A Framework for Analyzing Configurations of Deployable Software Systems

Dennis Heimbigner Richard S. Hall Alexander L. Wolf

Department of Computer Science
University of Colorado
Boulder, CO 80309-0430 USA
{dennis,rickhall,alw}@cs.colorado.edu

Abstract

Configuring and deploying a large software system is complicated when the system is composed of components and when there are numerous possible configurations for the system. In such a scenario, it is difficult for end-users to specify and install an appropriate configuration for their specific environment. Defining all valid configurations of a software system is challenging, and can be addressed through a concise specification that can generate all of the possible configurations. The Deployable Software Description (DSD), part of the University of Colorado Software Dock project, is one such specification format. But using the DSD runs the risk that the set of generated configurations includes some that are invalid with respect to constraints defined independently of the DSD. This paper describes a framework to support the analysis of DSD specifications to help developers detect potentially invalid configurations. This analysis assumes that the system components are annotated with properties, and an analysis tool is provided that takes a specific configuration and analyzes it for conflicts with respect to some set of constraints. Using the DSD, we can enumerate and analyze configurations to verify their validity. The results can be used to modify the DSD to avoid future generation of invalid configurations.

1. Introduction

Software deployment is the complex process [6] that covers all of the activities performed after a software system has been developed. These activities include configuring, releasing, installing, updating, reconfiguring, and removing a software system. For modern software systems, particularly those built from independently developed components and subsystems, the deployment process is complicated by the dimensions along which a system can be configured. That is, the "system" may in fact represent a family of software systems, where a family is defined as the set of all revisions and variants of a software system. Determining the particular configuration to deploy requires the coordination and combination of information about the software system and about the configuration of the *field site,* the site into which the software is to be deployed.

Describing all of the valid configurations of a software system family is challenging, and it may be effectively impossible to manually enumerate all of the possible combinations that might be deployed when that enumeration is very large. One approach to managing this explosion of configurations is to define a concise specification that can be evaluated with varying parameters to generate all of the possible configurations. The Deployable Software Description [7] (DSD) is such a specification developed for the University of Colorado Software Dock project [6, 4].

Using this kind of specification runs the risk that the set of configurations that can be generated may include invalid ones. For example, combinations of specific versions of components may be incompatible, or certain functional capabilities may not be available on all operating system platforms. The DSD format provides some built in constraint mechanisms (described in Section 2) to avoid generating invalid configurations. Unfortunately, some constraints may be too complex to represent in the DSD specification language. Further, there are any number of constraints that are defined externally to the DSD, but which someone may wish to apply to the software family to verify that those extra constraints are not violated.

Component interfaces provide a simple example. Suppose that all artifacts associated with a system are annotated with the set of interfaces that each *provides* and with the set of interfaces that each *requires*. A simple external constraint would enforce all required interfaces to be provided by some artifact. Any configuration for which this constraint is false would be considered invalid. While the DSD could represent this constraint, that representation would be extremely cumbersome, and would not be practical to maintain in the face of an evolving system.

Disk space footprint is another example that is used later in this paper to illustrate the analysis process. Each artifact in a particular configuration can be assigned a number representing the disk space it uses. Given this information, it is easy to compute the total disk space used by each configuration and verify it against a constraint specifying the allowed maximum.

The goal of this paper is to define an extensible framework for analyzing configurations to determine their validity with respect to externally defined constraints. We report on the procedures to be used in this framework. The framework itself is under construction and we expect to report separately on its performance and utility. This paper is organized as follows. Section 2 describes the format and content of a DSD specification. Sections 3, 4, 5, and 6, describe our analysis approach. Section 7 discusses possible improvements to the framework. Section 8 discusses related work. Appendix A describes the DSD example used to illustrate the analysis approach.

2. The Deployable Software Description

The Software Dock project supports automated deployment of complex versioned systems. It provides software producers with a *release dock* that acts as a repository of software system releases. The *field dock* component of the Software Dock supports the field site by providing an interface to the field configuration: its resources and deployed software systems. The Software Dock employs agents that travel from release docks to field docks in order to perform specific software deployment tasks. The agents perform their tasks by interpreting configuration descriptions for both software systems and target field sites. A wide-area event system [1] connects release docks to field docks and enables asynchronous, bi-directional connectivity.

Key to the operation of the Software Dock is the Deployable Software Description (DSD) format. The DSD serves two purposes within the Software Dock. First, it provides a standard schema for concisely specifying any one of the set of possible configurations of a software system. Second, the DSD is interpreted by the Software Dock agents to automatically perform various deployment processes such as installation and removal. It is the former purpose that is of most interest in our analysis framework.

The DSD is an application of the Extensible Markup Language (XML) [11]. DSD defines an XML vocabulary specifically designed to describe complex families of software systems. It models a software system configuration in terms of properties, resources, and constraints. This model is based on our previous analysis of the requirements for such a specification format [5].

Appendix A shows an example of a DSD for a simple application. The example should not be considered particularly realistic; it was constructed to illustrate as much of the algorithm as possible. The complete DSD format is complex and cannot be completely described here; a detailed description of it is available elsewhere [7, 4]. The complete format contains more information than needed for the purposes of this paper, and so inessential elements of the specification have been elided to save space and simplify the presentation. The essential elements are grouped into four sets of items: property definitions, composition rules, constraints, and artifact collections. Each of these sets is described in the following sections.

2.1. Property Definitions

Properties are simple, typed name-value pairs. Appendix Section A.1 of our example illustrates several such properties. Properties are actually divided into two classes: external and internal. External properties refer to properties that describe the field site. The *OS* property defined in Appendix Section A.1, lines 2-6 is an example of an external property. Other examples (not shown) might be *CPU* or *PostscriptPrinter*.

In contrast, internal properties describe features of the system itself. The *OnlineHelp* property (A.1, lines 35-40) is one such property. Other properties (not shown) might be *Version* or *DebugLevel*. Since our analysis presumes to target all possible system configurations independent of any specific field site, the external-internal distinction is irrelevant, and so all properties will be grouped together for the procedures described here.

A *property definition* specifies a number of attributes for the property, including the name, type, and default value. The name is any string, and the default value is any legal value of the appropriate type. The property type may be a String (e.g., *OS* (Operating System) and *Implementation*), an Integer (e.g., *MaxPlayers*), or a Boolean (e.g., *OnLineHelp*, *WinHelp*, and *HTMLHelp*).

For properties from infinite types (Integer and String), the property definition may optionally specify the expected set of values for the property. The *Implementation* property illustrates this by defining its expected set of values, namely "Java" and "Native" (A.1, line 13). The property definition also defines certain other values referred to as the "enabled" value and the "disabled" value. The use of these special values will be described in the next section.

2.2. Composition Rules

Composition rules define relationships between properties. For example, if the *OS* property has the value "Solaris", then all other properties that are inconsistent with that operating system must somehow be disabled.

The composition rules, along with the constraints discussed in the next section, are the primary means for controlling the set of configurations encoded into the DSD. The intent of composition rule sets is to use the minimal set of properties to support the generation of all the configurations of a software family while avoiding the generation of any invalid configurations.

Composition rules have three essential parts: a predicate, a relationship operator, and a set of *target* properties. Appendix Section A.2 shows three example composition rules. Consider the third rule (A.2, lines 26-33), which is essentially equivalent to this expression.

$\underline{\text{OS != “Win95” \&\& OS != “WinNT”}}$ EXCLUDES {WinHelp}

The underlined expression is, of course, the predicate. The relationship operator is *EXCLUDES*, and the set of target properties is {*WinHelp*}. This rule may be read as saying: “if the operating system is not Win95 and not WinNT, then the *WinHelp* property must be *disabled*.”

Four relationship operators are used in the DSD specification.

- **EXCLUDES** – If the Boolean expression evaluates to True, then the set of excluded properties must be assigned the appropriate disabling value. This operator is illustrated by the first, third, and fourth composition rules (A.2, lines 4-11 and lines 26-43).
- **INCLUDES** – If the Boolean expression evaluates to True, then the set of included properties must be assigned the appropriate enabling value. Unlike the *EXCLUDES* operator, the *INCLUDES* operator does not require the target properties to be assigned the exact enabling value specified in the property definition. As with the C programming language, any value that is not the disable value is considered to be an enabling value.
- **ONEOF** – If the Boolean expression evaluates to True, then *exactly one* of the listed properties must be enabled and the others must be disabled. This operator is illustrated by the second composition rule (A.2, lines 14-23). That rule requires either WinHelp or HTMLHelp to be enabled if OnlineHelp is enabled.
- **ANYOF** – If the Boolean expression evaluates to True, then zero or more of the of the listed properties may be enabled.

The first three operators require properties to be “enabled” or “disabled”. For Boolean valued properties, these states naturally map to the values True and False, respectively. But for Strings and Integers, there is no such natural mapping. Thus, properties with these types must specify the mapping of enabled and disabled to two different values in the type. The *MaxPlayers* property, for example, specifies that the enabled value is “2” (A.1, lines 27-29) and the disabled value is “1” (lines 30-32).

2.3. Constraints

The DSD makes provisions for the fact that not all property value assignments lead to valid configurations by allowing for the specification of constraint expressions defined in terms of property values. If the constraint evaluates to False, then the chosen property values are assumed to lead to an invalid configuration.

The DSD constraints are actually divided into two classes: *assertions* and *dependencies*. Assertions describe constraints that must be true else the configuration is invalid. Appendix Section A.3, lines 2-8 illustrates a simple assertion constraint that constrains the field site to use one of three operating systems.

(OS ==“Win95”) || (OS ==“WinNT”) || (OS ==“Solaris”)

Specification of any other operating system causes the assertion to fail, and the configuration of properties to be considered invalid.

Dependencies represent the other class of constraints. They differ from assertions in that a failed dependency may be resolvable. Thus a dependency has an associated action that is invoked to attempt to make the dependency true. Section A.3, lines 17-27 illustrates a simple dependency constraint. In the Software Dock, resolution is currently only provided for dependencies that refer to the installation of required subsystems. Since the analysis currently is limited to single systems, dependency constraints can safely be ignored.

2.4. Artifacts

The artifacts portion of the DSD describes the actual physical artifacts that comprise the software system. The artifacts are described by grouping them into sets (called “collections”) of related artifacts. Each artifact collection is controlled by a *guard*: an expression over properties that determines the inclusion of the artifact collection into a specific configuration. To simplify the guard expressions, artifact collections can be nested in other artifact collections that have additional guards.

Appendix Section A.4 shows the artifacts for our example system. The lines 1 and 90 delimit the outermost collection of artifacts. Within that collection, are two nested collections and two conditionally included artifacts. The first collection (lines 2-31) has the guard expression *Implementation == “Native”* (line 3) that controls which files are included for the non-Java implementation. In this case, the file *solitaire.exe* is unconditionally included. The second artifact in that collection (lines 18-30) itself has a nested guard *MaxPlayers* > 1 that specifies if the multi-player games file (*multiplayer.exe*) should be included.

The second collection (lines 32-63) has the guard expression *Implementation* == *"Java"* that indicates which files are included for the Java implementation. It is completely parallel with the first collection; it always includes *solitaire.class* and, if *MaxPlayers* > 1, includes *multiplayer.class*.

The third and fourth items in the outermost collection are single artifacts. The first of these two artifacts (lines 64-76) includes a Windows format help file (*cardgames.hlp*) if the *WinHelp* property is enabled. The second of these two artifacts (lines 77-89) includes an HTML format help file (*cardgames.html*) if the *HTMLHelp* property is enabled. Note that because of the configuration rule (A.2, lines 14-23), only one of these help files will be included, although it is possible for neither to be included.

3. An Analysis Framework

The ultimate goal is to provide a framework to support the analysis of a complete family of configurations with respect to a variety of external constraints. The approach is to iterate over a set of configurations and verify that some set of such constraints is true for each configuration.

Achieving this goal requires the attainment of several capabilities.

1. It must be possible to enumerate the set of all the legal configurations of a software system. This set essentially comprises the family for that system.

2. It must be possible to define the external constraints to be validated over a set of configurations.

3. It must be possible to analyze a given configuration to see if our constraints are true for that configuration. Note that this analysis must be independent of the particular configuration.

Subsequent sections will demonstrate how to achieve these capabilities.

4. Generating Deployable Configurations

The first capability is generating the set of configurations that make up the family for a given software system. Our approach is to use the DSD specification of that system to generate those configurations.

Before discussing how to do this, it is necessary to more carefully define the term "configuration". This term will refer to the set of artifacts selected by an assignment of values to all the properties specified in the DSD such that the values satisfy the types of the properties and such that all assertion constraints specified in the DSD are satisfied. Occasionally, we will blur the distinction between the property value assignment and the corresponding set of selected artifacts.

Briefly, the configurations are generated as follows.

1. Associate a finite set of values – the *candidate set* – with each property (both external and internal) defined in the DSD.

2. Iterate over all possible assignments of candidate sets to properties. Each such assignment will be referred to as a *candidate assignment*.

3. For each candidate assignment, validate the assignment against the composition rules in the DSD.

4. For each candidate assignment, validate the assignment against the assertion constraints in the DSD.

5. For each candidate assignment, evaluate the guards for the artifact collections in the DSD and accumulate the set of selected artifacts as designated by the guards which evaluate to True. This cumulative set is the *candidate configuration*.

6. Apply the appropriate analyzer to each resulting candidate configuration.

Sections 4.1 and 4.2 detail the processes for generating candidate sets and candidate assignments. Sections 4.3 and 4.4 describe the validation of the candidate assignments against the DSD. Section 4.5 shows how to convert the candidate assignment to a candidate configuration. The details of the analysis procedure are deferred to Section 5.

4.1. Finding Candidate Sets

The initial activity is to determine the set of candidate values associated with a given property. The goal is to exercise all of the expressions associated with composition rules, assertions, and artifact collections specified in the DSD. In other words, we need to determine the exact set of property value assignments such that each expression in the DSD evaluates at least once to True and at least once to False. Since the expression language used is reasonably expressive, it is not easy to calculate the *exact* set of property values necessary to properly and minimally exercise the expressions. In practice the expressions are written in a highly stylized form that allows for the calculation of a superset of the required set of property values. When we use this superset, we may waste some computational resources in trying to construct a configuration using a set of property values that is in fact invalid with respect to the DSD.

The initial candidate set for each property is constructed from information in the property definition itself. That definition provides three sources of values for the property. First, there is the default value specified for the property. Second, the "enabled" and "disabled" values provide additional values. Third, there may be an implied or explicit expected value set for the property. Properties of type Boolean always have the implied value set consisting of {True, False}. Infinite types (Integer and String) may optionally have an explicitly stated value set. The *Implementation* property shows an example of this (A.1, line 13).

Additional items for the candidate set are determined by examining the use of properties in expressions in the DSD. All types participate in equality and inequality comparisons of the following forms.

<property> = <value> or <property> != <value>

For these comparisons, the <value> is added to the set of candidate values for the <property>. For example, analyzing expressions involving the *OS* property (A.2, lines 6 and 28 and A.3, lines 4-5) indicates that the candidate set must include the values "Win98", "WinNT", and "Solaris." Note that technically, the != operator should require the inclusion of a value different from the comparison value, but this is not in fact necessary since all properties are guaranteed a second value as a result of specifying both enabled and disabled values.

In addition to the equality operators, numerically typed properties may be involved in inequality expressions of the following form.

<property> <inequality> <value>

For various inequality operators, Table 1 shows how to calculate elements to add to the candidate set. *MaxPlayers* is the only numerically typed property in the example. Applying the rule above for the ">" operator to A.4, line 19, causes the value "2" ($= 1 + 1$) to be included in the candidate set for this property.

Inequality	Candidates
$<$	$< value > -1$
$>$	$< value > +1$
$<=$	$< value >, < value > -1$
$>=$	$< value >, < value > +1$

Table 1. Candidate Values for Inequalities

Table 2 shows the candidate sets constructed by applying the rules described above to the example in Appendix A.

Property	Candidate Set
OS	{Win98, WinNT, Solaris}
Implementation	{Java, Native}
MaxPlayers	{1, 2}
OnlineHelp	{True, False}
WinHelp	{True, False}
HTMLHelp	{True, False}

Table 2. Example DSD Candidate Value Sets

4.2. Generating Candidate Assignments

Once a candidate value set is associated with each property, the complete set of all possible *candidate assignments* can be generated by choosing all possible combinations of values from the candidate sets of the properties. Again, referring to Table 2, we can see that there are $3 \times 2 \times 2 \times 2 \times 2 \times 2 = 96$ candidate assignments, which is greater than 2^6. This reflects the fact that the number of assignments is exponential in the number of properties defined in the DSD. This follows because every property has at least two values in its candidate value set, namely the enable and disable values. Problems in dealing with an exponential set will be addressed in Section 7.

The set of candidate assignments has another problem besides its potential size. Because it is generated as a naive cross-product, not all of the assignments will be consistent with the expressions in the DSD. Thus, any assignment that assigns the value "Solaris" to the *OS* property and assigns the value True to the *WinHelp* property will be inconsistent with the rule in A.2, lines 26-33. Table 3 shows the candidate assignments given the candidate sets from Table 2. The table only lists consistent assignments, so the size of the table is less than the 96 possible assignments. Note also that the value "WinXX" in that table is intended to stand for either "Win98" or "WinNT".

4.3. Composition Rule Processing

The set of composition rules is processed by iterating over each one in turn and evaluating it against the candidate assignment. As the composition rules are processed, it may be determined that the current assignment is invalid. This means that some composition rule requires a property to have some value different than is specified in the assignment. Recall this example from Section 2.2.

OS != "Win95" && OS != "WinNT" EXCLUDES {WinHelp}

If the excluded property (*WinHelp*) is not assigned the correct disabling value (False), then the assignment is invalid. If an invalid assignment is detected, then the current candidate assignment is abandoned and a new one is selected.

For each of the four relationship operators used in composition rules, we can define their evaluation with respect to the current candidate assignment. In each case, the associated expression is evaluated using the candidate values. If

Index	Order: {OS, Implementation, Players, OnlineHelp, WinHelp, HTMLHelp}
1	WinXX, Java, 1, T, F, T
2	WinXX, Java, 1, F, F, T
3	WinXX, Java, 1, F, F, F
4	WinXX, Java, 2, T, F, T
5	WinXX, Java, 2, F, F, T
6	WinXX, Java, 2, F, F, F
7	WinXX, Native, 1, T, T, F
8	WinXX, Native, 1, T, F, T
9	WinXX, Native, 1, F, T, F
10	WinXX, Native, 1, F, F, T
11	WinXX, Native, 1, F, F, F
12	WinXX, Native, 2, T, T, F
13	WinXX, Native, 2, T, F, T
14	WinXX, Native, 2, F, T, F
15	WinXX, Native, 2, F, F, T
16	WinXX, Native, 2, F, F, F
17	Solaris, Java, 1, T, F, T
18	Solaris, Java, 1, F, F, T
19	Solaris, Java, 1, F, F, F
20	Solaris, Java, 2, T, F, T
21	Solaris, Java, 2, F, F, T
22	Solaris, Java, 2, F, F, F

Table 3. Example Candidate Assignments

the expression evaluates to False, then we ignore that composition rule. If the expression evaluates to True, we apply the following further tests.

- **EXCLUDES** – The values assigned to the target set of properties are examined. If any one of them does not have the appropriate disabling value then the composition rule fails.

- **INCLUDES** – The values assigned to the target set of properties are examined. If any one of them does not have an appropriate enabling value then the composition rule fails. Recall that any value that is not the disable value may be considered enabled.

- **ONEOF** – The values assigned to the target set of properties are examined. If more than one of them has an appropriate enabling value or if none of them has an enabling value then the composition rule fails.

- **ANYOF** – This operator is always valid since it places no effective constraints on the values of the target properties.

If any composition rule fails the above tests, then the current candidate assignment is abandoned and a new assignment is selected.

4.4. Validating Constraints

After a candidate assignment of property values has passed the composition rules, it must be validated against the DSD constraints. As discussed in Section 2.3, dependency constraints are ignored for validation purposes. This is because dependencies always refer to the presence of other deployable systems and currently, our analysis is restricted to single systems. Future work on this framework may include analysis of dependent subsystems. In that case, dependency constraints must be considered.

Assertion constraints cannot be ignored, however, because they directly control the validity of various configurations. Having passed the composition rules, our configuration construction procedure has an initially valid assignment of property values. The next step is to iterate over all the assertion constraints specified in the DSD and evaluate them against the candidate assignment. If the assertion evaluates to False, then the assignment is invalid and must be abandoned in favor of the next candidate assignment.

4.5. Artifact Selection

The last step in configuration construction is to iterate over the guarded collections of artifacts in the artifact section of the DSD to collect the set of artifacts that comprise the valid configuration associated with the current candidate assignment. Note that it is possible for two different assignments to produce the same collection of artifacts.

For each artifact collection (and each artifact) specified in the DSD, any associated guard is evaluated using the current candidate assignment of property values. If the guard evaluates to False, then the collection and any nested collections are discarded. If the guard evaluates to True, then the artifacts specified in the collection are added to the set of artifacts comprising the candidate configuration under construction. If a collection contains nested collections, then the procedure recurses to iterate over those inner collections. The resulting set of artifacts is the output from the configuration construction procedure. It is passed to the analysis procedure to determine if it is a valid configuration with respect to the external constraints associated with the analysis package. Table 4 shows the candidate configurations generated by the candidate assignments in Table 3.

5. Analyzing Deployable Configurations

Given a candidate configuration of artifacts, it must be analyzed to see if it satisfies some set of constraints that are external to the DSD. Since the set of possible external constraints is large and expanding over time, the analysis framework must support an evolving set of analyses. The

Artifact Set	Generating Assignments	FootPrint
{solitaire.class}	{3, 19}	5
{solitaire.class, cardgames.html}	{1, 2, 17, 18}	5 $+30 = 35$
{solitaire.class, multiplayer.class}	{6, 22}	5 $+10 = 15$
{solitaire.class, multiplayer.class, cardgames.html}	{4, 5, 20, 21}	5 $+10$ $+30 = 45$
{solitaire.exe}	{11}	20
{solitaire.exe, cardgames.hlp}	{7, 9}	20 $+10 = 30$
{solitaire.exe, cardgames.html}	{8, 10}	20 $+30 = 50$
{solitaire.exe, multiplayer.exe}	{16}	20 $+40 = 60$
{solitaire.exe, multiplayer.exe, cardgames.hlp}	{12, 14}	20 $+40$ $+10 = 70$
{solitaire.exe, multiplayer.exe, cardgames.html}	{13, 15}	20 $+40$ $+30 = 90$

Table 4. Example Candidate Configurations

approach taken here is to annotate all of artifacts used in a system with sufficient information so that analysis can determine if any subset of those artifacts satisfies some specified external constraint. This annotation is independent of the artifact grouping defined in the DSD for the system. An analysis procedure is also provided that can verify adherence to some set of external constraints when given those annotations.

We will refer to the combination of the annotations, the external constraints, and the associated analysis tool as an *analysis package*. It turns out that the DSD can support the addition of any number of annotations to artifact definitions. This allows the set of analysis packages to grow over time just by adding new annotations.

Section 1 discussed interface mismatches and disk space footprint as examples of constraints that can be tested by an analysis package. The artifact descriptions in Appendix Section A.4 have been annotated with footprint information (see A.4, line 15, for example) indicating the amount of disk space used by that artifact. Given a constraint of the form $footprint <= 80$, a simple analyzer for a set of artifacts can total up the footprints of each of the artifacts in each configuration and verify that the total footprint is less than the maximum specified by the constraint. If this verification fails, then that particular set of artifacts is considered invalid. The last column of Table 4 shows the total footprint for each candidate configuration. The last entry fails the constraint.

We can now describe the overall operation of our analysis framework. We start with the DSD and use the procedure described in the last section to generate, one by one, the candidate property value assignments and the corresponding configurations as artifact sets. For each configuration, we iterate over all applicable analysis packages. For each analysis that fails, our framework records the candidate property assignment, the configuration, and any output from the analysis and reports the fact that this configuration is invalid with respect to the analysis.

6. Feedback from Analysis to DSD

When an analysis package reports that a given configuration is invalid, it is useful to feed that result back into the DSD to prevent the future generation of the invalid configuration. Note that this feedback differs from completely encoding the associated external constraint into the DSD; this latter encoding may not be feasible. Rather, the idea is to encode direct knowledge of invalid configurations into the DSD. If the number of bad configurations is large, then this approach may not be attractive.

The simplest way to prohibit a specific configuration is to add an additional constraint that is of the following form, where each of the $property_i$ and $value_i$ comes from the candidate assignment and the $operator_i$ is either an equality (=, ! =) or inequality (<, >, <=, >=).

```
not (
        $property_1\ operator_1\ value_1$
    and $property_2\ operator_2\ value_2$
    ...
    and $property_n\ operator_n\ value_n$
   )
```

Two complications attend the use of such assertions. The first concerns inequalities. Some values in the candidate assignment are really values of the form <value>-1 or <value>+1, and were added to satisfy inequalities in expressions in the DSD. To be strictly correct, the assertion expression should retain the actual inequality operator that generated those values. To do this, we need to shadow each value in a candidate assignment with the corresponding actual equality or inequality that generated that value. It is those actual inequalities that are used in the new assertion. Table 2 shows an example of such a shadow. The value "2" for the *MaxPlayers* property can occur in two ways. It can occur directly via the "enabled" attribute in the property description (A.1, lines 27-29). It can also occur by applying Table 1 to the predicate at A.4, line 19.

Since the last configuration in Table 4 failed the footprint test, the following assertion could be added to the

DSD to prevent its generation.

```
not   (
     (OS == “Win98” & Implementation == “Native”
        & MaxPlayers > 1 & OnlineHelp
        & not WinHelp & HTMLHelp)
     | (OS == “Win98” & Implementation == “Native”
        & MaxPlayers > 1 & not OnlineHelp
        & not WinHelp & HTMLHelp)
     | (OS == “WinNT” & Implementation == “Native”
        & MaxPlayers > 1 & OnlineHelp
        & not WinHelp & HTMLHelp)
     | OS == “WinNT” & Implementation == “Native”
        & MaxPlayers > 1 & not OnlineHelp
        & not WinHelp & HTMLHelp)
)
```

The other complication with a feedback assertion concerns its size. It will be proportional in length to the total number of properties referenced in the whole DSD, and this may produce a rather unwieldy assertion. In practice it is likely that only a subset of the properties are essential to the expression, but determining that subset is difficult and would require substantial help from the analysis tools. Alternatively, the assertion may be reducible to an equivalent composition rule. The expression above, for example, can be reduced to the following equivalent expression using knowledge about the DSD.

```
Implementation != “Native” | MaxPlayers == 1 | not HTMLHelp
```

Again, this is difficult to do automatically. These kinds of assertion reductions remain a subject for future research.

7. Optimizations

The configuration construction procedure described in Section 4 is essentially generate and test. That is, it generates all the possible candidate assignments and tests them against the DSD and then the analysis packages. As described in Section 4.2, the number of candidate assignments is exponential in the number of properties defined in the DSD. In practice this behavior may be tolerable when the number of properties is not large. For example, if the number of properties is no more than 20, then there would be some one million possible assignments, and that seems tractable if the analysis is only performed occasionally and non-interactively. But 30 properties would result in more than a billion assignments, and that seems intractable.

So any optimization capable of reducing the number of candidate assignments is welcome. One such optimization concerns the reduction of the number of candidate assignments to be considered. This can be accomplished in two ways: (1) by reducing the number of candidate values associated with properties and (2) by more quickly recognizing infeasible assignments. A simple optimization of the first kind concerns the selection of values for inequalities as described in Table 1, where we were given a numeric inequality of the following form.

$$<\text{property}> <\text{inequality}> <\text{value}>$$

In response to these inequalities, we added $<$value>-1 or $<$value$>+1$ as candidate values for the specified property. But clearly these values do not need to be included if there are values in the candidate set that already make the inequality true.

The second kind of optimization is more difficult to arrange. Table 3 shows such a reduction, but it was constructed by hand using knowledge about the predicates in the DSD. It is possible that we can exploit highly structured predicates (those in some normal form, for example) to detect large classes of infeasible assignments.

Avoiding unnecessary analysis is another form of optimization that would reduce the overall running time. Table 4 shows that the same artifact set can be generated by multiple candidate assignments. If we calculate and store a unique hash signature for each artifact set, then re-analysis of duplicated sets can be detected and avoided.

8. Related Work

Traditional configuration management (CM) systems provide an alternate platform on which to build an analysis framework similar to the one described in this paper. The Adele [3] CM system in particular embodies some kinds of automatic analysis for interfaces that is one kind of analysis package. Adele introduces the notions of interfaces and realizations of those interfaces. A specific interface may have many revisions, where each revision may have multiple realizations and these realizations may also have many revisions. Adele uses constraints over these attributes to express the composition of valid configurations. Adele, as with similar CM systems such as PCL [9], targets source code configurations as opposed to deployment configurations. It also does not provide for general and extensible analysis packages.

The Deployable Software Description format has a number of competitors that might appear capable of serving to generate our configurations for analysis. Again, both Adele and PCL have relatively sophisticated modeling languages that can serve that purpose. A number of other, less sophisticated description languages have been defined. The Open Software Description (OSD) [10] provides a vocabulary for describing software components, their versions, underlying structure, and relationships among components. The Desktop Management Task Force (DMTF) has defined the Management Information Format (MIF) [2], which is a common, hierarchical data model used in describing all aspects of computing systems, including software systems. Tivoli Corporation created the Application Management Specification (AMS) [8], which is an effort related to, and a superset of, the Software MIF. None of these last three description formats, OSD, MIF, or AMS, can serve to replace DSD as the basis of our framework. With the exception of OSD, none can describe multiple revisions, and none can describe variants for a software system in detail. As a consequence, none can describe consistent configurations for a complete software family and this is the core capability required for the analysis framework described in this paper.

9. Conclusion

We have presented a general framework for the analysis of highly configurable systems. This framework, using the Deployable Software Description format taken the University of Colorado Software Dock project, generates all the configurations for a system and then applies specified analysis packages to each of those configurations to detect problems with those configurations. The results of these analyses can be fed back to the DSD to avoid the future generation of those configurations. This framework is currently under construction and will eventually be integrated with the Software Dock to provide new capabilities for the developers of highly configurable and deployable systems.

Acknowledgements

This material is based upon work sponsored 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 does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] A. Carzaniga, E. Di Nitto, D. S. Rosenblum, and A. L. Wolf. Issues in supporting event-based architectural styles. In 3^{rd} *International Software Architecture Workshop*, Orlando, FL. USA, Nov. 1998.

[2] Desktop Management Task Force. *Software Standard Groups Definition, Version 2.0s*, 24 June 1998.

[3] J. Estublier and R. Casallas. The adele configuration manager. In *Configuration Management*, pages 99–134. Wiley, 1994.

[4] R. S. Hall. *Agent-based Software Configuration and Deployment*. PhD thesis, University of Colorado, Boulder, Colorado, June 1999.

[5] R. S. Hall, D. Heimbigner, and A. L. Wolf. Requirements for software deployment languages and schema. In *Proc. of the 8th Intl. Symposium on System Configuration Management (SCM-8)*, Brussels, Belgium, July 1998. Springer-Verlag Lecture Notes in Computer Science, number 1439.

[6] R. S. Hall, D. Heimbigner, and A. L. Wolf. A cooperative approach to support software deployment using the software dock. In *Proc. of the 1999 Intl. Conf. on Software Engineering*, pages 174–183. IEEE Computing Society, May 1999.

[7] R. S. Hall, D. Heimbigner, and A. L. Wolf. Specifying the deployable software description format in XML. Technical Report CU-SERL-207-99, University of Colorado Software Engineering Research Laboratory, Mar. 1999.

[8] Tivoli Systems. *Application Management Specification, Version 2.0*, 1997. (http://www.tivoli.com/ o_products/html/ body_ams_spec.html).

[9] E. Tryggeseth, B. Gulla, and R. Conradi. Modeling systems with variability using the proteus configuration language. In *Proc. of the 1995 Int'l Symposium on System Configuration Management*, pages 216–240. Springer, 1995.

[10] A. van Hoff, H. Partovi, and T. Thai. *The Open Software Description Format (OSD)*. Microsoft Corp. and Marimba, Inc., 1997. (http://www.w3.org/TR/ NOTE-OSD.html).

[11] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0*, 1998. (http://www.w3.org/ TR/1998/ REC-xml-1998-0210).

A. Example DSD Specification

A.1. Properties

```
<ExternalProperties>
    <Property>
      <PropertyName>OS</PropertyName>
      <PropertyType>STRING</PropertyType>
      . . .
    </Property>
</ExternalProperties>

<InternalProperties>
    <Property>
      <PropertyName>Implementation</PropertyName>
      <PropertyType>STRING</PropertyType>
      <PropertyValues>Native, Java</PropertyValues>
      <PropertyDefaultValue>Java</PropertyDefaultValue>
      <PropertyDefaultEnabled>
        Native
      </PropertyDefaultEnabled>
      <PropertyDefaultDisabled>
        Java
      </PropertyDefaultDisabled>
      . . .
    </Property>
    <Property>
      <PropertyName>MaxPlayers</PropertyName>
      <PropertyType>INTEGER</PropertyType>
      <PropertyDefaultValue>1</PropertyDefaultValue>
      <PropertyDefaultEnabled>
        2
      </PropertyDefaultEnabled>
      <PropertyDefaultDisabled>
        1
      </PropertyDefaultDisabled>
      . . .
    </Property>
    <Property>
      <PropertyName>OnlineHelp</PropertyName>
      <PropertyType>BOOLEAN</PropertyType>
      <PropertyDefaultValue>true</PropertyDefaultValue>
      . . .
    </Property>
    <Property>
      <PropertyName>WinHelp</PropertyName>
      <PropertyType>BOOLEAN</PropertyType>
      <PropertyDefaultValue>false</PropertyDefaultValue>
      . . .
    </Property>
    <Property>
      <PropertyName>HTMLHelp</PropertyName>
      <PropertyType>BOOLEAN</PropertyType>
      <PropertyDefaultValue>true</PropertyDefaultValue>
      . . .
    </Property>
</InternalProperties>
```

A.2. Composition Rules

```
<Composition>
    <!–Force “Implementation” to be Java if
        the operating system is not Win98/NT>
    <CompositionRule>
      <RuleCondition>
        (($OS$ != ”Win95”) && ($OS$ != ”WinNT”))
      </RuleCondition>
      <RuleRelation>EXCLUDES</RuleRelation>
      <RuleProperties>Implementation</RuleProperties>
      . . .
    </CompositionRule>
    <!–If “OnlineHelp” selected, then force selection of
        either “WinHelp” or “HTMLHelp”, but not both>
    <CompositionRule>
      <RuleCondition>
        $OnlineHelp$ == true)
      </RuleCondition>
      <RuleRelation>ONEOF</RuleRelation>
      <RuleProperties>
        WinHelp, HTMLHelp
      </RuleProperties>
      . . .
    </CompositionRule>
    <!–Force “WinHelp” property to be false if
      “OS” is not Win98/NT>
    <CompositionRule>
      <RuleCondition>
        ($OS$ != ”Win95”) && ($OS$ != ”WinNT”)
      </RuleCondition>
      <RuleRelation>EXCLUDES</RuleRelation>
      <RuleProperties>WinHelp</RuleProperties>
      . . .
    </CompositionRule>
    <!–Force “WinHelp” property to be false if
      “Implementation” is Java>
    <CompositionRule>
      <RuleCondition>
        $Implementation$ == ”Java”
      </RuleCondition>
      <RuleRelation>EXCLUDES</RuleRelation>
      <RuleProperties>WinHelp</RuleProperties>
      . . .
    </CompositionRule>
</Composition>
```

A.3. Constraints

```
<Assertions>
    <Assertion>
      <AssertionCondition>
        ($OS$==”Win95”) || ($OS$==”WinNT”)
        || ($OS$==”Solaris”)
      </AssertionCondition>
      . . .
    </Assertion>
    <Assertion>
      <AssertionCondition>
        $MaxPlayers$ >= 1
      </AssertionCondition>
      . . .
    </Assertion>
</Assertions>
<Dependencies>
    <Dependency>
      <Guard>($Implementation$ == ”Java”)</Guard>
      <DependencyCondition>
        (!installed(”Cards”))
      </DependencyCondition>
      <DependencyFamily>Cards</DependencyFamily>
      <DependencyInterface>
        Install
      </DependencyInterface>
      . . .
    </Dependency>
</Dependencies>
```

A.4. Artifacts

```
<Artifacts>
    <Artifacts>
      <Guard>($Implementation$ == ”Native”)</Guard>
      <Artifact>
        <ArtifactSourceName>
          solitaire.exe
        </ArtifactSourceName>
        <ArtifactSource>
          /cardgames/win/bin
        </ArtifactSource>
        <ArtifactDestinationName>
          solitaire.exe
        </ArtifactDestinationName>
        <ArtifactDestination>bin</ArtifactDestination>
        <FootPrint>20</FootPrint>
        . . .
      </Artifact>
      <Artifact>
        <Guard>($MaxPlayers$ > 1)</Guard>
        <ArtifactSourceName>
          multiplayer.exe
        </ArtifactSourceName>
        <ArtifactSource>/cardgames/win/bin</ArtifactSource>
        <ArtifactDestinationName>
          multiplayer.exe
        </ArtifactDestinationName>
        <ArtifactDestination>bin</ArtifactDestination>
        <FootPrint>40</FootPrint>
        . . .
      </Artifact>
    </Artifacts>
    <Artifacts>
      <Guard>($Implementation$ == ”Java”)</Guard>
      <Artifact>
        <ArtifactSourceName>
          solitaire.class
        </ArtifactSourceName>
        <ArtifactSource>
          /cardgames/java/classes
        </ArtifactSource>
        <ArtifactDestinationName>
          solitaire.class
        </ArtifactDestinationName>
        <ArtifactDestination>classes</ArtifactDestination>
        <FootPrint>5</FootPrint>
        . . .
      </Artifact>
      <Artifact>
        <Guard>($MaxPlayers$ > 1)</Guard>
        <ArtifactSourceName>
          multiplayer.class
        </ArtifactSourceName>
        <ArtifactSource>
          /cardgames/java/classes
        </ArtifactSource>
        <ArtifactDestinationName>
          multiplayer.class
        </ArtifactDestinationName>
        <ArtifactDestination>bin</ArtifactDestination>
        <FootPrint>10</FootPrint>
        . . .
      </Artifact>
    </Artifacts>
    <Artifact>
      <Guard>($WinHelp$ == true)</Guard>
      <ArtifactSourceName>
        cardgames.hlp
      </ArtifactSourceName>
      <ArtifactSource>/cardgames/doc</ArtifactSource>
      <ArtifactDestinationName>
        cardgames.hlp
      </ArtifactDestinationName>
      <ArtifactDestination>doc</ArtifactDestination>
      <FootPrint>10</FootPrint>
      . . .
    </Artifact>
    <Artifact>
      <Guard>($HTMLHelp$ == true)</Guard>
      <ArtifactSourceName>
        cardgames.html
      </ArtifactSourceName>
      <ArtifactSource>/cardgames/doc</ArtifactSource>
      <ArtifactDestinationName>
        cardgames.html
      </ArtifactDestinationName>
      <ArtifactDestination>doc</ArtifactDestination>
      <FootPrint>30</FootPrint>
      . . .
    </Artifact>
</Artifacts>
```