Investigating the Applicability of Architecture Description in Configuration Management and Software Deployment

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-862-98 September 1998

ABSTRACT

The discipline of software architecture has traditionally been concerned with high-level design. In particular, a variety of architecture description languages have been developed that are used to precisely capture a design. Additionally, analysis tools have been constructed that are used to verify particular properties of a design. However, today's trend towards the development of component-based software seems to suggest a new use of software architecture. Because an architecture captures components and the connections among them, it could potentially be used as an organizing abstraction for many of the activities in the software life cycle. In this paper we present an initial investigation into the feasibility of such use. In particular, we closely examine the role system modeling plays in the fields of configuration management and software deployment, and relate this role to the system modeling capabilities of architecture description languages. The outcome of this investigation is twofold. First, we conclude that, for the specific cases of configuration management and software deployment, the use of software architecture brings opportunities to significantly extend the provided functionality. Second, we present requirements for a number of extensions to typical architecture description languages that are needed to make our approach viable.

This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract Numbers F30602-94-C-0253 and F30602-98-2-0163. 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

Perhaps the most ubiquitous trend in today's software market is the move to component-based software development. Middleware technologies, new journals solely dedicated to the topic, an increased attention by researchers, and a growing market for standard software components are just a few of the examples that illustrate this trend.

Software architecture is an emerging discipline that is concerned with component-based software development. In particular, the discipline has developed a number of architecture description languages (ADLs) in which the high-level design of a system can be precisely captured as a set of components and interconnections. A large variety of languages have been created, and each language has its particular strengths and weaknesses. Besides being able to model the components and interconnections of a system, some languages allow, for example, the modeling of the interaction behavior among components [1, 6]. Others permit the modeling of constraints [17].

Additional research in the discipline of software architecture has been concerned with the verification of particular properties of an architecture once it has been created. Methods exist that, for example, can verify whether an architecture is free of deadlock [1] or whether an architecture eventually reaches a certain desired state [4]. Other ADLs lend themselves to the detection of inconsistencies among components that have been put together in an architecture. Architectural mismatches, such as competing threads of control, have been uncovered this way [6].

As we can see from this brief discussion, the focus of the discipline of software architecture has mainly been on the issues that arise during the design of component-based software systems. But, because an architecture precisely captures a design and, in addition, supports analyses of the design, it seems to offer a unique opportunity to serve as the basis for constructing tools and environments that support a component-based software development process. In particular, we envision a software development process such as the one that is illustrated in Figure 1. Shown are five typical phases in the software life cycle, as well as the artifacts that are normally created in each phase. However, as opposed to each phase operating in a different abstraction, we believe that a component-based process requires the use of a single abstraction across the life cycle. Moreover, it is our belief that, as shown in the figure, software architecture is a potential candidate to be this abstraction.[1]

Before committing to this approach and creating architecture-based software development tools and environments—a rather large undertaking—, it is necessary to first investigate the feasibility of the approach. In this paper we present this investigation. In particular, we examine the candidacy of software architecture by comparing the modeling capabilities that it provides to the required modeling capabilities of two activities in the software life cycle, namely configuration management and software deployment. Based on this comparison, we answer the following two questions:

- *Can software architecture be used as an organizing abstraction for activities in the software life cycle and, if so, what are some of the benefits that arise?*
- *Is software architecture by itself sufficient to support activities in the software life cycle, or do its modeling capabilities need to be enhanced?*

Although the answers to these questions are only valid for the specific activities of configuration management and software deployment, we believe that they generalize to other activities in the software life cycle as well. Certainly, more research is required to confirm this belief. However,

[1]Notice that in the analysis phase the architecture of a system is not necessarily available yet, hence the lighter shading.

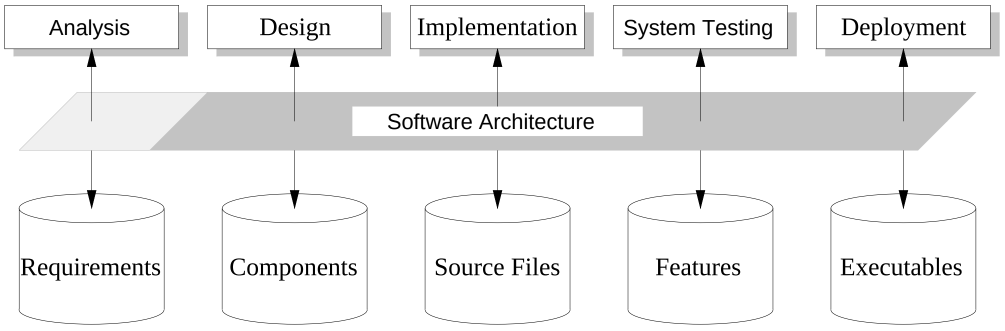

Figure 1: Software Architecture as an Organizing Abstraction.

the results presented in this paper should give us an indication as to whether a component-based software development process that is centered around the notion of software architecture is at all possible, as well as what kind of effort will be required to adapt current architecture description languages for this purpose.

The remainder of this paper is organized as follows. We first discuss, in Section 2, an example system that is used throughout the remainder of this paper as the canonical system to be modeled. In Section 3 we introduce the dimensions along which we carry out the comparison. Sections 4, 5, and 6 discuss, in terms of the example system and comparison dimensions, the modeling capabilities of the disciplines of software architecture, configuration management, and software deployment. In Section 7 we draw our conclusions from the comparison and present the answers to the above questions. We end in Section 8 with an outlook at the future work that remains to be done.

2 EXAMPLE

Figure 2 presents a simplified version of an existing system that is currently in use to carry out research in the field of numerical analysis [3]. 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 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, CHARMM, that was created at an external site.

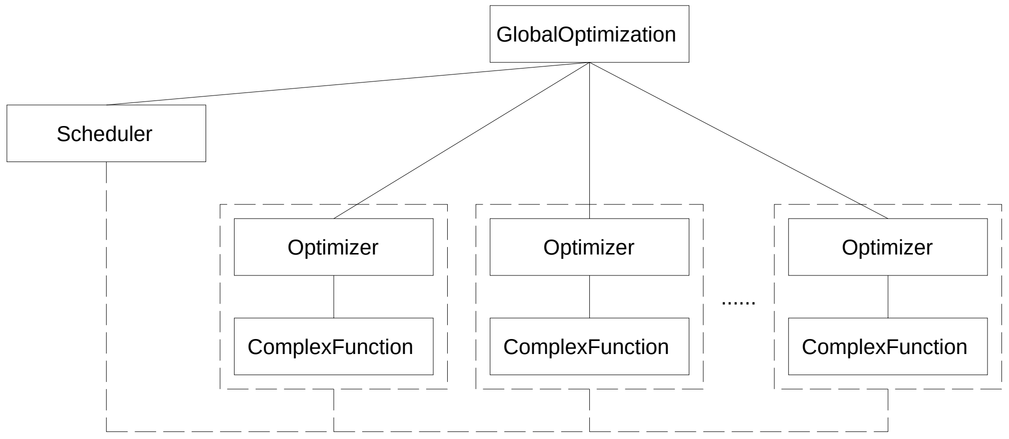

Figure 2: Example System.

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 COMPARISON DIMENSIONS

To compare the modeling capabilities that are provided by the discipline of software architecture to those that are required by the disciplines of configuration management and software deployment, we need a set of comparison dimensions. As the source of this set, we introduce the notion 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. All three disciplines employ the notion of a system model. In software architecture, system models are created that capture the design of systems, whereas in configuration management and software deployment system models are created to drive the respective configuration and deployment activities. It is from these existing system models that we draw our comparison dimensions.

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?

	Software Architecture	Configuration Management	Software Deployment
Composition	● ● ●	●	●●
Consistency	● ● ●	●●	●
Construction	●	● ● ●	
Versioning	●	● ● ●	●●
Selection		●●	●●
Dynamism	●●		

Table 1: System Model Capability Comparison Matrix.

- *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 software deployment. 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.

As a guide 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. It should be noted, though, that the relative number of bullets for a comparison dimension does not necessarily imply that the functionality of one discipline subsumes the functionality of another; the presence of one bullet could indicate the availability of a particular modeling capability that is not provided by another discipline that is ranked with three bullets.

In the following sections we substantiate the chosen rankings by illustrating the system models that have been developed by each discipline and evaluating their support for each of the comparison dimensions.

4 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. In particular, ADLs partition a system into individual components, describe the behavior of each component, and model the interconnections among the components. Figure 3 illustrates this focus of ADLs with an example architecture that is modeled in the Rapide [17] architecture description language. Shown are two components of the example system discussed in Section 2, 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 an `Evaluate` event, it is received as a `Compute` 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 relationships among 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 [16].

We now turn our attention to the comparison dimensions that we described in Section 3 and the rankings given to software architecture in Table 1. From the discussion of the Rapide example, it 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 [20]. The survey uses several key characteristics of components, connectors,[2] 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 [11], 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 level of consistency. A stronger, behavioral type of consistency is achieved by Wright [1] and CHAM [6]. 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 in, for example, CHAM.

The support for the other comparison dimensions besides composition and consistency is rather limited in current ADLs. Version control and the selection of components that constitute a configuration mostly have to be carried out by hand without any guidance from an architectural system model. UniCon [24] is an exception. Although it does not allow for versions of the actual compositional constructs (such as, for example, components, interfaces, or types), its system model does support variant implementations of components.

Construction has received some attention from the architecture community. One approach,

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

```
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 3: Example of an Architectural System Model in Rapide.

GenVoca [2], 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. However, because of its domain specificity, it would be rather difficult to use it to support the generic purposes of configuration management and software deployment. 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 because 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 supports external dynamism through its ArchShell [19] environment, whereas internal dynamism is supported by Rapide [17] and CHAM [14]. 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 [21] 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.

5 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 [5, 10]. Others are concerned with the management of versions and configurations of source files [15, 18]. 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 [9, 29].

To illustrate the strengths and weaknesses of a typical system model developed by the configuration management discipline, Figure 4 presents a revised version of our optimization example that is modeled in the PCL system modeling language [29]. Compared to the previous version in Figure 3, 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.

```
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 4: Example of a Configuration Management System Model in PCL.

Not only does Figure 4 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 are needed to construct a particular system version, and the executable system is compiled from this set of source files.[3]

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 [9] and the interface specifications introduced by Perry [23] 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 of configuration management. Advanced techniques and modeling capabilities have been developed over the years [7] 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.

- *Change sets.* A rather different approach to modeling system configurations is the change-set approach [25]. 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, which makes it inherently inconsistent.

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 dy-

[3]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.

namism, nor do they provide support for external dynamism; the system models that have been developed are all static in nature.

6 SOFTWARE DEPLOYMENT

In contrast to the disciplines of software architecture and configuration management, systems models in the field of software deployment tend to be declarative in nature; a schema is used to describe deployment information about a software system [8, 13, 27]. Based on the information in the schema, activities such as release, install, configure, update, and deinstall are supported by a deployment tool.

One of the more advanced system models that has been developed to support software deployment is incorporated in the Software Dock framework [12]. Figure 5 illustrates a part of our global optimization example as it would be described in a Software Dock schema. Modeled are two components, the `Optimizer` component and the `ComplexFunction` component.

The description of the global optimization system is partitioned into five sections, namely its properties, its composition, its assertions, its dependencies, and its constituting artifacts.[4] In the first part of the description, system properties are declared. In particular, we model the abstract components, `Optimizer` and `ComplexFunction`, as well as the specific instances of these components, namely `Fast`, `Precise`, `CHARMM`, and `OurOwnFunction`, as properties. Additionally, the version of the overall system is defined to be a property. The values of all these properties are not specified in this section. Instead, they are resolved when a deployment activity attempts to satisfy the constraints stated in the remainder of the specification with the user preferences that it takes as input.

The second part, which describes the composition of the global optimization system out of its components, demonstrates the use of constraints. In the first two rules, it is stated that the global optimization system has two parts to be deployed, namely the `Optimizer` and `ComplexFunction` components. The next two rules specify that, although each of these abstract parts is instantiated by two variant implementations, only one variant of each can be deployed at the time. The last two rules indicate some constraints between properties; the `Fast` component is not compatible with the `CHARMM` component, and the `CHARMM` component can only be deployed to the `Solaris` operating system.

In the next two parts, requirements for the system to be properly deployed are specified. Assertions state requirements that simply have to be true and can only be verified (e.g., the operating system has to be `Solaris` or `SGI-MIPS`). Dependencies, on the other hand, specify requirements that can be resolved. In particular, the dependency that is specified states that if the `CHARMM` component is to be part of the system, but its version 4.5 is not installed yet, this version can obtained by activating the deployment process at the given URL.

The final part of the specification is the target of the selection process that is governed by the rules and properties in the previous four sections. It provides a mapping from the abstract properties to the artifacts that actually constitute the system. Depending on the desired version of the global optimization system, on the target operating system, and on the target hardware platform, different executables—each stemming from a different source location at the producer site—are deployed to a target directory at the consumer site.

Abstracting away from the specifics of the Software Dock schema, we now evaluate the capa-

[4]In reality, a sixth part is used to describe the specifics of deployment activities that are not conform the standard processes supported. For brevity, this part is omitted from the discussion.

RegFamily:
```
GlobalOptimization
```

RegId:
```
Name:  GlobalOptimization
Producer:  University of Colorado
```

Properties:
```
P1:  Optimizer[boolean]
P2:  Fast[boolean]
P3:  Precise[boolean]
P4:  ComplexFunction[boolean]
P5:  CHARMM[boolean]
P6:  OurOwnFunction[boolean]
P7:  Version[string]
```

Composition:
```
C1:  true includes Optimizer
C2:  true includes ComplexFunction
C3:  Optimizer oneof(Fast, Precise)
C4:  ComplexFunction oneof(CHARMM, OurOwnFunction)
C5:  Fast excludes CHARMM
C6:  (OS != "Solaris") excludes CHARMM
```

Assertions:
```
A1:  (Arch == "sparc") || (Arch == "i86pc")
A2:  (OS == "Solaris") || (OS == "SGI-MIPS")
A3:  (OS == "SGI-MIPS") => (Memory >= 24)
A4:  (OS == "Solaris") => (Memory >= 48)
```

Dependencies:
```
D1:  (CHARMM && !installed("CHARMM", 4.5))
     solve("http://www.charmm.com/dist/agent")
```

Artifacts:
```
A1:  (Version == "1.0") => A1.1, A1.2, A1.3
A1.1:(Arch == "sparc" && OS == "Solaris") =>
         /v1.0/sparc/go -> ./bin
A1.2:(Arch == "i86pc" && OS == "Solaris") =>
         /v1.0/i86pc/go -> ./bin
A1.3:(Arch == "i86pc" && OS == "SGI-MIPS") =>
         /v1.0/sgi/go.exe -> ./sgi-bin
A2:  (Version == "1.2") => A2.1, A2.2, A2.3
A2.1:(Arch == "sparc" && OS == "Solaris") =>
         /v1.2/sparc/go1.2 -> ./bin
A2.2:(Arch == "i86pc" && OS == "Solaris") =>
         /v1.2/i86pc/go1.2 -> ./bin
A2.3:(Arch == "i86pc" && OS == "SGI-MIPS") =>
         /v1.2/sgi/go1.2.exe -> ./sgi-bin
A3:  (Fast) => fast.lib -> ./lib
A4:  (Precise) => precise.lib -> ./lib
A5:  (OurOwnFunction) => eval.lib -> ./lib
```

Figure 5: Example of a Software Deployment System Model in the Software Dock.

bilities of system models in the field of software deployment against the comparison dimensions listed in Section 3. A characteristic of all these system models is a focus on composition. Although the explicit modeling of connections and behavioral specifications is lacking, other compositional constructs are unique to the discipline of software deployment. In particular, it is possible to model a system out of multiple systems, possibly located at multiple geographical sites [28]. Another important feature is the the optionality of components. Instead of modeling a system as a single monolithic entity, it is possible for the system to be described as a set of core components and a number of optional components that may or may not be included when the system is deployed.

Similar to the discipline of configuration management, consistency in the field of software deployment is only supported via frozen configurations. If arbitrary components are selected to be jointly deployed, no mechanism is in place to verify the consistency of the overall system.

Construction of a system is not supported at all. The system models assume an approach in which binaries are shipped from a producer to a consumer site. Consequently, no constructs are available to model the building of a system out of its sources.

Versioning and selection are at the center of software deployment. Specific versions of components are selected to be deployed out of the pool of available versions. As compared to configuration management, the modeling capabilities are rather similar. The only difference is that software deployment does not have an explicit notion of a change set in its modeling capabilities. Although sophisticated delta mechanisms are available for efficient software updates [22], these mechanisms lack the expressiveness of change sets.

Support for the final dimension, dynamism, is not available in current software deployment systems. Despite the fact that it has been recognized as a problem and is an integral part of the software deployment life cycle [30], no solutions have been devised as of yet.

7 LESSONS TO BE LEARNED

Based on the comparison carried out in the previous three sections, we now attempt to answer the questions posed in the introduction. We first answer the question of whether architecture could be used as an organizing abstraction for configuration management and software deployment activities, and then present some requirements for extensions to software architecture that we believe are needed to make the approach a success.

7.1 Software Architecture as an Abstraction

The first question posed was whether software architecture can be used as an organizing abstraction for activities in the software life cycle, and if so, what some of the benefits that arise would be. For the specific cases of configuration management and software deployment, we offer the following observations.

- *The use of software architecture would support the development of new capabilities.* Both the discipline of configuration management and the discipline of software deployment would benefit from the provided modeling capabilities for connections and interaction behaviors. Connections can be used to reduce architectural erosion, whereas interaction behaviors can be used to verify the consistency of selections of components [31]. Both of these are important capabilities that are currently not available.

- *The use of software architecture would reduce the context switch between configuration management and software deployment.* Currently, once a system has been developed and needs to be deployed, a complete new system model has to be devised. This system model has to be created in a new modeling language that uses different modeling constructs. As a result, the context switch between configuration management and software deployment is rather large. If, instead, software architecture was used as an organizing abstraction for both activities, the context switch would be reduced because both activities would operate within a common abstraction.

- *The use of software architecture would reduce modeling effort.* At present, a system is modeled three times, once as an architectural model, once as a configuration management model, and once as a software deployment schema. Consequently, some capabilities, such as, for example, composition, that are needed in all three models have to be modeled three times. If software architecture would be used as an organizing abstraction, this redundancy would disappear because the common objects in the configuration management and software deployment system models can be automatically generated from the software architecture.

Based on these observations, there certainly is a desire to use software architecture as a common abstraction. With respect to the feasibility of this, we believe the answer is also positive. As we can see from Table 1, the modeling constructs provided by software architecture complement the ones provided by the other disciplines. Moreover, as we can see from the previous sections, common constructs, such as components, interfaces, and configurations, serve the same role in each discipline and can therefore be subsumed by the constructs provided by architecture description languages.

7.2 Extending Software Architecture

The second question to answer is whether software architecture by itself is sufficient to serve as the abstraction. To this question, our answer is negative. For the approach to be a success, software architecture needs to be enhanced with the following two modeling capabilities.

- *Versioning and selection.* Versioning and selection should be added to the abstraction for a variety of reasons. First, although an architecture is meant to be relatively stable, it does evolve as a system evolves. Second, different variants of a system could require different architectures. Finally, as we saw in the examples, the activities of configuration management and software deployment both require a system model that incorporates versioning and selection. For all of these, the versioning and selection capabilities that are needed are based on the same principles. Moreover, they all version and select the same construct, namely components. Therefore, it is desirable to put a single versioning and selection mechanism in the abstraction.

- *Optionality.* The ADLs developed to date all assume that a single software architecture exists that describes the components that are present in a single system configuration. As we have seen in the software deployment example, this assumption is not necessarily always true. Systems exist in which multiple configurations all adhere to the same architecture, but in which different sets of optional components are present beyond the set of core components. This optionality should be added to the abstraction because it is a property that needs to be managed in each of the areas of software architecture, configuration management, and software deployment.

If these two capabilities were to be added to software architecture and the resulting abstraction would be used to support activities in the software life cycle, it is our belief that this abstraction is rather complete and that the specific modeling capabilities needed for each activity reduce to rather simple extensions to the central abstraction. The configuration management and software deployment activities provide two examples of this. The extra modeling required in the configuration management discipline would reduce to the modeling of the build process, whereas the extra modeling needed in the software deployment discipline would reduce to the modeling of the constraints among component as well as the modeling of the source and destination locations of the artifacts to be deployed.

8 CONCLUSIONS

In this paper we have started to investigate how software architecture could serve as a basis for a component-based software development process. In particular, we have concluded that software architecture, extended with versioning and optionality, can and should be used as an organizing abstraction that supports configuration management and software deployment. We believe that this conclusion extends to other activities in the software life cycle, which is already demonstrated by initial work in architectural-based testing [26].

To realize this vision, much work remains to be done and difficult problems remain to be solved. We name:

- Creating new ADLs that incorporate modeling capabilities for versioning and optionality.
- Implementing new configuration management and software deployment environments that are centered around software architecture.
- Mapping architectural components to source files and executables.

Each one of these represents a complex problem that will require a significant amount of research. As such, this paper raises more questions than that it answers. However, it does answer one important question with respect to the candidacy of software architecture as an organizing abstraction for activities in the software life cycle. We believe that architecture provides the right modeling capabilities to be this abstraction, and strongly advocate it be taken out of its focus on high-level design and into a focus that centers around its role as a key element in a component-based software development process.

REFERENCES

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

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

[3] 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.

[4] S.C. Cheung, D Giannakopoulou, and J. Kramer. Verification of Liveness Properties Using Compositional Reachability Analysis. In *Proceedings of the Sixth European Software Engineering Conference*, number 1301 in Lecture Notes in Computer Science, pages 227–243, New York, New York, September 1997. Springer-Verlag.

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

[6] D. Compare, P. Inverardi, and A.L. Wolf. Uncovering Architectural Mismatch in Dynamic Behavior. *Science of Computer Programming*, 1999. To appear.

[7] R. Conradi and B. Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 1998. To appear.

[8] Desktop Management Task Force Inc. *Desktop Management Interface Specification*, March 1996.

[9] 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.

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

[11] 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.

[12] 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.

[13] R.S. Hall, D.M. Heimbigner, and A.L. Wolf. Requirements for Software Deployment Languages and Schema. In *Proceedings of the Eighth International Symposium on System Configuration Management*, Lecture Notes in Computer Science, New York, New York, 1998. Springer-Verlag.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] D. Nachbar. When Network File Systems Aren't Enough: Automatic Software Distribution Revisited. In *Proceedings of the USENIX 1986 Summer Technical Conference*, pages 159–171, Atlanta, GA, June 1986. USENIX Association.

[23] 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.

[24] 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.

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

[26] 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.

[27] Tivoli Systems, Inc. *Applications Management Specification*, 1995.

[28] Tivoli Systems, Inc. *TME/10 Software Distribution*, 1997.

[29] 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.

[30] A. van der Hoek, R.S. Hall, A. Carzaniga, D.M. Heimbigner, and A.L. Wolf. Software Deployment: Extending Configuration Management Support into the Field. *Journal of Defense Software Engineering*, 11(2):9–13, 1998.

[31] A. van der Hoek, D.M. Heimbigner, and A.L. Wolf. Versioned Software Architecture. In *Third International Software Architecture Workshop*, 1998. To appear.