots.

$$\begin{array}{|l}\multicolumn{1}{l}{\_\,\mathit{TabToTabBinding}\,\_\_\_\_\_\_} \\ \mathit{confTabID}, \mathit{tabCompID}, \\ \qquad \mathit{compTabID} : \mathit{ID} \\ \hline \end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{\_\,\mathit{SlotToSlotBinding}\,\_\_\_\_\_\_} \\ \mathit{confSlotID}, \mathit{slotCompID}, \\ \qquad \mathit{compSlotID} : \mathit{ID} \\ \hline \end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{\_\,\mathit{TabToSlotBinding}\,\_\_\_\_\_\_} \\ \mathit{tabCompID}, \mathit{tabID}, \mathit{slotCompID}, \\ \qquad \mathit{slotID} : \mathit{ID} \\ \hline \mathit{tabCompID} \neq \mathit{slotCompID} \\ \hline \end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{\_\,\mathit{ConfBindings}\,\_\_\_\_\_\_} \\ \mathit{tabBindings} : \mathbb{P}\, \mathit{TabToTabBinding} \\ \mathit{slotBindings} : \mathbb{P}\, \mathit{SlotToSlotBinding} \\ \mathit{compBindings} : \mathbb{P}\, \mathit{TabToSlotBinding} \\ \hline \end{array}$$

Finally, a configuration definition consists of a name, a set of tabs and slots, a set of components and three sets of bindings.

$$\begin{array}{|l}\multicolumn{1}{l}{\_\,\mathit{ConfDef}\,\_\_\_\_\_\_} \\ \mathit{UnitDef} \\ \mathit{ConfComponents} \\ \mathit{ConfBindings} \\ \hline \forall\, \mathit{tb} : \mathit{tabBindings} \bullet \\ \qquad \mathit{tb.tabCompID} \in \mathit{compIDs} \\ \forall\, \mathit{sb} : \mathit{slotBindings} \bullet \\ \qquad \mathit{sb.slotCompID} \in \mathit{compIDs} \\ \forall\, \mathit{ib} : \mathit{compBindings} \bullet \\ \qquad \mathit{ib.tabCompID} \in \mathit{compIDs} \\ \qquad\quad \wedge\ \mathit{ib.slotCompID} \in \mathit{compIDs} \\ \hline \end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{\_\,\mathit{ConfDefInit}\,\_\_\_\_\_\_} \\ \mathit{ConfDef}' \\ \hline \mathit{tabIDs}' = \varnothing \\ \mathit{slotIDs}' = \varnothing \\ \mathit{compIDs}' = \varnothing \\ \mathit{slotBindings}' = \varnothing \\ \mathit{tabBindings}' = \varnothing \\ \mathit{compBindings}' = \varnothing \\ \hline \end{array}$$

A ConForm repository consists of a set of interfaces, plus a set of units.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Repository}} \\
IDs \\
idIntf : ID \pfun Interface \\
idUnit : ID \pfun UnitDef \\
\hline
\mathrm{dom}\ idIntf = IntfIDs \\
\mathrm{dom}\ idUnit = UnitIDs \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RepositoryInit}} \\
Repository' \\
\hline
IntfIDs' = \{main\} \\
UnitIDs' = \varnothing \\
\hline
\end{array}
$$

A.3 ConForm Operations

Most of the operations detailed here concern the construction of new configuration definitions. StartNewConfig describes the start of a new configuration; initially, there are no tabs, slots, components or bindings. InstantiateInto describes the creation of an instance of an existing UnitDef within a configuration under construction.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{StartNewConfig}} \\
\Xi Repository \\
ConfDef' \\
name? : ID \\
\hline
name? \notin UnitIDs \\
ConfDefInit \\
name' = name? \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InstantiateInto}} \\
\Xi Repository \\
\Delta ConfDef \setminus (name) \\
\Xi TabsAndSlots \\
\Xi ConfBindings \\
Instance \\
\hline
compID \notin compIDs \\
idComp' = idComp \cup \\
\qquad \{compID \mapsto \theta Instance\} \\
compIDs' = compIDs \cup \{compID\} \\
\hline
\end{array}
$$

To add a tab to a configuration under construction, it is necessary only to provide a name for it; the tab kind is initially set to null. The tab kind must eventually be reset to the name of an existing interface. This can be done explicitly by SetConfTabKind or implicitly by BindConfTab, in which case the configuration tab kind is set to that of the indicated component tab. Note that there is a restriction on the use of SetConfTabKind: the configuration tab in question must not have been bound to. We omit the specification of the analogous operations AddConfSlot and SetConfSlotKind.

$$
\begin{array}{|l}
\hline
\textit{AddConfTab} \\
\Xi IDs \\
\Delta ConfDef \setminus (name, slotIDs, idSlot) \\
\Xi ConfComponents \\
\Xi ConfBindings \\
Tab \\
\hline
tabID \notin tabIDs \\
tabIDs' = tabIDs \cup \{name\} \\
idTab' = idTab \cup \{name \mapsto \theta Tab\} \\
tabKind = null \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{SetConfTabKind} \\
\Xi IDs \\
\Delta ConfDef \setminus (name, slotIDs, idSlot) \\
\Xi ConfComponents \\
\Xi ConfBindings \\
\Delta Tab \\
tabID?, tabKind? : ID \\
\hline
tabID? \in tabIDs \\
tabID? \notin \{tb : tabBindings \bullet \\
\quad tb.confTabID\} \\
idTab\ tabID? = \theta Tab \\
tabKind? \in IntfIDs \\
tabKind' = tabKind? \\
idTab' = idTab \oplus \{tabID? \mapsto \theta Tab'\} \\
\\
tabID' = tabID \\
tabIDs' = tabIDs \\
\hline
\end{array}
$$

For a configuration definition to be well formed, each of its tabs must be bound to a tab of a component; BindConfTab performs such a binding. Note that each configuration tab must (eventually) be bound to exactly one component tab, and that the initial kind of the configuration tab must be either null or the same as that of the component tab. We omit the specification of the analogous operation BindConfSlot.

$$
\begin{array}{|l}
\hline
\textit{BindConfTab} \\
\Xi Repository \\
\Delta ConfDef \setminus (name, slotIDs, idSlot, slotBindings) \\
\Xi ConfComponents \\
\Delta Tab \\
TabToTabBinding \\
\hline
confTabID \in tabIDs \\
idTabconfTabID = \theta Tab \\
tabCompID \in compIDs \\
compTabID \in (idUnit\ (idComp\ tabCompID).definer).tabIDs \\
\forall tb : tabBindings \bullet tb.confTabID \neq confTabID \\
tabKind \in \{null, ((idUnit\ (idComp\ tabCompID).definer).idTabtabID).tabKind\} \\
tabKind' = ((idUnit\ (idComp\ tabCompID).definer).idTabtabID).tabKind \\
tabBindings' = tabBindings \cup \{\theta TabToTabBinding\} \\
idTab' = idTab \oplus \{confTabID \mapsto \theta Tab'\} \\
\\
tabID' = tabID \\
tabIDs' = tabIDs \\
\hline
\end{array}
$$

ConnectTabToSlot binds a tab of one component to a slot of another. This corresponds to resolving the required imports of a component. There are several restrictions: the two components must be components of the same configuration under construction; the slot must not have been bound into a slot of the containing configuration (since such component slots are assumed to be unimplemented within the defining configuration) nor may the slot have been bound to another component tab; and the tab kind must be "compatible" with the slot kind. The relation canImplement defines tab/slot compatibility. As discussed above, what it means for a tab kind to be with "compatible" a slot kind will vary with the underlying platform; canImplement may be defined differently if the underlying platform allows a more interesting idea of tab/slot compatibility. The naive rule (shown below) is that the tab and slot kind must be identical.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CanImplement}} \\
\hline
IDs \\
canImplement : ID \leftrightarrow ID \\
\hline
\forall\, tabKind, slotKind : ID \bullet tabKind \ \underline{canImplement}\ slotKind \Leftrightarrow \\
\qquad (\{tabKind, slotKind\} \subseteq IntfIDs \land tabKind = slotKind) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ConnectTabToSlot}} \\
\hline
\Xi Repository \\
\Delta ConfDef \setminus (name, tabBindings) \\
\Xi TabsAndSlots \\
\Xi ConfComponents \\
CanImplement \\
TabToSlotBinding \\
\hline
\{tabCompID, slotCompID\} \subseteq compIDs \\
((idUnit\ (idComp\ tabCompID).definer).idTab\ tabID).tabKind\ \underline{canImplement} \\
\qquad ((idUnit\ (idComp\ slotCompID).definer).idSlot\ slotID).slotKind \\
\forall\, ib : compBindings \bullet \neg\, (ib.slotCompID = slotCompID \land ib.slotID = slotID) \\
\forall\, sb : slotBindings \bullet \neg\, (sb.slotCompID = slotCompID \\
\qquad \land\ sb.compSlotID = slotID) \\
compBindings' = compBindings \cup \{\theta\, TabToSlotBinding\} \\
slotBindings' = slotBindings \\
\hline
\end{array}
$$

Finally, a configuration may be added to the repository only if it is well formed. A configuration is well formed iff:

- it has at least one tab,
- each of its tabs has been bound to a tab of a component,
- each of its slots has been bound to a slot of a component, and
- each slot of each component has been bound to a slot of the configuration or to a tab of another component.

We omit the specifications for the analogous operations AddModDef and AddInterface.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{WellFormedConfDef}} \\
\hline
\textit{IDs} \\
\textit{Repository} \\
\textit{ConfDef} \\
\hline
\#\textit{tabIDs} > 0 \\
\forall\, \textit{tabID} : \textit{tabIDs} \bullet \exists\, \textit{tb} : \textit{tabBindings} \bullet \textit{tb}.\textit{confTabID} = \textit{tabID} \\
\forall\, \textit{slotID} : \textit{slotIDs} \bullet \exists\, \textit{sb} : \textit{slotBindings} \bullet \textit{sb}.\textit{confSlotID} = \textit{slotID} \\
\forall\, \textit{compID} : \textit{compIDs} \bullet \\
\quad \forall\, \textit{slotID} : (\textit{idUnit}\ (\textit{idComp}\ \textit{compID}).\textit{definer}).\textit{slotIDs} \bullet \\
\qquad (\exists\, \textit{ib} : \textit{compBindings} \bullet \textit{ib}.\textit{slotCompID} = \textit{compID} \\
\qquad\quad \wedge\ \textit{ib}.\textit{slotID} = \textit{slotID}) \\
\qquad \vee\ (\exists\, \textit{sb} : \textit{slotBindings} \bullet \textit{sb}.\textit{slotCompID} = \textit{compID} \\
\qquad\quad \wedge\ \textit{sb}.\textit{compSlotID} = \textit{slotID}) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddConfDef}} \\
\hline
\Delta \textit{Repository} \setminus (\textit{ModuleIDs}, \textit{IntfIDs}, \textit{idIntf}) \\
\textit{ConfDef} \\
\hline
\textit{WellFormedConfDef} \\
\textit{name} \notin \textit{UnitIDs} \\
\textit{ConfigIDs}' = \textit{ConfigIDs} \cup \{\textit{name}\} \\
\textit{idUnit}' = \textit{idUnit} \cup \{\textit{name} \mapsto \theta\, \textit{ConfDef}\} \\
\hline
\end{array}
$$