Designing Configuration Management Tools for Dynamically Composed Systems

Bradley Schmerl

Department of Computer Science, Clemson University, Clemson, SC 29634, USA

Chris Marlin

School of Informatics and Engineering, Flinders University, GPO Box 2100, Adelaide, SA 5001, Australia

Abstract

Dynamically composed systems (DCSs) are software systems that can be constructed, modified and maintained as they execute. Although this capability has existed in languages and operating systems for some time, it has more recently become popular with the spread of languages such as Java and those supporting mobile code. However, the increasing pervasiveness of dynamically composed systems has not yet resulted in the development of appropriate tools to manage their configurations. Critical problems, such as ascertaining the composition of a DCS or determining the effect of adding a new component to a DCS, in addition to other configuration management issues, have not been addressed by configuration management tools. This paper describes an approach to the design of configuration management tools for dynamically composed systems. The design process involves an analysis of the domain of dynamically composed systems to identify the various problems that are typically addressed by configuration management tools in the static domain. Following this, a model of configuration management for dynamically composed systems is developed. This paper provides an overview of these aspects of the design process, as well as an example of how the model is used to design configuration management tools for dynamically composed systems.

1 Introduction

In traditional software development, changes to software are made by a developer, the software is built and then reshipped to a customer, who decommissions the old version and starts the updated version. With the advent of networks where a software system may be distributed over a wide area (such as via the Internet), an approach to software composition has developed that allows systems to be constructed on the fly, as they execute. Systems utilising languages such as Java [7] and middleware such as OMG’s CORBA [13] and Microsoft’s distributed component object model (DCOM) [2] permit the incremental provision of services to programs. This results in systems that can be updated or changed without having to be stopped.

We classify such systems as dynamically composed systems, and they are examples of systems that can be constructed, modified and maintained as they execute. Although the examples mentioned above are recent, dynamically composed systems have been in use in various domains for some time and are not limited to the construction of programs.

Unfortunately, the recent increase in the pervasiveness of dynamically composed systems has not resulted in the development of software engineering tools that can effectively manage these configurations. If tools are provided, they are typically provided in an ad hoc manner, and reinvent work done on the management of static systems. We contend that many of the management issues not addressed for dynamically composed systems can be addressed by examining the configuration management of static systems. We advocate a methodical approach to the design of such tools, involving a careful analysis of the issues in constructing dynamically composed systems and how they may be addressed by extending or modifying the concepts behind current configuration management tools. This careful analysis results in models that can be used to
delineate the design space for configuration management tools to manage and coordinate the change to software as it executes.

This paper describes a methodology for designing software engineering tools, and the application of that methodology to designing configuration management tools for dynamically composed systems. The design methodology is derived from an approach to designing programming languages (first described in [11]). When applied to tool design, this approach advocates the design of the semantics of tools before their interface. This approach also involves using semantic models to design and compare alternative approaches.

Section 2 of this paper presents an example dynamically composed system, and uses it to discuss the issues we wish to resolve with configuration management tools. We then report the results of our analysis of this domain in the context of configuration management. In Section 3, we present an overview of our semantic modeling language, called the Dynamic Configuration Description Language (DCDL), that allows us to design tools. This design process is described in Section 4. Finally, we present our conclusions and ideas for future work.

2 Domain analysis of dynamically composed systems

As mentioned in the previous section, we wish to examine dynamically composed systems (DCSs) in the context of configuration management (CM) concepts, to determine the configuration management issues extant in a range of dynamically composed systems. This section presents an example of a dynamically composed system to motivate a discussion of the problems, and then outlines how configuration management concepts can aid in resolving these problems. It is beyond the scope of this paper to address in detail either the CM concepts or the model used to more formally examine DCS. For the former, the reader is referred to [18] and for the latter, [15].

2.1 An example dynamically composed system

As observed in Section 1, dynamic composition can occur in a variety of areas. Our overall work is concerned with this wide range; however, for the purposes of this paper, we present an example from one particular domain. The example we use is that of an Internet browser of the kind specified in [7]. The browser is required to display and manipulate data in a way that is dependent on the form of the data. The browser dynamically downloads software components to perform these functions on a particular kind of data.

Figure 1 illustrates a potential browser session in two stages of examining a particular data set. The session involves browsing some information about the weather, and involves several types of data that the browser must be able to view. Consider that the browser session needs the following types of data:

- image maps, one of which appears at the top of Figure 1(a),
- satellite images, two examples of which appear in the Figures 1(a) and (b),
- plain images, like the one showing the average rainfall in Figure 1(b), and
- spreadsheet data, as shown at the bottom of Figure 1(b).

As the browser executes, it encounters data of different sorts. Associated with each data item is a component¹ that can be used by the browser to display (and manipulate) the data in various ways. When the browser encounters the data, it can dynamically include these components. The initial browser (i.e., one not viewing any data) may already include some components that can display a default set of commonly used data types (e.g., plain images).

In Figure 1(a), a user has requested a document to be viewed that contains two different types of information: a map of Australia that allows the user to select a state, thus indicating the user’s desire to display a document containing weather information about that state, and a satellite image of Australia for a particular day. To use the map of Australia, the browser is required to know what to do when a certain state is selected. In our example, we can imagine that the browser downloads a component to perform this particular task, and that the required component is indicated in some way by the data. A similar action occurs for displaying the satellite image. After viewing this document, the configuration of the browser includes components for displaying image maps and satellite images.

On selecting the state of South Australia in the image map at the top of Figure 1(a), the browser is required to display another satellite image, an image showing average rainfall and a spreadsheet that shows rainfall observations and allows user input.

The browser needs to contain components to view all these types of data. It is apparent that the full set of these components cannot be specified before the browser is executed because they are only known at the time that this particular document is viewed. Therefore, the components must be loaded dynamically. At the particular stage of the browsing in Figure 1(b), a number of choices need to have already been made, each of which may be valid in certain circumstances. These decisions concern which viewing component is used for the plain image and the satellite image on the new page, considering that the browser already contains viewing components for these sorts of data. The choice could be to select viewing components already in the browser, or to choose those referred to by the data, or to use some rule (such as to choose the latest one).

¹ In the context of this application, components may be called plug-ins or classes, for example.
Fig. 1. An example Internet browser session.

Another important issue is that of the consistency of the browser as it is being constructed. For example, we may require that every resource required by a viewer (e.g., for displaying in the browser’s window) is supplied by the browser. Furthermore, rather than being able to express general consistency in this way, it may be desirable to require the more browser-specific notion that every sort of data encountered has an associated viewer. For a particular application, there may be any number of sensible specific requirements.

When examining a dynamically composed system such as the one described in this example, it is difficult to find:

- the components that comprise the configuration, and their organisation in terms of a system model,
- whether or not the configuration is consistent,
- how the configuration may be affected by a new component, and
- whether or not it makes sense to incorporate a component.

These questions are typically answered by configuration management tools, but only for static systems. The next section discusses the configuration management concepts we examined, and presents how to provide support for dynamically composed systems in a configuration management tool.

### 2.2 Configuration management

In examining dynamically composed systems in terms of configuration management, we constructed a model of the configuration management concepts of classification, versioning, validation and composition, concepts which were first enumerated in [18].

#### 2.2.1 Classification

Classification can aid in providing information about configuration items by partitioning all configuration items according to their class (such as source code, object code, interface, design documents), as well as describing the attributes of these configuration items (the contents, the name, the author(s)). This partitioning and attribution is used to identify and classify configuration items.

Attribution provides a rich way of describing configuration items [4,8,10]. One of the novel aspects of more recent configuration management systems, such as Jason [18], is the ability of a configuration item to reference other configuration items as attributes. Thus, it becomes possible to encode in a configuration item various ways in which it relates to other configuration items. In fact, it is possible to have container configuration items whose sole purpose is to reference (contain) other configuration items. In this way, for example, subsystems can be formed. Using such containers, a system can be organised into manageable pieces. A collection of configuration items organised in some manner is called a configuration; if it contains all configuration items needed to form a particular software product, it is called a complete configuration (sometimes called a total configuration [18] or bound configuration [20]).

The classification concept of configuration management can therefore aid in answering the questions of what a system comprises, how the system is organised, how various pieces are identified, and also, through attribution,
the part of the language of discourse for selecting and determining the validity of configuration items.

2.2.2 Versioning

Versioning characterises a particular relationship between configuration items. It collects configuration items together that can be treated conceptually as the same. A group of configuration items representing the same concept is called a family [1], or version family [18], or even a version group [17]. Versioning allows these configuration items to be identified as a set, with properties of its own. The typical relationships that are encapsulated within a version family are the historical changes to a particular configuration item, usually called revisions [1], and the configuration items that are variations of each other, commonly referred to as variants [1,17].

Treating version families as identifiable entities in a configuration management system allows the description of a partial configuration. Unlike a complete configuration, a partial configuration is a set of version families, organised in some way. This means that the complexity of configurations is managed by allowing the specification of a configuration generically, in terms of the concepts, rather than particular configuration items.

In the context of dynamically composed systems, versioning can provide a way of determining whether one component is a version of another. This helps address the question of whether it makes sense to incorporate a component or what happens if a new version of a configuration item is added.

2.2.3 Validation

Validation is a way of determining whether a collection of configuration items is consistent in some sense. The language of discourse for arguing about validity consists of the attributes of documents, and some operators for comparing these attributes. Many configuration management systems implicitly define not only the attributes and the operators, but also the meaning of a valid configuration. For example, Make uses the filename and time-modified attributes of files to determine consistency; the language employed by Make is used to give a description of the configuration, which lists dependencies between objects. A configuration, according to Make, is inconsistent if any of the files on which a particular file depends are more recently modified than that file.

ICE [19] uses feature logic [16] to determine consistency; no two members of a configuration can have mutually exclusive features. Jason uses classification and a first-order logic language to allow the explicit definition of validity. Constraints are written in this language, and then configurations are passed to them for validation.

When constructing a dynamically composed system, it is assumed that the system is being composed out of configuration items that have been built from source code. Thus, the dependency aspect of validation is not directly related to dynamic composition. However, the general notion of validity can be used to answer some implicit or unstated questions. For example, the question of whether it “makes sense” to incorporate a component can be rewritten as whether incorporating a component results in a valid configuration. Thus, validation provides a more precise way of saying what “makes sense” when adding a component to a dynamically composed configuration.

2.2.4 Composition

In configuration management, the act of composition, often referred to as building, is the fabrication of a complete configuration from a partial configuration. A build operation:

(1) takes a partial configuration,
(2) uses selection rules and dependency relations to select already existing configuration items, and
(3) if configuration items do not exist, or the configuration items selected result in an inconsistent configuration according to the dependency relation, build actions are invoked that attempt to construct new configuration items (perhaps from source code) to build a complete configuration.

Each of these steps assumes that the entire configuration or subsystem is known.

The areas of concern directly related to dynamic composition are how these rules for selection are expressed (so that, given multiple versions of the same configuration item, the best can be chosen) and how configuration items are added to the configuration.

2.3 Requirements for CM tool support of DCS

The results of analysing dynamically composed systems from a configuration management perspective concluded that dynamically composed systems provided only rudimentary support for configuration management. In particular:

- Validation is typically hardwired, but varies from one dynamically composed system to another. Thus, when designing dynamically composed systems and the tools to support them, there is a decision to be made as to the meaning of a valid configuration.
- The way in which versions are handled is embedded in the hardwired notion of validation. The implicit rules dictating what makes a configuration consistent also deal with changing versions. Dynamically composed systems do not have a sophisticated notion of versions.
- Composition of dynamically composed systems is different to that for static systems. An incremental approach is required, and partially bound configurations need not be treated erroneously, as is currently done with most configuration management tools [14].
Thus, a set of new requirements for configuration management tools were developed. A configuration management tool supporting dynamically composed systems must:

- allow incompleteness,
- validate incomplete configurations, and
- provide incremental composition.

Based on these issues and new requirements, various tool requirements were identified. These requirements were in turn used to derive requirements for a configuration management model. For the purposes of this paper, we will discuss two of the five tool requirements from this research:

**TR2:** A configuration management tool should be able to determine whether a configuration is consistent.

**TR5:** A configuration management tool should give feedback on the components that are sensible to incorporate.

### 3 Deriving a model of configuration management for dynamically composed systems

With the information presented so far, it would have been possible to design and implement tool support for managing configurations of dynamically composed systems in an ad hoc fashion. However, a more methodical approach to the design of configuration management tools was adopted, leading to a more general result than merely the design of a single tool. This methodical approach to design is derived from the approach to programming language design presented in [11] and used since then to design various programming languages or programming language features, such as [3, 5]. The design approach advocates the design of the semantics of a language feature before its syntax, and the use of specific semantic models to make comparisons and to delineate the design space for the domain concerned.

The design of a software engineering tool involves choosing between a range of design alternatives for different aspects of the tool concerned, as does the design of programming languages. Consequently, the design approach mentioned above has been adapted to the design of software engineering tools [12], and this paper describes how it was used to design tools for the configuration management aspect systems supporting of dynamic composition.

The relevant design process involved constructing a model of configuration management for dynamically composed systems that delineates the design space for the corresponding tools. This design space includes the design of appropriate information structures, manipulations of those structures, and interactions between various aspects of the structures and manipulations. By producing a model, rather than a tool, it is anticipated that the knowledge encapsulated in the model can be used to design appropriate tool support for a number of the different domains in which dynamic composition manifests itself, rather than being confined to the contribution of a particular tool for a single domain.

Using the requirements presented at the end of Section 2, this section briefly describes a model, called the Dynamic Configuration Description Language (DCDL), that allows tools to be designed that satisfy the requirements mentioned at the end of the previous section (as well as others, omitted for the sake of brevity). The two tool requirements mentioned produce some requirements for the model of configuration management. Our reasoning is that if a model satisfies the model requirements, which result from the tool requirements, tools designed using the model will satisfy the tool requirements. The model requirements to be discussed are:

**MR4:** Validation must be well-defined for incompleteness.

**MR8:** The classification aspects of the model must interact with a versioning model, as a way of modeling incompleteness.

**MR9:** The model of validation should include a model of validation in the presence of versions.

### 3.1 DCDL - A Dynamic Configuration Description Language

DCDL is based on a configuration language called Jason [18], which covers the four aspects of configuration management described in Section 2, but in the static context only. Like DCDL, it uses an object-oriented notation for classifying objects and first order logic for validation and selection. Both DCDL and Jason are formalised using many-sorted algebras [6].

Figure 2 presents an overview of how versioning is modelled in DCDL. The code in the shadowed boxes represent class and family definitions in DCDL. Surrounding these boxes are indications of how the various DCDL components are modelled using many-sorted algebras. On the left-hand side of the figure, a Viewer is described, which consists of five attributes and one consistency constraint. The class is modelled as a many-sorted signature (denoted \( \Sigma \)), and instances of this class are modelled as \( \Sigma \)-algebras. Each attribute is modelled as an operator, with the types of the operators modelled as functions on sorts. Finally, associated with each class is a set of constraints that define how a valid object of this class can be composed. In the figure, a consistent viewer component is one where a provision and a requirement do not have the same name.

To the right of Figure 2 is a definition of a version family. Version families are used to satisfy MR8, which relates classes and objects to versions. The definition in the figure states that in a given repository (modelled as \( X \)), a version family is the set of components of the class Viewer that satisfy the constraint (\( \text{sort} = \text{Image} \)). A version family is modelled as a model of a basic specification (a \( \Sigma \)-model). The definition on the bottom right of the figure states that the instance set of a version family is all the algebras...
that are instances of a signature ($\Sigma$) in a repository (X) such that the algebra satisfies the set of constraints (K) for the family. To model an entire system, a system model is defined as an algebraic closure— an object and all the objects in the repository reachable from it.

In the model, anywhere that a signature $\Sigma$ appears in a class definition’s operator type set, a $\Sigma$-model may appear in an instance of that class. This allows objects to refer to version families, thus satisfying MR8.

Version families represent one way of allowing incompleteness in a dynamically composed system. Because the constraints associated with classes define whether an instance of the class is valid, and an instance of a class may contain references to version families, the constraints must also evaluate over version families, to satisfy MR9 and partially satisfy MR4.\(^2\)

Figure 3 illustrates an example of how this works. On the left of the figure is an example collection of instances in a repository, and how they are related. Above each instance is the name of the class of which each object is an instance. The viewers attribute refers to a set, of which currently the only member is a version family (the two objects in the grey box) The version family consists of the possible versions for a viewer that can view images (a family of viewers with sort="Image"). The constraint defined on the top right of the figure is the constraint that defines a valid composition of objects of the class Browser. To alleviate the need for users of this system to concern themselves with version families, the underlying model of DCDL defines how a constraint expressed in terms of a total configuration is evaluated in terms of a version family. The constraint at the top states that a valid composition is one where each resource required by a viewer is provided by the composition. In the case of dealing with a version family, we require that the version family be restricted to contain only those versions that will result in a valid configuration. In the figure, the member of the version family at the bottom (which requires a resource named “OpenWindow”) will not result in a valid configuration. Thus, we require that this version family be refined so that this member is discounted from any selection. This is illustrated in the translation of the constraint at the bottom right of the figure. The first line remains the same, the second line is translated into a version for the case that a particular viewer is actually a version family of viewers. If this is the case, then it is refined so that it must satisfy the rest of the constraint (i.e., the formula to the left of the $\triangleleft$ symbol in the figure is added to the set K of constraints for that version family instance).

This translation means that validation can work in the presence of versions, one of the forms of incompleteness allowed in DCDL, thus satisfying MR9. Once all the model requirements have been satisfied, the design space for the tool requirements can be used to provide structure for the design space of a configuration management tool, with DCDL providing the language of discourse for this design space.

### 4 Designing configuration management tools for dynamically composed systems

To design the aspects of a tool corresponding to the tool requirements, DCDL can be used to discuss them in terms of the associated model requirements. These can then be related back to the tool requirement as a design space for that requirement. To discuss the design space relating to tool requirement TR2, the design space for the model requirements MR8 and MR9 can be discussed. By discussing this design space, a set of design choices can be determined that satisfies TR2.
DCDL constraints are associated with classes in a DCDL schema. The formal definition of DCDL prescribes the meaning of a consistent instance of a class as one for which none of the constraints associated with it are unsatisfied. Thus, for every object in a system model, it is possible to determine whether it is consistent. More than that, the formal model says that a complete algebraic closure is consistent if every object in the closure is consistent according to that object’s class constraints. Because complete algebraic closures model system models, this gives a way of determining whether a system model is consistent.

To satisfy MR5, DCDL defines operators that may give results even when an undefined attribute is encountered. A tool facility for determining the consistency of a configuration must obviously provide some way of reporting the result. DCDL specifies that a constraint can return “undefined”, “true” or “false”. For tool design, this means that when consistency is evaluated, only three results are possible. A tool designer should decide which of these is important to report, and when. For example, the question of whether an undefined result should be reported, or only a false one, should be addressed by the tool designer. When consistency is reported is guided by the fact that, if an update occurs, consistency is evaluated at three times (as defined by the formal definition of DCDL): after the update, after a selection, and after the maintained constraints have been evaluated. Thus, the tool designer must consider when it is appropriate to report consistency.

Another issue that arises when reporting consistency is that of whether to report the consistency on a coarse or fine level of granularity. The coarse level would state the consistency of the configuration as a whole; the fine level would report on the basis of individual objects in the system model. This represents a design choice of how to report consistency: design a tool facility that highlights each inconsistent object in the model, or allow the user to find this information in some other way.

DCDL allows the reporting of more than simply whether a configuration is inconsistent; it also allows the determination of which constraint reported the inconsistency. Thus, there is another level of granularity for a tool designer to decide upon: whether to report that a configuration is inconsistent or whether more information should be available. In reporting this inconsistency, a tool designer is faced with the question of how to display which constraint reported inconsistency. Depending on the intended user of the tool, reporting the constraints in terms of their first order logic definitions could be confusing and impenetrable. Thus, a tool designer might use the comment associated with a constraint in a DCDL specification or design some other way of reporting inconsistency. Either way, this needs to be associated with the particular constraint in some way.

Figure 4 shows how a tool might display information about consistency. It indicates the object in a configuration which is inconsistent (surrounded by a dashed box), and displays the constraint which was unsatisfied.

The above discussion considers designing a tool facility where information is presented, but it may also be possible to allow constraints to be defined by the tool facility. In this case, the constraint language of DCDL may be used to define the constraints. This would mean that a tool would need to include a compiler for constraints. Another issue is that constraints need to be carefully designed so that they do not interfere with each other (e.g., express consistency in conflicting ways or attempt to set an attribute to have two different values). Thus, a tool mechanism that reported constraint conflicts would need to be considered. Such a mechanism could identify conflicts, for example, in much the same way as conflicting constraints in version family refinements are identified.

When designing a tool facility for determining whether a configuration is consistent, a tool designer must:
• provide the constraints defining a valid collection of DCDL objects – DCDL allows the expression of these constraints,
• supply a mechanism for evaluating constraints – DCDL defines how this mechanism will behave,
• provide a compiler and conflict resolution mechanisms, if allowing the tool to define constraints – DCDL defines how to construct correct constraints,
• decide how to report results of validation – there are a number of possibilities:
  - which of the three results (“true”, “false”, “undefined”) should be reported,
  - when should the results be reported, and
  - how detailed the report of results should be.
DCDL defines all these cases.

The tool requirement TR5 requires a configuration management tool for dynamically composed systems to provide feedback on the configuration items that it makes sense to incorporate. This relates to the following model requirements.

MR8: The classification aspect of the model must interact with a versioning model, as a way of modelling incompleteness.
MR9: The validation aspects of the model should include a model of validation in the presence of versioning.

In terms of tool design, this implies that the tool facility for presenting versions should be integrated with the tool for presenting configurations, because they are part of the same entity (i.e., a system model). If a tool were to display them separately, it would need to have placeholders representing an incomplete configuration in places where version families occur in the DCDL information structure representing a system model. Figure 5 provides an example of how a tool might display version families in the system model. The grey box represents the fact that what is being referred to is a version family; the components within the grey areas are the members of a version family.

Displaying version families with the configuration leads to a number of possibilities about when version families should be displayed, corresponding to steps in the model that may work on version families (not described in this paper). These steps correspond to three possible views of a configuration involving version families.

1. Let the defining members be those members of a version family satisfying the equations used to define a version family. A version family instance is added to the configuration, and may be displayed prior to validation. Thus, one possible view is to display the version family with all its defining members.
2. Once validation occurs, it is possible that the version family has been refined so that it is consistent in the context in which it is used. The members will form a subset of the defining members of the version family. It is therefore possible for a view to show these members.
3. After selection, a version family will be replaced in the configuration with one of its members. Thus, one view could display only the selected member.

All possible views may be needed at different times: For example, it may be desirable to know from which family a selected configuration item came or which constraint(s) precluded a member of the defining family. Figure 5 shows an example of all three possibilities. The member of the version family in white is the one which has been selected; those in grey are consistent, and those that are crossed out will not result in a consistent configuration. In this view, a designer might allow a user to select a particular member and display the constraint which precluded it, in much the same way as constraints are displayed in Figure 4.

In the context of changes to a configuration, the definition of version families means that, if a new object is added to the repository, then it is added to any version family instances whose constraints it satisfies. This new object might be the correct one to select according to a selection rule (for example, it is the latest version). In terms of design issues, the designer should consider whether to allow this selection to occur automatically, or wait until that component is selected again. To some extent, this
Configuration Edit View Families

Represents a back door method for change to occur. This could be solved by including a notion of workspaces [9] into the DCDL model. A workspace could be a subset of the objects of a main repository, with the semantics that adding a new object to the main repository would not necessarily propagate the addition to the workspace. However, this can be implemented independently of the model by leaving the meaning of a repository to a tool designer.

Another implication of the integration of version families and configurations is that, instead of adding particular objects to a configuration, version families can be added. This is similar to the static case, where a system template is defined as a set of version families rather than a particular object, and selection rules select members from these families to produce total configurations. In a dynamically composed system, a version family can be defined dynamically and added to the system model. This means that the addition of components is more general than is possible in previous dynamically composed systems, which only deal in adding particular components.

DCDL aids the design of a tool facility for versioning dynamically composed systems by:

- defining version families to be part of configurations,
- defining how consistency constraints can refine version families,
- elucidating the steps at which version families could be displayed, and
- providing automatic update of the system by adding a new version, if this is desired.

5 Conclusion and future work

This paper has illustrated how DCDL can be used to design suitable configuration management tools for dynamically composed systems whose requirements are derived from a methodical analysis of dynamically composed systems. The underlying model components (e.g., version families and constraint evaluating over them) have been implemented using C++, and some performance evaluation has been conducted. This technique has also been used to design and implement a prototype tool for dynamically linking object code into an executing program. In this case, the tool acts as an intermediary between the executing program and a dynamic linking library. It is anticipated that this is how a configuration management tool will be implemented in other domains (such as in the context of dynamic linking in programming languages).

Although the facilities designed in this paper are not fully implemented or assessed, the way in which DCDL elucidates various design decisions and provides a framework for being able to discuss various design alternatives for configuration management facilities in tools should now be clear. How DCDL can aid the process of designing tool facilities for the configuration management of dynamically composed systems was illustrated in Section 4. The key to these facilities is that they are not designed in an ad hoc way: the information presented to users is based on the information structures introduced in DCDL, and manipulations of the dynamic composition adhere to strict configuration management principles such as versioning, construction of a valid configuration and specification of selection criteria.

Work building upon the research described in this paper is advancing along three fronts:
(1) Although a prototype implementation of the underlying DCDL concepts has been produced and analysed, the necessary user interfaces discussed in this paper have not been constructed. This is the most pressing area of future work.

(2) DCDL only models the configuration management aspect of tool support for dynamically composed systems. Other areas, such as interaction with specific dynamic linking mechanisms, the dynamic semantics of a system and the transfer of state between one version and another have either not been addressed or have been included in an ad hoc fashion. Future work involves modelling these aspects of a dynamically composed system, and their interaction with DCDL.

(3) Because DCDL has its roots in a model of static configuration management, there is an opportunity to examine the precise interaction between static configuration management and dynamic configuration management. The obvious differences between static and dynamic configuration management are the greater importance of incomplete systems, and of incremental construction, in the latter. More subtle interactions come from how static configuration management can be used to update configurations dynamically.

References


