Visual Formalisms for Configuration Management

Michael W. Godfrey
Department of Computer Science
University of Toronto
and
IBM Centre for Advanced Studies

Abstract

As reuse of software components becomes more commonplace, being able to understand, manipulate and reason about software system architectures acquires new importance. Although commercial software development environments have addressed many of the issues of configuration management, there is still a need for visual formalisms that can aid in representing and manipulating architectures of software systems. This paper introduces ConForm (*Con*figuration *Form*alism), a graphical notation for representing configurations of software systems. Several of the basic concepts of ConForm were inspired by the C/Mesa language [8]; however, ConForm is both language and tool independent. ConForm is notable because it is both a visual and a formal approach to representing software architectures.

1 Introduction

Configuration management plays a key role in the engineering of large software systems; it comprises several sub-areas, including versioning, access control, system modelling and system building. While several commercial software development environments have successfully attacked many of the problems of configuration management, most approaches have been tightly integrated with a particular set of tools. Consequently, there is a need for notations and tools for configuration management that are both powerful and intuitive yet not bound to particular environments or languages. Visual notations hold promise, yet they have not yet been investigated very thoroughly.[1]

This paper introduces ConForm (*con*figuration *form*alism), a visual formalism for representing software architectures. Many of the concepts in ConForm were inspired by the C/Mesa language [8]; however, ConForm is programming language independent. Indeed, much of the investigation into ConForm was done using example systems written in a variety of programming languages, including Mesa and Object-Oriented Turing [6].

The main intended benefit of ConForm is to facilitate system modelling. Other aspects of configuration management — such as versioning, access/concurrency control and system building — are not explicitly addressed by ConForm. The reasons are manifold: different versions of a program unit can be considered to be different implementations of the same interface; access control is inherently process and/or tool dependent; and once a system model has been completely configured, system building (loading, binding and running) is then a straightforward mechanical process that can be carried out by the tools of the particular environment. System modelling, however, is common to all languages and environments

[1] "It is our duty to forge ahead to turn system modelling into a predominantly visual and graphical process. I believe this is one of the most promising trends in our field." [5]

that have the concept of configurations, and many of the issues of system modelling can be attacked at a language-independent level. This is the area in which ConForm is intended to aid.

2 An Overview of ConForm

A ConForm implementation consists of a repository of software components, tools for viewing and querying the repository, plus mechanisms for changing existing components or creating new ones (which then may be added to the repository). ConForm is intended to aid system modelling; that is, it is intended to help the user build new configurations of systems from existing components. ConForm entities have both a graphical and a textual representation; ConForm's tools can operate using either representation.

ConForm is intended to augment 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; these will be discussed below.

Program Units

ConForm requires that the underlying programming language have a unit of abstraction at the level of a class or module. This concept is called a a "program unit" or just "unit" in ConForm. ConForm does not model language concepts of any finer granularity than the program unit.

A program unit is represented graphically by a blue box labelled with the unit's name. The boxes may also be "decorated" by green tabs and slots whose labels indicate the interfaces the unit implements and requires. Figure 1 shows a simple repository of four units. The left-most box represents an Object-Oriented Turing (OOT) class named `StackImpl`; it implements an interface named `stackDefs` and requires an instance of a class that implements the interface named `listDefs`.

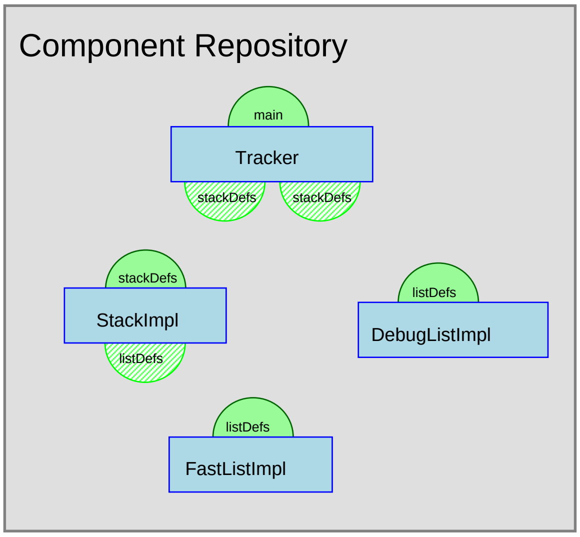

Figure 1: A simple repository. The blue boxes represent program units. The green "tab" on the top indicates the interface the unit implements. The green "slots" (striped tab) indicated the required imports.

The textual representation of a program unit is, of course, entirely dependent on the underlying programming language. As this paper is aimed at in-the-large concepts, the textual equivalents of the units in Figure 1 have been omitted.

Program Units *Implement* Interfaces

Each program unit is assumed to *implement* a named *interface*. Although this rule might seem restrictive, it can be added on top of most conventional programming languages with minimal effort. In some languages, such as Modula-3 [1] and Mesa [8], this convention is actually part of the language.[2] In oth-

[2] In practice, Modula-3 programmers often omit creating an explicit interface when no other Modula-3 module will implement the same interface. In this case, a default interface is considered to be created with the same name as the implementing module. However, the language definition presumes the existence of the interface, even if it is not created explicitly by the programmer.

ers, such as Object-Oriented Turing [6], it is optional; in this case, interfaces can be trivially extracted from unit definitions and given names.

In most programming languages, an interface provides a listing of items that will be implemented and "exported" by a subsequent program unit. For example, an interface for a stack abstract data type might be called `stackDefs` and define entry points for procedures called `Push` and `Pop`. One can further imagine the existence of several program units that implement `stackDefs`, such as a fast implementation, and a slower one that provides a tracing facility.

It should be noted, however, that the particular kinds of entities contained in an interface is not a concern of ConForm. At the very least, it might reasonably be expected that an interface contain some syntactic information, such as signatures of exported procedures. More information might also be present, such as pre- and post-conditions or even default implementations. But precisely what constitutes an interface will depend on the underlying programming language.

For the sake of uniformity, ConForm requires that a main program within a system also implement an interface. Consequently, there is a special interface called `main` that is considered to be implemented by programs intended to be used as the main program in a system. The `main` interface is not fundamentally different from any other interface; it exports a single procedure called `Run` that initializes the system and starts it running.

A further restriction on the underlying programming language is that each program unit must completely implement exactly one interface. This restriction makes it difficult to model a language such as Modula-3, where a module may implement all or part of multiple interfaces. This restriction may yet prove to be easy to relax, but for now simplicity is felt to be a more important design goal than such a flexible but potentially confusing language feature.[3]

[3]Modula-3 is a descendant of Mesa, which also has this flexibility. This is a point of divergence between ConForm and C/Mesa.

The *implements* relationship is indicated graphically by a green "tab" on top of the blue box representing the program unit. The tab is labelled with the name of the interface implemented by the program unit. Configurations may also implement interfaces; this will be discussed later.

Program Units Define *Classes* and *Import* Unit Instances

ConForm supports object-based computing [9]; that is, it supports instantiation of program units. Thus, a program unit defines a *class* of entities rather than a single entity.[4] Instantiation of program units occurs only within *configurations*; if a client unit requires an instance of another server unit, the client unit may specify only the name of the interface of the server unit. It is the responsibility of the configurer to decide which instance of which implementation to provide to the client unit.[5]

If a program unit `P` requires an instance that implements interface `I`, it is indicated graphically by a green "slot" drawn on the bottom of the box that represents unit `P`. The slot is labelled by the name of the required interface. If a unit requires multiple instances of the same interface, then a slot is shown for each required instance. For example, Figure 1 shows the unit `Tracker` requiring two stacks.

A program unit instance is drawn as a box with a striped interior. The instance name appears near the centre of the box; the name of the corresponding class is indicated in small type at the bottom right. An instance has the same tabs and slots as its parent unit. Figure 2 shows several unit instances within a configuration. Note that `s1i` and `s2i` are distinct

[4]ConForm does not yet support class inheritance. This possible extension is discussed at the end of the paper.

[5]A configuration definition also defines a class rather than an instance; configuration definition and instantiation is discussed below.

instances of the same class.

Configurations

As mentioned above, program units define classes rather than instances; instantiation is done at configuration time. A *configuration* is a construct in which units are instantiated and import requirements are resolved by binding instances together.

A configuration is drawn as a red box. Within the configuration reside instances of program units (and possibly sub-conifigurations).

Imports are resolved by binding instances that implements a given interface to instances requiring an instance of that interface. This binding is indicated graphically by a thick arrow from tab (exported interface) to slot (required interface). Each instance has its own data space and may be bound into more than one slot; thus, two unit instances may, for example, share the same stack. A system is fully configured once all slots have been filled in.

The internals of a displayed configuration may be hidden from view most of the time; usually, all that is important to someone examining the configuration is an indication of which interface it implements and which ones it requires.

Sub-Configurations

In many ways, a configuration can be considered to be a kind of "super-unit", as it has much in common with program units. Both configurations and units have names, can be instantiated and are assumed to implement an interface. A self-contained configuration, as exemplified by `c1` in Figure 2, can be considered to implement the `main` interface. A sub-configuration, as exemplified by `FastStackConf` and `DebugStackConf` in Figure 4, may implement an interface by creating an instance of a unit that implements the interface and exporting the instance. Finally, a configuration may also require an instance that implements an interface in the same way that a unit can.

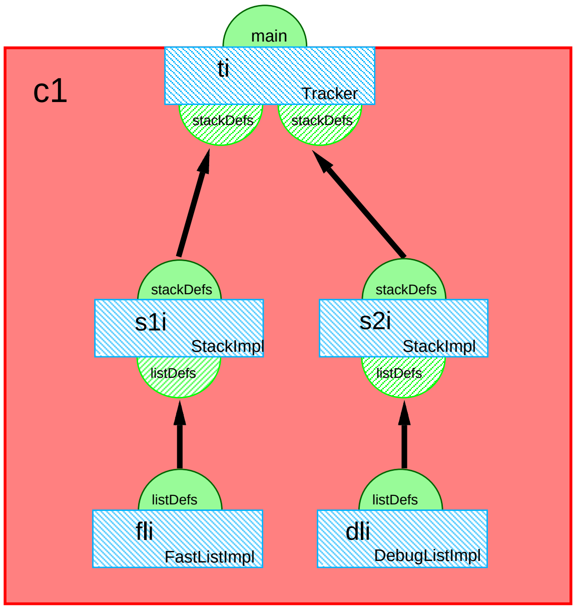

Figure 2: A simple configuration. Each blue box represents an instance of a program unit found in the repository of Figure 1. The imports of each unit instance are resolved by linking each slot with a tab. This system is fully configured since all slots have been filled in.

```
configuration c1
    implements main by ti
    configure
        var fli : FastListImpl
        var s1i : StackImpl [fli]
        var dli : DebugListImpl
        var s2i : StackImpl [dli]
        var ti  : Tracker [s1i, s2i]
end c1
```

Figure 3: The textual representation of the configuration `c1`.

Logically, sub-configurations are unnecessary; any configuration that uses sub-configurations can also be defined using only unit instances. However, having a repository of well-used components clearly aids in system construction and in the quest for the object-oriented "holy grail" of component reuse.

The graphical representation of configurations is the same as that of units, except that configurations are red instead of blue. Interfaces implemented or required are indicated in the same way (green tabs and slots), and configuration instances are indicated by striped red boxes. Note that the tab of a configuration corresponds to the tab of an instance it instantiates and exports. Furthermore, the slots of a configuration correspond to the unresolved slots of internal unit instances.

The Repository

As mentioned above, a ConForm implementation consists of a repository of components, tools for querying the repository, and mechanisms for creating/modifying/adding/deleting repository elements. The ConForm repository stores two kinds of components: program units and configurations. However, these entities define classes, not instances.[6] To get a running system, the user simply selects a complete configuration from the repository, or creates one using repository components. The system building process is then automatic; the mechanics will depend on the underlying platform.

Constructing a complete configuration from repository components bears some explanation. An extended example is presented in the next section.

The organization of the repository has been deliberately underspecified. Clearly, having a flat namespace of repository elements is not practical for more than a very small system. However, repository organization is considered to be an implementation issue, and will also depend on the particular tool platform being used. Consequently, no more will be said about repository organization.

[6] Unit and configuration instances within a configuration can be thought of as attributes of the class; no space is allocated for them until an instance of the containing configuration is created.

3 Constructing a Configuration — An Example

Consider the example repository of Figure 1. There are four unit definitions in the repository: `Tracker`, a main program that requires two stacks (*i.e.*, instances of units that implement the interface `stackDefs`); `StackImpl`, which implements a stack and requires a list; and `FastStackImpl` and `DebugStackImpl`, both of which implement lists. Since `Tracker` is the only main program in the repository (it is the only component that implements the `main` interface), let us consider constructing a configuration that completely implements an instance of it.

To start building a new configuration, we choose the `Build New Config` option from the main ConForm menu. This creates a new window containing an untitled configuration box. Thus, as our first action on the new configuration, we can give it a name, say `c1`.

Next, we browse the repository for instances to add to the new configuration. To add an instance of `Tracker` to our configuration, we click on the box representing `Tracker` in the repository and drag the mouse into the new configuration window. The result is that an unnamed instance of `Tracker` now appears within the box representing our new configuration `c1`. Furthermore, since `Tracker` implements the interface `main`, the new instance is automatically positioned at the top of `c1` with its green interface tab extending beyond the top boundary of `c1`; this indicates that `c1` exports the interface `main`, and thus contains a main program.

The act of clicking on a component (unit or configuration) in the repository and dragging it into a configuration under construction corresponds to component instantiation. The new instance must be given a name by the configurer, or the system can provide a default one.

ConForm uses context to constrain the building of configurations. The instantiation of a main program unit described above is one example. Trying to instantiate multiple main programs in a single configuration would (correctly) be disallowed by the system.

So far, `c1` consists of a single instance of `Tracker`. Let's call the instance `ti`. We note that two instances that implement `stackDefs` are required (*i.e.*, these are the tabs attached to the bottom of `ti`). By double-clicking on one of the tabs, all elements of the repository that implement `stackDefs` are highlighted. In this case, only the unit `StackImpl` satisfies the query. We can now "click and drag" twice to instantiate two instances of `stackDefs`; let us name them `s1i` and `s2i`. Next, we perform *binding*: we click and hold on the tab (exported interface) of `s1i` and release the mouse button when the mouse is pointing to one of the slots of `ti`. Since the tab and slot refer to the same interface, the binding is legal and `s1i` is now considered to be bound into the slot of `ti`. This is indicated graphically by a thick arrow from `s1i` to `ti`. Similarly, we can bind `s2i` to the second tab of `ti`.

We could decide to stop at this point and simply store the new configuration in the repository. If we did so, the ConForm configuration editor would detect that there were two unresolved slots, and these would become slots of the configuration `c1`. A configuration with unfilled slots is not capable of being built as a system; however, it can be instantiated within a new configuration and have its slots filled in there.

To make `c1` a complete configuration, the imports of `s1i` and `s2i` must be resolved. Each requires an instance of a unit that implements the interface `listDefs`. A double-click on the interface slot reveals that there are two such units within the repository: `FastListImpl` and `DebugListImpl`. We can create one instance of each and bind them in to the slots of `s1i` and `s2i`. The configuration is now complete; since neither of the list instances have slots, there are no more slots to be filled in.

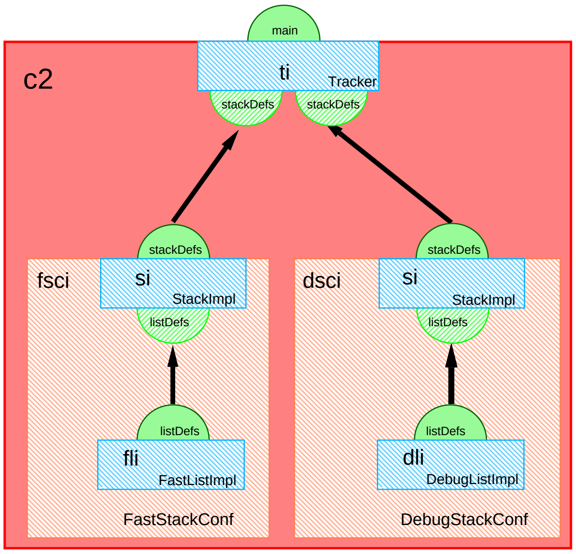

Figure 4: A configuration that uses sub-configurations. Usually, the internals of the sub-configurations are hidden; the interfaces that it provides and/or requires are of most interest.

We can now add the new configuration `c1` to the repository. Figure 2 illustrates the final version of `c1`. Figure 3 gives the textual equivalent.

A repository entity (configuration or program unit) is said to be *complete* if it implements `main` and has no (unfilled) slots. This is the case with our newly-created configuration `c1`. A complete component can be built, but exactly what this entails will depend on the underlying platform. However, this is transparent to the configurer.

4 Summary and Research Directions

Currently, no tools exist to perform checking or enforce constraints. However, previous experience with the Object-Oriented Turing (OOT) environment [7] suggests that building and integrating ConForm tools into the OOT system should be straightforward. This will be

```
configuration FastStackConf
    implements StackDefs by si
    configure
        var fli : FastListImpl
        var si  : StackImpl [fli]
end FastStackConf

configuration DebugStackConf
    implements StackDefs by si
    configure
        var dli : DebugListImpl
        var si  : StackImpl [dli]
end DebugStackConf

configuration c2
    implements main by ti
    configure
        var fsci : FastStackConf
        var dsci : DebugStackConf
        var ti   : Tracker [fsci.si, dsci.si]
end c2
```

Figure 5: The textual representations of configurations `FastStackConf`, `DebugStackConf` and `c2`.

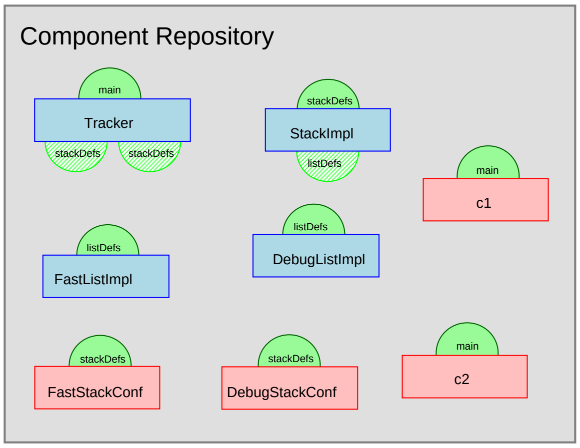

Figure 6: Repositories may contain both program units and configuration classes.

investigated in the near future.

ConForm is also restrictive in that it requires a program unit to implement exactly one interface and it insists that a program unit specify only the underlying interface of a unit it wishes to use. These restrictions were introduced mainly to simplify the basic paradigm of ConForm. Once more experience has been gained, these restrictions may be relaxed.

Currently, ConForm provides no object-oriented features, apart from the ability to instantiate program units and configurations. This is not a considered a major shortcoming, as many industrial programming languages do not support the object-oriented paradigm. However, if the underlying programming language does support class inheritance, a trivial extension to the "implements" relation allows ConForm to model class inheritance: we consider that class `C` implements interface `I` if `I` is the "natural" interface of `C` or an inheritance ancestor of `C`. This trivial extension allows object-oriented software systems to be modelled easily by ConForm. Despite this extension, object-orientation in ConForm has not yet been fully explored: inheritance of configurations has not yet been addressed, and this is a future research topic.

About the Author

Michael Godfrey is a Ph.D. candidate at the University of Toronto and is member of the IBM Centre for Advanced Studies. His research interests include making formal specification and other formalisms practical for industrial software engineering. He can be reached by email at *migod@turing.toronto.edu*. His thesis supervisor is Richard C. Holt.

References

[1] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow and Greg Nelson, "Modula-3 Report (revised)", Technical Report 52, Digital Systems Research Center, November 1989.

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

[3] David Garlan, Mary Shaw, Chris Okasaki, Curtis M. Scott, and Roy F Swonger, "Experience with a Course on Architectures for Software Systems", Technical Report CMU/SEI-92-TR-17, August 1992.

[4] David Harel, "On Visual Formalisms", *Communications of the ACM*, vol. 31, no. 5, May 1988.

[5] David Harel, "Biting the Silver Bullet", *IEEE Computer*, vol. 25, no. 1, January 1992, pp 8-20.

[6] R. C. Holt, *Turing Reference Manual*, Third Edition, Holt Software Associates, Toronto, 1993.

[7] Spiros Mancoridis, R. C. Holt and David Penny, "A Conceptual Framework for Software Development", *Proc. of the Twenty-First ACM Computer Science Conference*, Indianapolis, Indiana, February, 1993.

[8] James G. Mitchell, William Maybury and Richard Sweet, *Mesa Language Manual*, Version 5.0, Technical Report No. CSL-79-3, Xerox-PARC, April 1979.

[9] Peter Wegner, "Dimensions of Object-Based Language Design", *OOPSLA '87 Proceedings, ACM SIGPLAN Notices*, vol. 22, no. 12, 1987.