Software Architecture, Configuration Management, and Configurable Distributed Systems: A Ménage a Trois

André van der Hoek, Dennis Heimbigner, and Alexander L. Wolf

Software Engineering Research Laboratory
Department of Computer Science
University of Colorado
Boulder, CO 80309 USA

{andre,dennis,alw}@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-849-98 January 1998

ABSTRACT

Software architecture, configuration management, and configurable distributed systems are three areas of research that until now have evolved separately. Contributions in each field have focused on their respective area of concern. However, as solutions in the three fields tend to center around some notion of a system model*, it is worthwhile to investigate their relationship in detail. In particular, the large amount of overlap among the system models developed in each area, combined with the complementary nature of the differences among them, suggests that an approach based on a common system model is viable. In this paper, we illustrate the benefits of using such a unified system model, identify the commonalities and differences among the existing system models, and present some of our initial experiments that we believe will lead to the development of a single system model that is usable in all three fields.*

This work was supported in part by the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

1 Introduction

Design, implementation, and deployment are three activities that are normally carried out during the lifetime of a software system. In support of these activities, three distinct software engineering disciplines have emerged: software architecture, configuration management, and configurable distributed systems. *Software architecture* addresses the high-level design of a system. A system design is partitioned into its primary, coarse grain components. These components are then combined into a complete system by explicitly modeled connections. Often, a software architecture description language that formally describes the components and connections is provided. *Configuration management* supports the implementation phase of a software system. Typical solutions manage multiple versions of the source files that implement a system, provide for a selection mechanism to choose a consistent system configuration out of the version space, and subsequently construct the software system out of the selected source files. *Configurable distributed systems* concentrate on managing a system once it is "out in the field". Work focuses on the ability to reconfigure a system after it has been deployed. In particular, component updates need to be administered in such a way that consistency of the deployed system is guaranteed, even in cases where it is required that a system continues executing while the update takes place. Support for this capability is most often provided by specialized programming constructs and system configuration managers.

Until now the three disciplines have largely evolved separately. However, evidence suggests that they are intimately related. A first indication is that the disciplines share a certain amount of terminology. For example, configurable distributed systems and configuration management share the notion of a *configuration* that is composed from multiple parts, software architecture and configurable distributed systems both consider *components* as the level of granularity, and all three disciplines share the goal of maintaining a *consistent* system.

More evidence is provided when we examine some of the contributions in each field and realize that they are crossing the implicit boundaries that have existed among the disciplines. For example, C2 [40], an architecture description language, has recently begun to incorporate dynamism, which until now was addressed exclusively by configurable distributed systems research. As another example, UniCon [36], an architecture description language, includes a build facility similar to the build utilities created by the configuration management discipline. Finally, Darwin, a language to describe configurable distributed systems, has started to incorporate certain features commonly found in ADLs, such as, for example, interface types.

In this paper we investigate the relationship among the three disciplines in detail. We conduct this investigation in terms of a system model, which is an abstraction that describes the structure of a system in terms of its components and the relationships among them. Individual system models have been developed in each of the three disciplines, but here we take a more fundamental approach that is based on the use of a *common* system model. We have three reasons to believe such a unified system model is possible:

- Most, if not all, solutions in the three disciplines are based on an underlying system model.
- Among these solutions, there is a significant amount of overlap among the system models;

many concepts are shared across disciplines.

- Of the non-overlapping parts of the system models used in the various disciplines, most are complementary in nature.

Although we do present some initial experiments that we believe will lead to the development of a unified system model, these are not the main contribution of this paper. The first and foremost goal of this paper is to illustrate the close relationship among software architecture, configuration management, and configurable distributed systems. We present our initial experiments only to illustrate our secondary goal, which is the identification of the unified system model as a promising avenue to advance the state of the art in all three fields.

The rest of this paper is organized as follows. In Section 2 we explain *why* we believe an approach based on a unified system model is desirable. Section 3 explores the system models proposed in the three disciplines and highlights their commonalities and differences. Following that, Section 4 describes our initial experiments in combining some of the capabilities of existing system models. We conclude with some thoughts on future work.

2 Benefits

Before we analyze existing system models and develop our unified system model, we have to demonstrate that such a common system model is indeed desirable. Therefore we present in this section what we believe are the main benefits of using a single system model across the disciplines of software architecture, configuration management, and configurable distributed systems.

2.1 Reduced Modeling Effort

Typically, system models are constructed for each of the activities of design, implementation, and deployment. Because of the existence of separate system modeling languages for each of these activities, a completely new system model has to be constructed from scratch for each of these activities. A significant amount of cost and modeling effort is therefore wasted because common aspects of a system model have to be modeled repeatedly. A unified system model would leverage modeling efforts from early activities when modeling other aspects of a system during later ones. The common aspects of the system model have to be modeled only once, and the complete system model is constructed in an incremental fashion, thereby reducing effort.

2.2 Reduced Architectural Erosion

One of the main problems identified in the software architecture literature is architectural erosion: once the conceptual architecture of a system has been created, it becomes out of date with respect to the actual architecture that is embedded in the implementation of the system [35]. Several corrective approaches have been proposed that rediscover an architecture from an implementation [13, 32], but these solutions provide only a snapshot in time of the mapping between a software architecture and its implementation. If instead the software architecture and configuration management disciplines share a system model, a continuous mapping can be maintained while a system is being implemented and stored in a configuration management repository. In particular, the architecture of a system can be updated with new connections and architectural refinements as they are introduced, thus reducing architectural erosion.

Another kind of architectural erosion takes place once a system has been deployed. Changes made to a system in the field are seldom propagated back to the development site, resulting in a loss of knowledge and a duplication of effort. Especially when similar changes have to be made at multiple field sites, this can account for serious costs to an organization. A system model that is shared between the configuration management and configurable distributed systems disciplines provides an infrastructure to establish communication between the development site and the field sites. Through this communication channel, changes made in the field can be fed back into the development environment. The loss of changes made to a deployed system can thus be avoided, and architectural erosion is reduced.

2.3 Increased Level of Abstraction in Configuration Management

Until now, the system models used in the configuration management discipline have largely been based on source files. Although it is possible to group source files into architectural components, it is often a manual process for which little support is provided. Moreover, such a grouping is often imprecise, since the separation of source files into components cannot be made cleanly. An explicit, architectural system model enhances the configuration management discipline with several important new capabilities. The most important advance is the ability for developers to manipulate versions of architectural components as opposed to versions of individual source files. For example, in a system that has a client-server architecture, developers should be able to "check out" a server component as such or create a new version of a client component. This capability further enchances the understanding of a system under development, as work is carried out at the conceptual level; the level of abstraction is raised from source files to architectural components.

The configuration management discipline is not only concerned with versioning; change impact analysis and system construction are two other areas of interest that could benefit from a rise in abstraction level. In particular, applying architectural dependence analysis techniques [39] on a shared system model enables change impact analysis at the component level as opposed to the source-code level. In addition, better system construction tools can be developed that, in a similar fashion, operate on architecture-level constructs as opposed to source-code directives.

2.4 Automated Support for the Selection of Versions of Components

Most systems are not developed just once, but evolve over time. Bug fixes, new features, and preventive maintenance result in new versions of the components of a system. The disciplines of software architecture and configurable distributed systems both need to be able to select a subset of these components that, when put together, yields a consistent system. However, neither area has a good way of representing versions of components and supporting the selection of valid configurations out of the available version space. Instead, they rely on a user to select components by hand. The configuration management discipline on the other hand has developed sophisticated and automated techniques that support a user in the selection process. Therefore, the disciplines of software architecture and configurable distributed systems would greatly benefit if these configuration management techniques are adopted. Their system models would then contain information that would guide and automate the selection process, even if the version space is large and complex.

2.5 Improved Reuse of Components

Most current configuration management systems store only the source code of a system. Even when the design of a system is versioned as well, little support is available for cross references between the design and its implementation. Given a common system model, a configuration management system can store, version, and provide access to the implementation, design, and configuration information of the components of a software system. Moreover, because the configuration management system understands the underlying system model, various relationships among these artifacts can be established and manipulated. Improved reuse can then be achieved because a design and its implementation are reused jointly, as opposed to the reuse of just a set of source files. As techniques to uncover architectural mismatch mature [17], the compatibility of a reusable component with a set of already selected components can be assessed at a higher level.

3 A Comparison of Existing System Models

The construction of a unified system model should build upon the lessons learned from the system models that have been developed to date. This section therefore compares and contrasts the strengths and weaknesses of the system models that have been developed in each of the disciplines of software architecture, configuration management, and configurable distributed systems. Below, we first introduce an example system that is used throughout the rest of this section to illustrate the various modeling capabilities of each discipline. Following that, we present the system model dimensions that we use for our comparison and highlight, per discipline, the system model support for each of these comparison dimensions. We conclude this section by contrasting the system models and summarizing the discussion.

3.1 Example

Figure 1 presents a simplified version of an existing system that is currently in use to carry out research in the field of numerical analysis [5]. The purpose of the system is to globally optimize a mathematical function, i.e., to find the point in the domain of the function that yields the absolute lowest function value. The system consists of about 15,000 lines of Fortran and C code, and is modularized into a set of components. In the figure, each solid box represents such a component and each solid line indicates the existence of interaction between two components. For example, each `Optimizer` component interacts with a single `ComplexFunction` component. The dashed

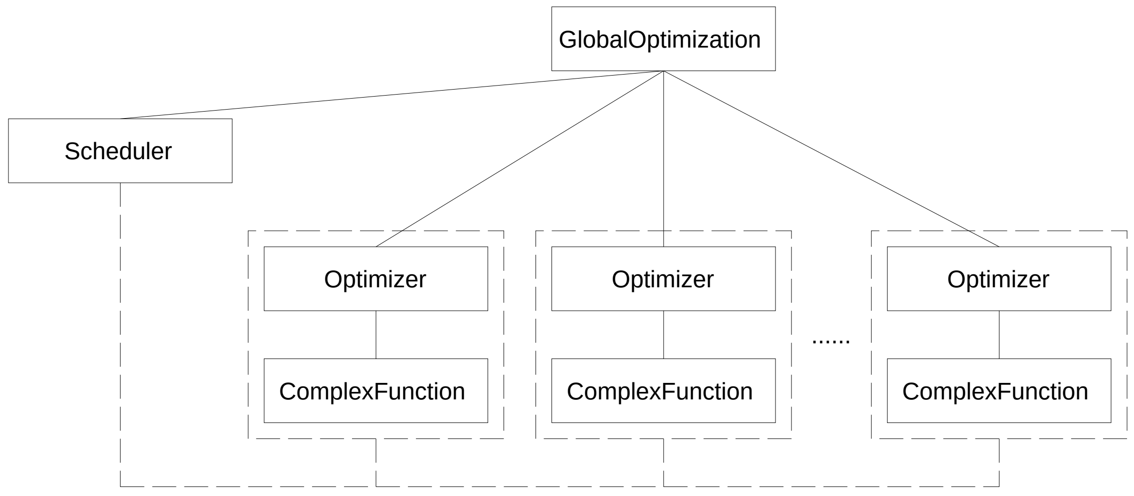

Figure 1: Example System.

lines indicate a different kind of relationship among components: instantiation. As illustrated by the dashed boxes, the `Scheduler` component instantiates new `Optimizer` and `ComplexFunction` components in pairs.

In the system, the `GlobalOptimization` component manages the computation that takes place. It uses the `Scheduler` to create new `Optimizer` and `ComplexFunction` components, and allocates a particular interval of the domain to each `Optimizer` component. The `Optimizer` component carries out an optimization algorithm on the interval it has been allocated, and uses its `ComplexFunction` component to evaluate the function at the particular points that the algorithm requires. The net effect is that the function is optimized by optimizing multiple intervals in parallel.

Throughout its lifetime, the system has been highly variable. Initially, the `ComplexFunction` component consisted of about 3,000 lines of Fortran code that were created at the local site, but it has since been replaced with a separate system that was created at an external site. Also, alternative `Optimization` components exist that each exhibit unique characteristics with respect to the encapsulated optimization algorithm; some are fast but produce less precise results, whereas others are slow but very accurate. Finally, new versions of the `GlobalOptimization` component are created on a regular basis as new approaches are being tried to find better results.

3.2 Comparison Dimensions

The dimensions along which we compare existing system models are all already present in one or more system models. However, the isolation in which the system models of the various disciplines were developed has until now prevented the combination of the dimensions into a single set of requirements for a unified system model.

The dimensions that we have chosen are practical in nature; we focus on expressive capabilities as opposed to more general requirements such as maintainability, reusability, and evolutionary support. In fact, such abstract capabilities are often a direct result of the use of the more expressive capabilities we selected. The following dimensions are used in our comparison.

- *Composition.* What modeling facilities are available to model a system as a set of interconnected components?
- *Consistency.* What modeling facilities are available to enforce consistency when components are combined to form a system?
- *Construction.* What modeling facilities are available to support the construction of an executable system out of its components?
- *Versioning.* What modeling facilities are available to model the existence of, and relationships among, multiple versions of components?
- *Selection.* What modeling facilities are available to support the selection of components from the set of available components?
- *Dynamism.* What modeling facilities are available to model the dynamic changes of a system once it has been deployed?

Below we use these dimensions to compare system models in each of the fields of software architecture, configuration management, and configurable distributed systems. It should be noted that we do not choose a single, representative system model of each discipline as the system model that is compared, but instead match capabilities from multiple system models against the dimensions listed above. Although it is therefore possible that no single system model supports all capabilities listed in the discussion of a discipline, this choice results in a more accurate overview of the contributions of a discipline. It should also be noted that some of the system models address issues in more than one field. However, for the purpose of the discussion below we classify each system model in the discipline it originated.

As a guideline to the rest of this section, Table 1 provides a summary of the discussion that follows. The table ranks the relative support that is provided by the various system models of a discipline for each of the comparison dimensions. The larger the number of bullets that is listed in a category, the better the support that is provided by the system models of a discipline for that particular capability. In the following subsections we substantiate the chosen rankings by illustrating the system models that have been developed by each discipline, and evaluate their support for each of the comparison dimensions.

3.3 Software Architecture

System models in software architecture are captured in architecture description languages (ADLs). At the heart of all ADLs is the ability to model a system out of multiple components.

	Software Architecture	Configuration Management	Configurable Distributed Systems
Composition	●●●	●	●●
Consistency	●●	●	●
Construction	●	●●●	●
Versioning		●●●	●
Selection		●●	
Dynamism	●		●●

Table 1: System Model Capability Comparison Matrix.

In particular, ADLs partition a system into individual components, describe the behavior of each component, and model the interconnections among the components. Figure 2 illustrates this focus of ADLs with an example architecture that is modeled in the Rapide [23] architecture description language. Shown are two components of the example system discussed in Section 3.1, a component that evaluates the function at a particular point in its domain, `ComplexFunction`, and a component that performs an optimization algorithm, `Optimizer`. Each component is modeled with an interface that specifies both the functionality that is provided by the component and the functionality that is expected to be provided by the other components. In Rapide, these functionalities are specified with events. For example, the `ComplexFunction` component is capable of receiving a `Compute` event and producing a `Result` event.

At the architectural level, the components are connected by binding the provided functionality of one component to the required functionality of another component. In our example, the required `Evaluate` functionality of the `Optimizer` component is attached to the `Compute` functionality provided by the `ComplexFunction` component. This implies that when the `Optimizer` component generates a `Compute` event, it is received as an `Evaluate` event by the `ComplexFunction` component. Such explicit modeling of the interconnections among components is one of the distinguishing characteristics of ADLs as compared to other system modeling languages.

Another unique aspect of ADLs is their ability to model the interaction behavior of a component. In Rapide, this is done by specifying the relationship between the events that a component receives and the events that it produces. The behavior of the `ComplexFunction` component, for example, is one where each `Compute` event that is received results in a single `Result` event that is produced. Understanding the interaction behavior of a component is an important capability for ADLs. Combined with the architecture-level connections among components, the specification of the interaction behavior of all components results in a completely specified system on which important analyses can be carried out. For example, the Rapide tool set contains tools that simulate an architecture to uncover such architectural faults as deadlock [22].

We now turn our attention to the comparison dimensions we describe in Section 3.2 and the rankings given to software architecture in Table 1. From the discussion of the Rapide example, it

```
type ComplexFunction is interface
action  in  Compute(Point: Float);
        out Result(Value: Float);
behavior
    NewValue : var Float;
begin
    (?x in Float) Compute(?x) => Result($NewValue);;
end ComplexFunction;

type Optimizer is interface
action  in  FuncValue(Value: Float);
        out Evaluate(Value: Float);
behavior
    Minimum    : var Float := 100000.0;
    StartPoint : var Float := 0.0;
begin
    Start => Evaluate($StartPoint);;
    (?x in Float) FuncValue(?x) => ...
end Optimizer;

architecture GlobalOptimization() return root is
   O : Optimizer;
   F : ComplexFunction;
connect
    (?x in Float) O.Evaluate(?x) => F.Compute(?x);
    (?y in Float) F.Result(?y)   => O.FuncValue(?y);
end GlobalOptimization;
```

Figure 2: Example of an Architectural System Model in Rapide.

should be clear that Rapide is focused on the composition dimension. This focus is shared by other ADLs, which is illustrated by a recent survey of existing ADLs [28]. The survey uses several key characteristics of components, connectors,[1] and configurations as its comparison dimensions. The characteristics chosen by the survey, such as `interface`, `type`, and `constraint`, are all directly geared towards the composition of a system out of its components, and demonstrate the belief of the architecture community that these characteristics are the important ones for system modeling. Further proof of the importance of composition in ADLs is presented by ACME [14], an architecture description language that has been proposed to unify existing ADLs. ACME is centered around the notion of components, connectors, and configurations, which are all system modeling constructs that are used to model the composition of a system.

Consistency is enforced by most ADLs through their strong support for composition. Because components and connections are typed, type checking at system composition time ensures a certain

[1]A connector is a formalized notion of a connection that has its own language constructs in some ADLs.

level of consistency. A stronger, behavioral type of consistency is achieved by Wright [2] and CHAM [17]. Both ADLs formally define architectures. Inconsistencies in an architecture are uncovered by carrying out analyses on its formal definition. Architectural mismatches, such as competing threads of control, have been uncovered this way [7].

The support for the other comparison dimensions besides composition and consistency is rather limited in current ADLs. Versioning and selection are not addressed at all; version control and the selection of components that constitute a configuration both have to be carried out by hand without any guidance from an architectural system model. UniCon [36] seems to be an exception because its system model supports variant implementations of components. However, UniCon does not allow for versions of the actual compositional constructs, such as, for example, components, interfaces, or types; versioning is only supported at the implementation level, not at the architectural level.

Construction has received some attention from the architecture community. One approach, GenVoca [3], is generative in nature: based on an architectural description, a system implementation is generated. Because this approach limits itself to domain-specific applications, it is very powerful. Unfortunately, the underlying generic principles are at too low a level to be considered for inclusion in a unified system model. The other approach, pioneered by UniCon, is based on the existence of a mapping between an architecture and its implementation. Currently, the mapping is limited, since a single component is assumed to be implemented by a single source file.

Our last comparison dimension, dynamism, exists in two forms: external and internal. External dynamism is the ability to dynamically reconfigure a system through some external support environment. Internal dynamism, on the other hand, is the ability to create and destroy components from within the system model. Both external and internal dynamism are present in ADLs. Most notably, C2 [40] supports external dynamism through its ArchShell [27] environment, whereas internal dynamism is supported by Rapide [23] and CHAM [16]. In either case, though, support is limited because the system model itself provides no constructs to support the architectural changes with guidelines and constraints. It is not possible, for example, to specify in the system model what particular topology needs to be maintained while an architecture is being modified. The graph grammar approach developed by Le Métayer [29] addresses this problem and provides a means for constraining the topology of a system. As an inherent part of a system architecture, a coordination component is modeled. Through the use of graph-checking algorithms, this coordination component controls the dynamic evolution of a system.

3.4 Configuration Management

In the past, a variety of system models have been devised in the configuration management discipline. Some of these focus on the construction of a system out of a set of individual source files [6, 12]. Others are concerned with the management of versions and configurations of source files [20, 26]. Only recently, the two have been combined into unified system models that not only address versioning and construction, but also raise the level of abstraction from source files to system-level components [11, 21, 41].

To illustrate the strengths and weaknesses of a typical system model developed by the configuration management discipline, Figure 3 presents a revised version of our optimization example that is modeled in the PCL system modeling language [41]. Compared to the previous version in Figure 2, one additional component (or `family` in PCL terminology) has been introduced: the `FastOptimizer` component carries out an optimization in less time than the regular `Optimizer` component, but sacrifices precision to gain the time benefit. Modeling such variability and providing mechanisms to select an appropriate subset out of the available components are the central foci of system models in the configuration management discipline. In PCL, attributes are used to support the versioning and selection process. Attributes specify key characteristics of a component, and can be used in both a descriptive and a selective manner. In our example, the attributes `precision` and `complexity` are used to precisely describe the difference between the `FastOptimizer` and `Optimizer` components. These attributes have no further influence on the actual composition of the system. The attribute `fast`, on the other hand, is used as a selection criterion between the `FastOptimizer` and `Optimizer` components. Depending on its value, the parts that constitute the `GlobalOptimization` component differ. Modeled here is the version `fast-version` of our system, for which the attribute `fast` is selected to be `true`. Consequently, the `FastOptimizer` and `ComplexFunction` components are selected by the `GlobalOptimization` component to be included in the system.

Not only does Figure 3 illustrate the versioning and selection capabilities of PCL, it also demonstrates the integrated support for the construction of an executable system. To this extent, a mapping is maintained within a PCL system model between its components and their implementation. The `FastOptimizer` component, for example, is implemented by the files `fast.c` and `optimizer.h`. The selection rules are used to determine the set of source files that is needed to construct a particular system version, and the executable system is compiled from this set of source files.[2]

To evaluate the capabilities of the system models that have been developed by the discipline of configuration management, we now extend the discussion to include other system models besides PCL. Typical in all these system models is the rather limited support for the first two comparison dimensions, composition and consistency. Although the hierarchical composition of a component out of multiple parts is supported by most system models, it is only one of the capabilities that is needed. Two key concepts that are missing are explicitly modeled connections and behavioral specifications.

Consistency is only guaranteed for frozen configurations, i.e., versions of systems that have been deemed correct by a user and are permanently labeled as non-modifiable. However, when arbitrary components are selected to be combined in a new configuration, potential inconsistencies are not revealed by the information that is modeled. The typing mechanism of Adele [11] and the interface specifications introduced by Perry [34] provide some improvement, but behavioral consistency cannot be achieved since both are static in nature.

The next three comparison dimensions, construction, versioning, and selection, are at the heart

[2]PCL includes a set of standard, extensible derivation rules that are part of its system modeling capabilities. For brevity, these rules are not presented here.

```
family ComplexFunction
    ...
end

family FastOptimizer
    attributes
       precision  : string = ``0.01'';
       complexity : string = ``n squared'';
    end
    physical
        fastoptimizer => (``fast.c'', ``optimizer.h'');
    end
end

family Optimizer
    attributes
       precision  : string = ``0.00001'';
       complexity : string = ``n cubed'';
    end
    physical
        optimizer => (``precise.c'', ``optimizer.h'');
    end
end

family GlobalOptimization
    attributes
       created-by : string = ``Andre van der Hoek'';
       created    : string = ``97/11/06'';
       fast       : boolean default false;
    end
    parts
        O => if fast then FastOptimizer else Optimizer endif;
        F => ComplexFunction;
    end
    physical
        main => (``main.c'');
        exe  => ``calc.exe'' classifications status := standard.derived; end;
    end
end

version fast-version of GlobalOptimization
   attributes
      fast := true;
   end
end
```

Figure 3: Example of a Configuration Management System Model in PCL.

of configuration management. Advanced techniques and modeling capabilities have been developed over the years [8, 10, 38] of which the example has highlighted the essential contributions. However, two additional concepts deserve special mention.

- *Variants and revisions.* Our example contains two versions of an optimization algorithm: the `Optimizer` component and the `FastOptimizer` component. Although different, these components provide the same functionality and they are therefore termed variants. A different relationship exists when a new version of a component represents a change over time. If, for example, a new version of the `FastOptimizer` component is developed that fixes a problem in the existing version, the new version would be called a revision of the existing one. Both the variant and revision relationship are explicitly modeled; typically, the variants and revisions of a component are organized in a version tree. We believe that these concepts need to be carried over into a unified system model to fully incorporate the essence of configuration management.

- *Change sets.* A rather different approach to modeling system configurations is the change-set approach [37, 42]. As opposed to managing versions of components, change sets model changes as first-class entities. Changes can be simple modifications to a single component, but can also be complex modifications having a system-wide impact. Using change sets, a particular system configuration is selected as a baseline and a set of desired changes. The desired system is then constructed by applying the changes to the baseline. Although the change-set approach is elegant and easily understood by its users, it has the problem that it depends on merging [4], which makes it inherently inconsistent. However, its application to software architecture and configurable distributed systems is appealing because the coarse granularity of the objects of concern could lessen the merging problem considerably.

The last comparison dimension, dynamism, has not been addressed by the configuration management community as of yet. None of the system models are capable of modeling internal dynamism, nor do they provide support for external dynamism; the system models that have been developed all are static in nature.

3.5 Configurable Distributed Systems

The discipline of configurable distributed systems is a relatively new research area. Although a variety of tools and techniques have been developed [19, 25], few formally defined system models exist at this time. Even the ones that do exist [1, 15, 24] do not yet address the full breadth of problems that have been identified.

Perhaps the most advanced system model to emerge from the configurable distributed systems area to date is Darwin [24]. We therefore use Darwin to model the configurable and distributed aspects of our optimization example. Figure 4 illustrates the resulting specification. All four components that are described in Section 3.1 are modeled, but central to this example is the `Scheduler` component. The `Scheduler` component has a `portal` of type `Factory`. Types in Darwin are specified using the keyword `interface`, and consist of a set of opaque members that

```
interface Point {
   x : double;
}

interface Value {
   y : double;
}

interface Factory {
   NewPair;
   IntervalMinimum : Value;
}

component ComplexFunction {
    provide portal Compute : Point;
    require portal Result  : Value;
}

component Optimizer {
    provide portal FuncValue : Value;
    require portal Evaluate  : Point;
    require portal IntervalMinimum : Value;
}

component Scheduler {
    export portal F : Factory @ host("trader");
    bind
       F.NewPair -- dyn Optimizer @ host("machine1");
       F.NewPair -- dyn ComplexFunction @ host("machine2");
       Optimizer.FuncValue -- ComplexFunction.Compute;
       ComplexFunction.Result -- Optimizer.FuncValue;
       Optimizer.IntervalMinimum -- F.IntervalMinimum;
}

component GlobalOptimization(int NumIntervals) {
    provide portal IntervalMinimum;
    require portal NewPair;
    import portal F : Factory @ host("trader");

    forall i = 0 to NumIntervals - 1 {
       bind
          NewPair -- F.NewPair;
          F.IntervalMinimum -- IntervalMinimum;
    }
}
```

Figure 4: Example of a Configurable Distributed Systems System Model in Darwin.

each in turn can potentially be typed. For example, the type `Factory` has two members, `NewPair` and `IntervalMinimum`, of which only `IntervalMinimum` is typed. A `portal` is either a required or a provided functionality of a component, that may or may not be typed. Required functionalities are denoted by `require portal` and `import portal`. Provided functionalities are specified by `provide portal` and `extern portal`. The difference between a `provide portal` and an `extern portal` is that a `provide portal` is only available within a system, whereas an `extern portal` is made publicly available and can be used by other systems. The `portal F` that is provided by the `Scheduler` component is `extern`, and is made available at the host named `trader`. The `Scheduler` component in our system can thus be used by other systems as long as they import, in a fashion similar to the `GlobalOptimization` component, a `portal` of type `Factory` from the host `trader`.

The bodies of components use the statement `bind` to create one or more relationships between `portal` pairs. For example, in the `GlobalOptimization` component, a binding is placed between its required `portal NewPair` and the member `NewPair` of the `import portal F`. The aggregation of all bindings specifies the topology of a system; the bindings in Figure 4 create the topology that is illustrated in Figure 1.

The `Scheduler` component employs a special kind of binding: it dynamically binds a new `Optimizer` component to the member `NewPair` of its `export portal F`. This use of the keyword `dyn` specifies that each time the member `NewPair` of `portal F` is bound by another component, a new `Optimizer` component is instantiated and this new instance is bound to the `NewPair` member. The `GlobalOptimization` component makes use of this facility and dynamically creates, by binding `NumIntervals` times to `F.NewPair`, a set of `Optimizer` components that it can use in parallel to carry out the optimization. In the example, these components are allocated by the `Scheduler` component to particular machines according to a simple, static algorithm: `Optimizer` components are created at one machine, `machine1`, `ComplexFunction` components at another, `machine2`. In more realistic situations, of course, this allocation algorithm could be more complex, since Darwin allows the use of variable, not just constant, machine declarations.

It should be noted that the `GlobalOptimization` component uses the `Scheduler` component in a straightforward manner; it simply creates `NumIntervals` pairs of `Optimizer` and `ComplexFunction` components. The dynamism that is provided by Darwin is not fully exploited. In more complex optimization algorithms, this could change and components could be created and destroyed as needed throughout the whole computation.

Turning our attention to the capabilities described in Section 3.2, we now include in our discussion the other system models that have been developed in the configurable distributed systems discipline. Central to all these system models is the support for composition and dynamism. Similar to the discipline of software architecture, composition is supported through the modeling of components, interfaces, connections, and configurations. Not modeled in any of the system models, however, is the behavior of components and systems, which leads to rather weak support for controlling the consistency of a system. This is especially troublesome in the presence of dynamism; except for type checking, no control can be imposed on changes made to a deployed system. Despite this problem, support for dynamism in the configurable distributed systems discipline is well established. Both internal dynamism, as illustrated in the example, and external dynamism are supported. External dynamism is, for instance, supported by some of the support environments of Darwin [9, 33]. These environments allow a user to dynamically reconfigure a system while it is executing.

The other comparison dimensions are supported in a limited fashion. Only a few attempts to support these dimensions have been made so far. A form of construction is supported by Darwin, but it suffers from the same problem as UniCon: a component is assumed to be implemented by a single source, and thus a single binary, file. Versioning is addressed by the Software Dock [15], but it has only incorporated the concept of temporal revisions until now. Selection has not been addressed at all.

It is important to note that system modeling in the configurable distributed systems discipline is still in its infancy. A variety of techniques are currently being developed that simply have not matured into system model capabilities as of yet. For example, advanced replication [18] and internal reconfiguration [31] are in fact available as individual techniques, but none of the system models incorporate primitives that capture these notions.

3.6 Comparison Summary

To summarize our comparison, we revisit Table 1. Looking at the support provided by each discipline for the comparison dimensions, we see that each discipline has its own focus: software architecture mostly manages composition and consistency, configuration management addresses construction, versioning, and selection, and configurable distributed systems focuses on dynamism. In addition, we observe that a non-trivial amount of overlap occurs. We believe that this overlap is due to the fact that some systems have indeed started to cross the artificial boundaries that have existed among the various disciplines. UniCon and the Software Dock are two prime examples; both systems have incorporated aspects from system models outside of their native category. In fact, Darwin and C2 are even stronger examples since both systems could have easily been placed in two categories, software architecture and configurable distributed systems.

Although none of these approaches has been comprehensive in covering all the dimensions of system modeling, they do provide convincing evidence that a unified system model is an achievable goal. Their mere existence, combined with the complementary nature of the system modeling strengths of the three disciplines, has led us to start developing such a unified system model.

4 Initial Approach

We believe that a unified system cannot be developed until the interactions among the various system modeling dimensions can be fully understood. We are therefore carrying out a series of experiments in which we examine the ramifications of combining certain system modeling capabilities. In this section, we present two such experiments. We have examined how versioning and selection capabilities could be added to Rapide [23]. Below, we illustrate how such an addition can be achieved by integrating two different techniques with the language. We first illustrate how PCL attributes can be used to model and select variants of components. We subsequently illustrate how architectural changes can be modeled using change-set technology.

4.1 Architectural Versioning and Selection through Attributes

A straightforward way of adding versioning and selection to Rapide is to adopt the attribute mechanism used by PCL [41]. In Figure 5 we have remodeled the optimization example of Figure 3 in Rapide while retaining the attributes of PCL. For each component, its required and provided functionalities, its behavior, and a set of descriptive attributes are modeled. The functionalities and behavior are modeled in the original Rapide language, and the attributes in the PCL extensions.

Because component attributes are descriptive, these attributes are independent of the other concepts and the impact on the modeling of individual components is minimal. The impact of attributes on the modeling of the overall architecture, however, is more intricate. At the architectural level, attributes are used to select components, not to describe them. Consequently, we have to introduce the `parts` construct together with the conditional that is based on the value of the attribute `fast`. The conditional is used to differentiate between the instance of the system that has a fast optimization component and the instance that has a slow optimization component.

In the case of the simple variability of the modeled system, the extension of Rapide with the `parts` construct and conditionals is sufficient. However, if the behavior, the required functionality, or provided functionality differs between two variant components, the connections of the architecture have to change depending on the component selection that is made. Thus, the connections made in the `connect` part of the system specification have to be based on conditionals as well, which makes the specification more complex and less clear. Traditional configuration management system models do not have this complexity problem because a system is specified only as a hierarchy of components; other relationships among components are not modeled. Therefore, in configuration management attributes have no impact on a system model other than selecting versions and variants of components.

This example illustrates that, although it seems at first sight simple and straightforward to adopt the attribute-based versioning and selection of PCL in Rapide, it is in reality more difficult. Hidden issues arise to which close attention must be paid.

4.2 Architectural Versioning and Selection through Change Sets

Although there certainly exists a place for attribute-based versioning and selection in a unified system model, a system specification that is solely based on attributes can become complex and unwieldy. Especially in the presence of a large amount of variability, or when components are replaced with architectures in their own right, additional techniques to manage the version space are needed. Change set technology, as described in Section 3.4, provides such a complementary technique.

Figure 6 illustrates the application of change sets to a Rapide architecture. Assuming the system baseline is the architecture that is presented in Figure 2, Figure 6 depicts two change sets that are constructed from it. The first one, `NewEval`, replaces the existing `ComplexFunction` component with a `NewComplexFunction` component. The second, `Fast`, replaces the `Optimizer` component

```
type ComplexFunction is interface
   ...
end ComplexFunction;

type Optimizer is interface
action in  FuncValue(Value: Float);
       out Evaluate(Value: Float);
attributes
   precision  : string = ``0.00001'';
   complexity : string = ``n cubed'';
behavior
    ...
end Optimizer;

type FastOptimizer is interface
action in  FuncValue(Value: Float);
       out Evaluate(Value: Float);
attributes
   precision  : string = ``0.01'';
   complexity : string = ``n squared'';
behavior
    ...
end Optimizer;

architecture GlobalOptimization() return root is
attributes
   created-by : string = ``Andre van der Hoek'';
   created    : string = ``97/11/06'';
   fast       : boolean default false;
parts
   O => if fast then FastOptimizer else Optimizer endif;
   F => ComplexFunction;
connect
   (?x in Float) O.Evaluate(?x) => F.Compute(?x);
   (?y in Float) F.Result(?y)   => O.FuncValue(?y);
end GlobalOptimization;

version fast-version of GlobalOptimization
   attributes
      fast := true;
   end
end
```

Figure 5: Versioning and Selection in Rapide through Attributes.

```
change set NewEval {
   base
      baseline
   contents
      replace 1 -- 8 {
         type NewComplexFunction is interface
         action in  InitEval;
                in  Compute(Point: Float);
                out Result(Value: Float);
         behavior
            NewValue : var Float;
         begin
            InitEval => $NewValue := 0.0;;
            (?x in Float) Compute(?x) => Result($NewValue);;
         end NewComplexFunction;
      }
      add 11 {
         out InitEval;
      }
      add 16 {
         Start => InitEval;
      }
      replace 23 -- 23 {
         F : NewComplexFunction;
      }
      add 24 {
         O.InitEval => F.InitEval;
      }
}

change set Fast {
   base
      baseline
   contents
      replace 10 -- 19 {
         type FastOptimizer is interface
            ...
         end FastOptimizer;
      }
      replace 22 -- 22 {
         O : FastOptimizer;
      }
}
```

Figure 6: Versioning and Selection in Rapide through Change Sets.

with a `FastOptimizer` component. Both change sets represent logical changes to the system, but these changes have to be physically captured. Although other approaches are possible that are based on the structure of the language, the most common approach is to represent the changes as additions, deletions, and substitutions of source lines. The change set `Fast`, for example, replaces lines 10 through 19 and line 22 with new ones. Similarly, the change set `NewEval` adds new lines and replaces existing ones. This mechanism allows for complicated changes to be represented as a single entity. For example, the `NewComplexFunction` component requires an initialization, which is easily incorporated in the change set as the addition of extra lines to the base architecture. Change sets are thus capable of capturing coordinated changes to an architecture.

A particular version of a system is selected by specifying a baseline and the change sets that should be applied to the baseline. For example,

$$\textit{system} = \textit{baseline} + \textit{NewEval} + \textit{Fast;}$$

incorporates both the change sets `NewEval` and `Fast` into the system, whereas

$$\textit{system} = \textit{baseline} + \textit{Fast;}$$

incorporates only the change set `Fast`. The system that is selected is constructed by a merging algorithm that combines all change sets with the baseline. It is possible that conflicts arise while merging. For example, the addition of lines 11 and 16 in the `NewEval` change set is in conflict with the replacement of lines 10 through 19 in the change set `Fast`. If both change sets are selected to be part of a system, the conflict will need to be resolved. Although most merging algorithms are capable of uncovering a conflict, they require manual assistance in resolving the conflict [4]. For the management of source code by configuration management systems that are based on change-set technology, this has not been a problem because conflicts arise only occasionally [37]. It remains to be seen whether this also holds for unified system models or whether it turns out that conflicts arise too often for change sets to be practical in this domain.

In the example of Figure 6, both change sets are based on the baseline of the system. As more and more change sets are developed, it cannot be expected that each is developed from the baseline. Therefore, change sets cannot be based solely on the baseline of a system, but also on arbitrary combinations of the baseline and a collection of change sets. Techniques have been developed [30] that keep track of the dependencies among change sets and are therefore capable of assuring the inclusion of the correct change sets when a particular system is selected. Additionally, a new baseline can be designated that consists of the old baseline combined with a collection of change sets. New change sets can then be constructed that are based on this new baseline.

Two additional aspects of change sets need to be highlighted. First, change sets do not need to be constructed by hand. Almost all configuration management systems incorporate tools that can calculate the difference between a new and an old specification. This difference is in essence the change set. Second, change sets are beneficial to the discipline of configurable distributed systems as well; the ability to deploy incremental changes as opposed to complete new versions of systems is obviously an attractive notion.

5 Conclusions

Until now, the disciplines of software architecture, configuration management, and configurable distributed systems have largely existed in isolation. However, the complementary nature of these disciplines leads to opportunities for cross fertilization. In this paper we have examined these opportunities in more detail. In particular, we have investigated an approach that advocates the joint evolution of the three disciplines: a single, unified system model that is shared by all three fields. As argued, it is our belief that such a unified system model is a very appealing alternative to advance the state of the art in all three fields. This belief is fueled by our detailed comparison, which has identified the complementary strengths of each field. Furthermore, our initial experiments have shown how one can actually combine some of these capabilities into a single model.

Besides reducing the modeling effort throughout the lifetime of a system, a unified system model reduces architectural erosion, increases the level of abstraction in configuration management, facilitates automated selection of versions of components, and improves reuse of components. It is these benefits that drive our efforts to develop a unified system model. To advance our research towards this goal, we are planning on continuing to experiment with various combinations of capabilities to further understand the detailed relationships among them. In particular, we believe it is necessary to first understand the intricacies of the interactions among the dimensions of dynamism, selection, and architectural behavior. Once we have a clear understanding of these, we believe that the development of a complete unified system model can be pursued.

6 Acknowledgements

We thank Jacky Estublier for his contributions to initial discussions on the topic, Richard S. Hall and Nenad Medvidovic for ongoing conversations on the subject, and Naranker Dulay for his help in modeling the Darwin example.

REFERENCES

[1] B. Agnew, C. Hofmeister, and J. Purtilo. Planning for Change: A Reconfiguration Language for Distributed Systems. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, pages 15–22, Los Alamitos, California, March 1994. IEEE Computer Society Press.

[2] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[3] D. Batory and B.J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, 23(2):67–82, February 1997.

[4] J. Buffenbarger. Syntactic Software Merging. In *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*, number 1005 in Lecture Notes in Computer Science, pages 153–172, New York, New York, 1995. Springer-Verlag.

[5] R.H. Byrd, E. Eskow, A. van der Hoek, R.B. Schnabel, and K.P.B. Oldenkamp. A Parallel Global Optimization Method for Solving Molecular Cluster and Polymer Conformation Problems. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 72–77. SIAM, 1995.

[6] G.M. Clemm. The Odin Specification Language. In *Proceedings of the International Workshop on Software Versioning and Configuration Control*, pages 144–158, 1988.

[7] D. Compare, P. Inverardi, and A.L. Wolf. Uncovering Architectural Mismatch in Dynamic Behavior. *Science of Computer Programming*, 1998. To appear.

[8] R. Conradi, editor. *Proceedings of the Seventh International Workshop on Software Configuration Management*, number 1235 in Lecture Notes in Computer Science, New York, New York, May 1997. Springer-Verlag.

[9] S. Crane and K. Twidle. Constructing Distributed Unix Utilities in Regis. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, pages 183–189, Los Alamitos, California, March 1994. IEEE Computer Society Press.

[10] J. Estublier, editor. *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*, number 1005 in Lecture Notes in Computer Science, New York, New York, 1995. Springer-Verlag.

[11] J. Estublier and R. Casallas. The Adele Configuration Manager. In W. Tichy, editor, *Configuration Management*, number 2 in Trends in Software, pages 99–134. Wiley, London, Great Britain, 1994.

[12] S.I. Feldman. MAKE — A Program for Maintaining Computer Programs. *Software—Practice and Experience*, (9):252–265, April 1979.

[13] H. Gall, M. Jazayeri, R. Klösch, W. Lugmayr, and G. Trausmuth. Architecture Recovery in ARES. In L. Vidal, A. Finkelstein, G. Spanoudakis, and A.L. Wolf, editors, *Joint Proceedings of the SIGSOFT '96 Workshops*, pages 111–115, New York, New York, 1996. ACM Press.

[14] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON '97*. IBM Center for Advanced Studies, November 1997.

[15] R.S. Hall, D.M. Heimbigner, A. van der Hoek, and A.L. Wolf. An Architecture for Post-Development Configuration Management in a Wide-Area Network. In *Proceedings of the 1997 International Conference on Distributed Computing Systems*, pages 269–278. IEEE Computer Society, May 1997.

[16] P. Inverardi and A.L. Wolf. Formal Specification and Analysis of Software Architectures using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[17] P. Inverardi, A.L. Wolf, and D. Yankelevich. Checking Assumptions in Component Dynamics at the Architectural Level. In *Proceedings of the Second International Conference on Coordination Models and Languages*, number 1282 in Lecture Notes in Computer Science, pages 46–63, New York, New York, September 1997. Springer-Verlag.

[18] C.T. Karamanolis and J.N. Magee. A Replication Protocol to Support Dynamically Configurable Groups of Servers. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 161–168, Los Alamitos, California, May 1996. IEEE Computer Society Press.

[19] J. Kramer and J. Purtilo, editors. *Proceedings of the Second International Workshop on Configurable Distributed Systems*, Los Alamitos, California, March 1994. IEEE Computer Society Press.

[20] D.B. Leblang, R.P. Chase, Jr., and H. Spilke. Increasing Productivity with a Parallel Configuration Manager. In *Proceedings of the International Workshop on Software Versioning and Configuration Control*, pages 144–158, 1988.

[21] Y.-J. Lin and S.P. Reiss. Configuration Management with Logical Structures. In *Proceedings of the 18th International Conference on Software Engineering*, pages 298–307. Association for Computer Machinery, March 1996.

[22] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[23] D.C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[24] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the Fifth European Software Engineering Conference*, number 989 in Lecture Notes in Computer Science, pages 137–153, New York, New York, September 1995. Springer-Verlag.

[25] J.N. Magee and K. Schwan, editors. *Proceedings of the Third International Conference on Configurable Distributed Systems*, Los Alamitos, California, May 1996. IEEE Computer Society Press.

[26] K. Marzullo and D. Wiebe. A Software System Modelling Facility. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* ACM SIGSOFT, April 1984.

[27] N. Medvidovic. ADLs and Dynamic Architecture Changes. In L. Vidal, A. Finkelstein, G. Spanoudakis, and A.L. Wolf, editors, *Joint Proceedings of the SIGSOFT '96 Workshops*, pages 24–27, New York, New York, 1996. ACM Press.

[28] N. Medvidovic and R.N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference*, number 1301 in Lecture Notes in Computer Science, pages 60–76, New York, New York, September 1997. Springer-Verlag.

[29] D. Le Métayer. Software Architecture Styles as Graph Grammers. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, SIGSOFT Software Engineering Notes, pages 15–23. Association for Computer Machinery, November 1996.

[30] J. Micallef and G.M. Clemm. The Asgard System: Activity-Based Configuration Management. In *Proceedings of the Sixth International Workshop on Software Configuration Management*, number 1167 in Lecture Notes in Computer Science, pages 175–186, New York, New York, 1996. Springer-Verlag.

[31] N.H. Minsky, V. Ungureanu, W. Wang, and J. Zhang. Building Reconfiguration Primitives into the Law of a System. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 89–97, Los Alamitos, California, May 1996. IEEE Computer Society Press.

[32] G.C. Murphy. Architecture for Evolution. In L. Vidal, A. Finkelstein, G. Spanoudakis, and A.L. Wolf, editors, *Joint Proceedings of the SIGSOFT '96 Workshops*, pages 83–86, New York, New York, 1996. ACM Press.

[33] K. Ng, J. Kramer, and J. Magee. A CASE Tool for Software Architecture Design. *Journal of Automated Software Engineering*, 3(3/4):261–284, August 1996.

[34] D.E. Perry. System Compositions and Shared Dependencies. In *Proceedings of the Sixth International Workshop on Software Configuration Management*, number 1167 in Lecture Notes in Computer Science, pages 139–153, New York, New York, 1996. Springer-Verlag.

[35] D.E. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[36] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[37] Software Maintenance & Development Systems, Inc, Concord, Massachusetts. *Aide de Camp Product Overview*, September 1994.

[38] I. Sommerville, editor. *Proceedings of the Sixth International Workshop on Software Configuration Management*, number 1167 in Lecture Notes in Computer Science, New York, New York, March 1996. Springer-Verlag.

[39] J.A. Stafford, D.J. Richardson, and A.L. Wolf. Chaining: A Software Architecture Dependence Analysis Technique. Technical Report CU–CS–845–97, Department of Computer Science, University of Colorado, Boulder, Colorado, September 1997.

[40] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead, Jr., J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *ACM Transactions on Software Engineering and Methodology*, 22(6):390–406, June 1996.

[41] E. Tryggeseth, B. Gulla, and R. Conradi. Modelling Systems with Variability using the PROTEUS Configuration Language. In *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*, number 1005 in Lecture Notes in Computer Science, pages 216–240, New York, New York, 1995. Springer-Verlag.

[42] D. Wiborg Weber. Change Sets Versus Change Packages: Comparing Implementations of Change-Based SCM. In *Proceedings of the Seventh International Workshop on Software Configuration Management*, number 1235 in Lecture Notes in Computer Science, pages 25–35, New York, New York, 1997. Springer-Verlag.