Dependable software deployment

John DeTreville
Microsoft Research
john.detreville@microsoft.com

Daan Leijen
Microsoft Research
daan@microsoft.com

Wouter Swierstra
University of Nottingham
wss@cs.nott.ac.uk

Abstract

Large and complex software systems, like those common on personal computers, often contain many components *that can be deployed separately—applications, libraries, drivers,* etc.*—but that must then be bound together into working* configurations. *Configuring software is difficult and error-prone in practice, and it is not well understood in theory. As a result, real software systems are often configured in ways that are fragile and undependable. To address these problems, we present a novel and precise model for reasoning about software configurations, and the processes by which they are constructed. We can compare our framework to existing software deployment tools, such as Windows Installer and the RPM Package Manager, and formalize desirable properties of software configurations.*

1. Introduction

Configuring computer systems can be just as difficult as programming them. A system administrator must choose from among a large number of packaged software components *(e.g.,* applications) and bind them together—by copying files, setting component options, updating system options, *etc.*—to create a system that correctly provides some desired functionality for an end user. When the components do not fit perfectly together, or when additional customizations become necessary, system administrators may need to fall back on the same sorts of problem-solving skills as programmers. It is therefore unfortunate that most end users of personal computers must serve as their own system administrators even though they lack the necessary skills and training. As a result, a great many of the world's personal computers are misconfigured. They do not behave as expected; they are fragile; they are insecure; they are undependable.

We would like to lower the cost of configurability by making configuration less error-prone. Some simple aspects of system configuration are already automated via the use of installers, package managers, dependency analyzers, *etc.* It seems clear that further automating system configuration can, in principle, broadly improve system dependability by reducing misconfigurations, especially misconfigurations of personal computers by end users. This paper provides several novel contributions to that end.

We present a functional approach to system configuration that we contrast with the traditional imperative approach (Section 2). The functional approach makes it easy to reason about, and to enforce rules on, entire configurations.

We further present a framework for reasoning about software configurations (Section 3). We not only describe the *deployment* of individual software components—*e.g.,* the sequence of network fetches and disk writes going from some "old" state to some "new" state—but also demonstrate how to reason about the *consistency* of an entire software system configuration at a given time. Our framework is buttressed by a mathematical model that lets us state formal, verifiable properties of software configurations, and of the software configuration process.

We illustrate the expressiveness of our framework using realistic examples. We present a precise model of configurations (Section 3), and discuss potential implementations (Section 4). We extend the basic model with arbitrary constraints on configurations (Section 5) and parameterisable components that form the basis of deployment (Section 6). We make clear comparisons of our approach against existing software deployment tools (Section 7). Finally, we report our experience developing a small prototype implementation that serves to illustrate our approach (Section 8).

2. A functional approach to software configuration

Personal computers traditionally take an imperative approach to system configuration. They allow a series of incremental updates to the configuration state by installing new software components, updating old components, uninstalling and perhaps reinstalling yet other components, and so on. A system's current "ground truth" configuration—its current collection of files, directories, registry settings, *etc.,*

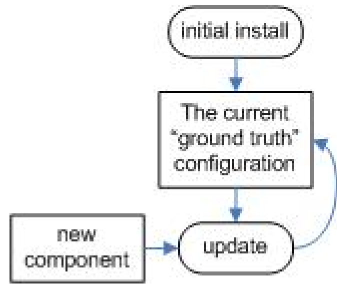

Figure 1. The imperative approach to software configuration.

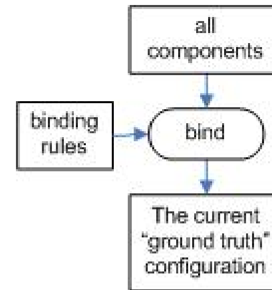

Figure 2. A functional approach to software configuration.

that belong to these components—is the result of all of these imperative incremental updates that the system administrator has performed so far, as depicted in Figure 1. It is not uncommon for hundreds of updates to have been performed on a system's configuration to date, and it is not uncommon for one update to create errors that are then propagated to all future configuration states. It is little wonder that end users so often choose to reconfigure their computers from scratch.

In contrast, we take a functional approach to software configuration, as depicted in Figure 2. We take the complete set of desired components, and we *bind* them together, based on a set of binding rules, to compute a complete "ground truth" configuration. If we change the set of components, or we change the rules on how they are to be bound together, we can repeat this process to produce another "ground truth" configuration, which we may then make current. The current configuration is never an input to an "update" step, so errors cannot propagate as in the imperative approach, and adding components in a different order cannot give a different result. In essence, all changes cause the configuration to be recomputed from scratch, which more readily allows us to ensure global consistency. The challenge is to do this easily and efficiently.

With the imperative approach, the system configuration is *mutable,* and it is updated frequently. With the functional approach, the system configuration is *immutable,* and it can only be replaced atomically and in its entirety. (User files remain mutable, of course.) We can always recompute an earlier configuration if we retain access to the appropriate unbound components and to the appropriate binding rules, and we can even compute configurations for distant machines or devices without accessing their current configurations. Multiple configurations can exist side by side on one personal computer, and we can switch between them freely. If the multiple configurations are similar, they can easily share disk storage since they are immutable.

3. Software configuration management

Before we can talk more precisely about configuration management, we must first define what a configuration means. Intuitively, a configuration is set of components that can refer to each other, and consist of values like directories, libraries, or registry entries. In general though, the particular values are arbitrarily diverse and we choose here to treat many values abstractly as uninterpreted bits, as we are not interested in the values per se.

In order to reason about values that introduce namespaces, we assume that all values can define child values. More precisely, we have a function *children* that returns the children of a namespace value, where the children are defined as a partial function from names to values. For example, the children of a library value bind names to function entries, while the children of a directory define the contents of a directory.

We do treat not all values as completely abstract. *Attributes* have values of fixed type that we can interpret. Attribute names are underlined to distinguish them from normal names. For example, each value defines a child $\underline{sort}$ that defines the type of a value, where typical examples are dir or component. Other common attributes are $\underline{version}$ and $\underline{name}$.

Under the above model, a configuration is simply a value with sort config that has as its children values of sort component, that in turn define other values. Here is an example of a configuration C consisting solely of a *libc* component, which contains a single file named *libc.a*.

$$
\begin{array}{l}
C = \textsf{config} \\
\quad\quad libc = \textsf{component} \\
\quad\quad\quad \underline{name} \quad = \texttt{"libc"} \\
\quad\quad\quad \underline{version} = 5 \\
\quad\quad\quad libc.a \quad = \ldots
\end{array}
$$

We compose values by writing the sort in sans serif (config), followed by the children defined by the value. Indentation

is used to scope the children and values. Since actual raw values are not important, we do not denote them here. Of course, attributes do have significant values and those are denoted directly, as in $\underline{name}$ = `"libc"`.

We distinguish between identifiers and component names, just as programming languages distinguish between identifiers and strings. Here, the identifier *libc* is our name for this component, while the string `"libc"` is the component's own name for itself.

Since values can have children, we can specify more detail. For example, We know that *libc.a* is a library that defines function entries and we can name them explicitly:

$$
\begin{array}{l}
libc = \mathsf{component} \\
\quad \underline{name} \quad = \texttt{"libc"} \\
\quad \underline{version} = 5 \\
\quad libc.a \quad = \mathsf{library} \\
\qquad\qquad\qquad exit \quad = \mathsf{function} \\
\qquad\qquad\qquad printf \ = \mathsf{function} \\
\qquad\qquad\qquad \ldots
\end{array}
$$

Of course, each function entry can itself contain more information, like the type or calling convention:

$$
\begin{array}{l}
exit = \mathsf{function} \\
\quad \underline{ctype} \quad = \texttt{"void (int)"} \\
\quad \underline{callconv} = \texttt{"ccall"}
\end{array}
$$

The framework gives a detailed declarative definition of software components and configurations. These declarations are extremely flexible and can capture precise information about how components are made up. Most importantly, these declarations lend themselves well to analysis. We can now state and check properties over a configuration since all values are explicitly declared, and because they do not depend on an unknown "current" configuration state.

3.1. Binding

The *libc* component has a single isolated file, but more interesting issues arise when components refer to one another. Here we define the larger *ghc* component, which contains a directory *bin* containing the exported executables. (Comments follow a -- on a line.)

$$
\begin{array}{ll}
ghc = \mathsf{component} & \\
\quad \underline{name} \quad = \texttt{"Glasgow Haskell Compiler"} & \\
\quad \underline{version} = 6.4.2 & \\
\quad \underline{pred} \quad = libc \cdot libc.a & \\
\quad bin = \mathsf{dir} & \\
\qquad ghc.exe \ = \ldots & \text{-- the compiler executable} \\
\qquad ghci.exe = \ldots & \text{-- the interpreter executable}
\end{array}
$$

(The actual *ghc* component defines many more files and other values, which we elide here.) For the *ghc* component to function correctly, it must be bound to the files and components it depends on. Here, we refer to a file within another component using the path $libc \cdot libc.a$. The attribute $\underline{pred}$ declares a predicate that must hold for the component to be valid. We return to predicates later in Section 5; for now, it is enough to know that a plain path can be considered an import statement used to express dependencies between components.

Of course, child values can also refer to other components. In particular, executables might require specific function entries in libraries. Beside (or instead of) the general $libc \cdot libc.a$ library, the ghc executable could require specific library functions to be defined:

$$
\begin{array}{l}
ghc.exe = \mathsf{executable} \\
\quad \underline{pred} = libc \cdot libc.a \cdot exit \wedge libc \cdot libc.a \cdot read
\end{array}
$$

The bottom line is that *ghc* refers to the *libc* component that it does not define itself. For such definitions to make sense, we must resolve names unambiguously. The *ghc* component is well defined only in combination with an independent definition of $libc \cdot libc.a$. If the *ghc* component requires specific functions, such as $libc \cdot libc.a \cdot exit$, these procedures must be present. We cannot always consider unbound components in isolation but must always reason about entire software configurations.

To formalize the notion of binding, we first define some helper functions that return the bound names and child values of value:

$$
\begin{array}{l}
values\ v = codom\ (children\ v) \\
names\ v = dom\ (children\ v)
\end{array}
$$

Note that we write the application of a function f to an argument x as $f\ x$, which associates to the left and binds stronger than any operator. We can also recursively enumerate all values defined by a value using *allc*:

$$
\begin{array}{l}
allc\ v = \{v_1 \mid v_0 \in values\ v, v_1 \in all\ v_0\} \\
all\ v \ \ = \{v\} \cup allc\ v
\end{array}
$$

A path is constructed using the (right-associative) $\cdot$ operator, as in $libc.a \cdot read$. The empty path is written as ε. Given a value and a path, we can select a value at that path with the selection operator @:

$$
\begin{array}{ll}
v@\varepsilon & = v \\
v@(n \cdot p) & = ((children\ v)\ n)@p \\
\perp @p & = \perp
\end{array}
$$

We assume that *children* is a partial function where $children\ v\ n = \perp$ if $n \notin dom\ (children\ v)$. When $v@p = \perp$, we say that the path is unbound.

Well-formed Using the above definitions, we can now state useful properties over configurations. For example, a configuration C contains only components as its children:

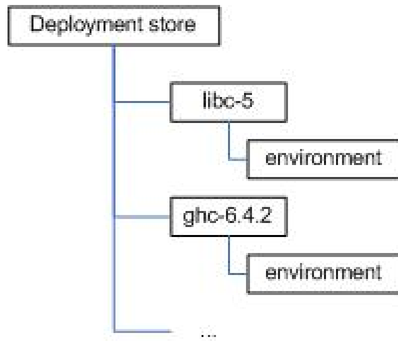

Figure 3. A deployment store.

$\forall (c \in \mathit{values}\ C).\ c \cdot \underline{\mathit{sort}} \equiv \mathsf{component}$

Another constraint is that components cannot be nested:

$$v \cdot \underline{\mathit{sort}} \equiv \mathsf{component} \\ \Rightarrow \forall (w \in \mathit{allc}\ v).\ w \cdot \underline{\mathit{sort}} \neq \mathsf{component}$$

When the above two predicates hold, we say that a configuration is *well-formed*. In general, there might be many predicates that we expect to hold over particular structures, for example, files cannot be nested. We return to this in Section 5 that discusses how general predicates can be used to constrain valid configurations.

Resolved The function *free q* returns the free paths in a predicate q in the obvious way. We can extend this definition to values too:

$$\mathit{free}\ v = \{p \mid p \in \mathit{free}\ (\mathit{values}\ v), v@p = \perp\}$$

Note that paths are lexically scoped, and that all paths are bound unambiguously since *children* is a function.

We say that a well-formed configuration C is *resolved* when all paths are bound:

$$\mathit{free}\ C = \varnothing$$

A resolved configuration is a strong notion since all bindings are unambiguously resolved. *Binding* is a crucial part of system configuration. Unbound components may have many dependencies, and a set of components might be assembled in many different ways. Only when we bind components together (*e.g.*, when we bind *ghc* to a particular $\mathit{libc} \cdot \mathit{libc.a}$) do we create one particular bound configuration, where every component has an unambiguous definition of the other components and contents that it depends on.

4. The deployment store

How can we represent such bindings? Inspired by Nix [8], we make a software deployment tool responsible for managing the actual files. Bound components are installed in a special *deployment store;* an example is depicted in Figure 3. Every component lives in a separate directory. Here, we name these directories using the component's name and version. An alternative is to follow Nix's lead and use a cryptographic hash of the bound component to name the directory.

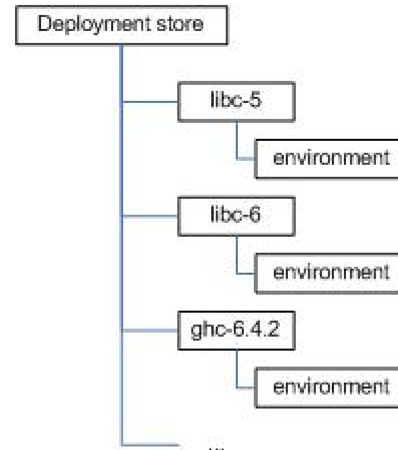

Figure 4. A deployment store containing multiple versions.

How can we manage bindings of the components in the store? Most operating systems provide a form of scope management by means of environment variables. In particular, by manipulating the *search path* we can affect which files are found. Another (and better) mechanism would be to change the linker to use the static information from the configuration to directly link libraries without any search at all.

The files and directories in the deployment store must be immutable. The software deployment tool can insert new bound components—including components whose bindings are changed—and delete old ones, but other programs must not. This allows us to reason about a bound configuration statically, since later changes are disallowed.

Of course, one configuration can replace another. This can be achieved by wholesale replacement of the bound components in the deployment store, or by computing and applying a minimum edit set between old old configuration and the new one.

How should a program access files stored in the store? Clearly, a principal point of automated software deployment is to hide implementation details behind well-designed interfaces. We propose that bound components can advertise certain functionality that may be of interest to system administrators or to end users. These can produce, for example, shortcuts that do little more than set the environment variables and run the desired application from the store.

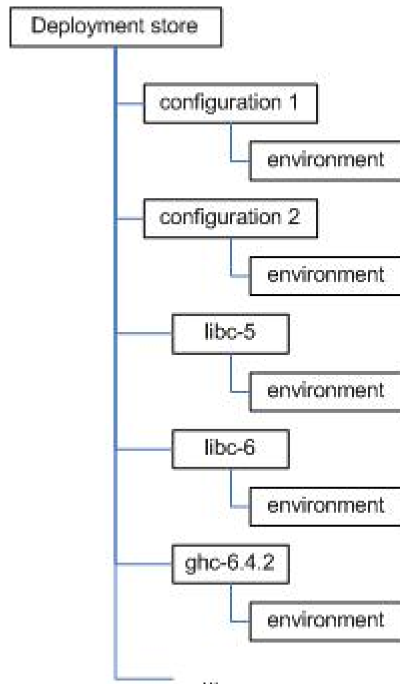

Figure 5. A deployment store containing multiple configurations.

4.1. Multiple versions

A given bound configuration can contain multiple side-by-side versions of *libc*, as shown in Figure 4. Some bound components may then use the latest version, while others rely on an older version. Both versions can coexist peacefully; the environment settings enforce components' bindings. This guarantees that every component is unambiguously bound to the components and contents that it is intended to depend on.

4.2. Multiple configurations

A deployment store can even contain multiple side-by-side independent configurations, as shown in Figure 5. It can contain all of the bound components in either configuration, plus each configuration. Multiple configurations can exist side by side, where each is unambiguously bound to the correct components. Similar configurations will of course have many files and directories in common, but these can easily share storage on a disk. Additionally, a new configuration can be constructed atomically while an old one is running. As with components, we can give these configurations friendly names, or we can name them by their hashes.

All this is possible on today's operating systems with no extensions whatsoever. This is an important consideration when designing a software deployment tool. In an ideal world, we would have the freedom to design a fresh operating system with much better support for ensuring immutability and scoping names and hiding the deployment store. The fact of the matter, though, is that managing software configurations is a problem *now* and so it is useful to come up with a solution that can work well with today's technology.

5. Predicates

A personal computer, when booted, will tend to run whatever software it finds on its disk, but most possible disk contents are more or less meaningless. Similarly, just because a component refers to a file called *libc.a* does not necessarily mean that any file with that name will do. We allow programmers to be much more specific about which components are known to work together smoothly. A *ghc* programmer may require that the component *libc* has the name `"libc"` and a sufficiently recent version. Such additional requirements can be specified in a $\underline{pred}$ attribute. Here's the relevant fragment of the *ghc* component:

$$
\begin{aligned}
&ghc = \mathsf{component}\\
&\quad \underline{name} \;\; = \texttt{"Glasgow Haskell Compiler"}\\
&\quad \underline{version} = 6.4.2\\
&\quad \underline{pred} = libc \cdot version \geqslant 5 \wedge libc \cdot name \equiv \texttt{"libc"}\\
&\quad \ldots
\end{aligned}
$$

When these predicates are satisfied, we can avoid many kinds of surprises when we use the identifier *libc*. Requiring components with a certain name and version is a recurring pattern in software deployment tools.

What kinds of expressions can we allow in the predicate language? Many predicate languages are possible, but here we assume predicates over first-order logic. The ability to quantify over components is particularly powerful, as we will see below.

Valid We say that a *resolved* configuration C is *valid* if all predicates hold:

$$\forall (v \in all\ C).\ v \cdot pred$$

A valid configuration is a very strong notion. Since predicates range over first-order logic, we can state many kinds of requirements without special mechanisms. We have already seen how we express dependencies. Other common features of package managers are *conflicts* and *uniqueness*.

Conflicts Some components may *conflict* with others. Despite programmers' best efforts to minimize potential interference between components, there may still be components that simply cannot coexist. Such relationships are,

in a sense, dual to the dependencies: where dependencies specify that a component can *only* work in the presence of a certain other component, conflicts state that it will *never* work if a certain other component is present. For instance, we may want to express that a component can never function in the presence of any component called `"ASCII printer driver"`:

$$\begin{aligned}&C = \mathsf{config}\\&\quad \mathit{printer} = \mathsf{component}\\&\quad\quad \underline{\mathit{pred}} = \forall(c \in \mathit{values}\ C).\\&\quad\quad\quad\quad c\cdot \mathit{name} \neq \texttt{"ASCII printer driver"}\\&\quad \ldots\end{aligned}$$

Uniqueness In a similar vein, some components must be *unique*: *i.e.,* there is only one version of that component installed. This is similar to the conflict relation; in a sense, such a component conflicts with any other version of itself, *e.g.* a component *mon* called `"Monitor driver"` may desire to be the only component with that name. We can express this as follows:

$$\begin{aligned}&C = \mathsf{config}\\&\quad \mathit{mon} = \mathsf{component}\\&\quad\quad \underline{\mathit{pred}} = \forall(c \in \mathit{values}\ C).\\&\quad\quad\quad c\cdot \mathit{name} \equiv \texttt{"Monitor driver"} \Rightarrow c \equiv \mathit{mon}\end{aligned}$$

These two example predicates can be expressed by some existing tools. More often than not, however, such tools cannot express more complicated examples; for instance, you may want to define a color printer driver that works with either a Unicode or an ASCII printer driver, but not both simultaneously.

Having a general language for expressing predicates is, we believe, vital. While some software deployment tools advocate the importance of a flexible predicate language [4], many others fix a subset of what we can express. An insufficiently expressive predicate language can cause a great deal of trouble when additional dependencies arise between components. Having a general predicate language will alleviate some of the pain involved with expressing complicated component dependencies.

6. Abstraction and parameterization

Up to now, we have only considered static configurations that could be well-formed, resolved, and valid. However, we have not considered the definition of components in isolation, and how we can bind such component descriptions into a configuration. In particular, if we develop a component, we generally need to refer to other components but we do not know in which configuration our component will be deployed.

For example, *ghc* refers to an unbound identifier *libc*, which must be bound in a well-formed configuration. We will treat *libc* as a parameter, so that *ghc* can be bound to various values of *libc* in different configurations. Let's make the *libc* value a parameter *libcArg* of a new *ghc component function ghcF*.

$$\begin{aligned}&\mathit{ghcF}\ \mathit{libcArg} = \mathsf{component}\\&\quad \underline{\mathit{name}} \quad = \texttt{"Glasgow Haskell Compiler"}\\&\quad \underline{\mathit{version}} = 6.4.2\\&\quad \underline{\mathit{pred}} = \mathit{libcArg}\cdot\mathit{version} \geqslant 5\\&\quad\quad\quad\quad \wedge\ \mathit{libcArg}\cdot\mathit{name} \equiv \texttt{"libc"}\\&\quad \mathit{bin} \quad = \mathsf{dir}\\&\quad\quad\quad\quad \mathit{ghc.exe} \ = \ldots\\&\quad\quad\quad\quad \mathit{ghci.exe} = \ldots\end{aligned}$$

When specifying components, programmers *must* be abstract about dependencies, and parameterization lets them do so. Here, we allow any *libc* value to be passed as an argument. We use a predicate to specify additional constraints on the value that may be passed as an argument to this function. Specifying and resolving such dependencies is an important part of this approach.

How can we deploy *ghc* using *ghcF*? In order to make an actual component *ghc* that we can deploy, we must pass *ghcF* a suitable argument. If we have a candidate component *libc* installed, we could build the following component:

$$\begin{aligned}&C = \mathsf{config}\\&\quad \mathit{ghc} = \mathit{ghcF}\ \mathit{libc}\\&\quad \mathit{libc} = \ldots\end{aligned}$$

It goes without saying that we could define components that are parameterized with more than one argument; after all, many components have more than one dependency. Creating a configuration is thus nothing more than instantiating component functions to the desired arguments, thereby binding all identifiers. The goal is of course to create a *valid* configuration that meets certain user requirements.

By passing different arguments, we could bind different versions of *libc* to a *ghc* value. By passing in components, we bind the arguments' names and construct an actual component value. This is what software deployment is really about: binding components' abstract imports and exports over to actual components and actual contents on the target system.

The arguments of a component function determine the dependencies of a component. By adding a $\underline{\mathit{depends}}$ attribute one can state predicates over the transitive closure of dependents of a component. This is important when one want to ensure for example that within one process, we never load two different versions of the same component. It is beyond the scope of this paper to discuss this 'diamond problem' in detail, and we solve this particular problem in

practice by assuming an initial set of implicit global predicates that hold for every configuration.

The deployment problem The user requirements are simply a predicate that state the desired components that need to be deployed. Such predicate is parameterized over the target configuration C, and consists of a conjunction of existential quantifications over components, as in $\exists(c \in \mathit{values}\ C).\ c \cdot \mathit{name} \equiv \texttt{"Word"}$.

The *deployment problem* is to take the user requirements, and a collection of component functions and instantiate those functions in such a way that the resulting configuration is valid and satisfies the user requirements. If no valid configuration can be found, the deployment fails. Of course, in general no unique or best solution exists, but we argue that in practice there is often a best possible configuration under an appropiate *policy*.

Furthermore, instantiating parameters can be automated. Provided a software deployment tool knows which components are available—perhaps because they are available for download, or on a compact disc, or already installed—it easy to decide when a component is a valid argument to a parameterized component. Clearly, the candidate argument must define all the imports of the parameterized component.

Deployment can be viewed as recursively selecting needed components, instantiate arguments of component functions, filter valid configurations, and use a policy to select a 'best' configuration.

6.1. Policies

What should an automatic tool do if it can find more than one component that fits the bill? Perhaps some valid arguments may, after all, be preferable to others. A *policy* fixes when one component is better than another. Formally, a policy defines a partial order on components: a binary relation between components that is reflexive, transitive, and anti-symmetric. Using this partial order, we can select a maximally desirable component from amongst the set of all possible valid component arguments. We can imagine policy servers that provide common policies that a user subscribes to. The concept is best illustrated by some examples, defining when a component c_1 is less desirable than another component c_2.

State of the art One obvious (but naïve) policy might be to always prefer the latest version of each software component. To express this policy, we can simply define a partial order on components using the name and version attributes of individual components:

$$c_1 \cdot \mathit{name} \equiv c_2 \cdot \mathit{name} \Rightarrow c_1 \cdot \mathit{version} \leqslant c_2 \cdot \mathit{version}$$

Security For many people, security of software systems would be more important. Many companies and agencies assign security ratings to common software components. Given a function *rate* that assigns a security rating to components, we can express such an example of such a policy.

$$\mathit{rate}\ c_1 \leqslant \mathit{rate}\ c_2$$

Parsimony As a final example, many people want to minimize the time spent on software deployment. Others may want to minimize the amount of disk space a configuration requires. Once again, we define a policy that abstracts over both the notion of *size* a user may have and the current configuration, called *system*. We then try to minimize the size of the components that remain to be installed:

$$\begin{aligned} & \textbf{if}\ c_1 \in \mathit{system}\ \textbf{then}\ 0\ \textbf{else}\ \mathit{size}\ c_1 \\ \leqslant\ & \textbf{if}\ c_2 \in \mathit{system}\ \textbf{then}\ 0\ \textbf{else}\ \mathit{size}\ c_2 \end{aligned}$$

These examples are already quite complex. A serious software deployment tool must have a customizable set of policies. System administrators could then choose the policy that best suits their needs.

Policies do not guarantee that an automated deployment tool always makes the "best" choice. They specify a *partial* order: there may be many elements that are maximal yet incomparable. In that case, we choose from among the maximal elements non-deterministically. We do not believe that this will lead to unexpected behavior, since policies only specify preference between *valid* candidate components. Most existing software deployment tools have some notion of policy built-in, but to our knowledge no such tool can be parameterized over arbitrary policies.

In practice, we depend on the policies when generating bound configurations. Provided we generate potential configurations in a decreasing order, the first valid configuration found will be optimal. We discuss such implementation issues in greater detail in Section 8.

7. Related systems

This section compares our approach to system configuration against similar systems. One comparison is valid for all: we focus on the model of the final configuration, while other approaches stress installation as an action. We generate an immutable ground truth configuration from scratch each time. Since configurations have a well defined value, we can state properties about the final configuration, and check configurations for global consistency. Configurations do not succumb to bit-rot after too many updates, and multiple configurations can readily exist side by side.

7.1. Windows Installer

Microsoft Windows Installer [1] is an installation and configuration service for the Windows operating system; it replaced an earlier hodgepodge of custom-written installers. Windows Installer packages contain definitions (held in tables in a small relational database) that describe a package's contents and how it is to be installed.

The description of a package's contents is similar to ours. A package consists of components. One table lists the files every component contains. This makes it easy to analyze *which* files a package imperatively installs. Similar tables describe the registry settings, fonts, and icons that a component contains.

Unfortunately, it is much less clear *where* files are installed. The contents of a package can be copied to a single directory, but there are also many ways to copy subdirectories of a package to arbitrary locations on the target system. This is understandable—some shared files must be readily available to other components—but as we have argued earlier, sharing files between components in this fashion can be brittle and undependable.

To make matters worse, packages may use imperative *custom actions* to run custom code as part of the installation process. Such custom actions run executables or scripts; it is not uncommon, for example, to use such custom actions to determine where files must be installed. As a result, there is no reliable way to analyze a package statically and determine where it will actually install its files.

Windows Installer works well when describing standalone applications, but the predicates that capture dependencies between separate packages can be awkward. For instance, packages can check whether another component has been installed, and perform special actions if it has or if it has not. Such checks do not query any sort of global configuration database, but simply check whether a certain file associated with the desired component is present. Such dependency checks can be rather undependable! There is also no abstraction mechanism for queries about installed components: the installer for Microsoft Works must separately check for PowerPoint 9, PowerPoint 10, and PowerPoint 11. Without the appropriate abstractions, the opportunity for error clearly increases.

Finally, there is no mechanism for specifying which components may not work together, meaning that programmers must resort again to custom actions. MSN Messenger's installer has a special custom action to uninstall any previous version of Messenger. We argue that the lack of a expressive predicate language limits Windows Installer's suitability for managing many complex software configurations.

7.2. RPM

The RPM Package Manager [10] is one of the most popular deployment tools on Linux. Packages specify a component's name, version, dependencies, and series of instructions to configure, build and install the package. RPM keeps track of which packages have already been installed. This makes it possible for a package to state dependencies on other packages; the package manager ensures that any required packages are also installed.

Dependencies name other packages that must be present, along with simple version requirements. The predicate language is very restrictive; there is no way, for instance, to say that a package will work with either gcc or Microsoft Visual Studio's C compiler.

RPM forbids multiple versions of the same component. As packages install files in shared directories, this is an understandable, yet severe, limitation. As a result, there are many ways in which the order in which packages are installed may affect the final outcome [11].

Even though RPM has a much richer language for predicates and dependencies than Windows installer, it lacks a declarative description of the content of a package and scripts can create files or do other custom actions.

7.3. EDOS

Many of the problems with RPM are addressed in recent work on the EDOS project [5, 14]. The EDOS system is similar to RPM, but extends the predicate language to express *disjunctive* dependencies and *conflicting components*. Rather than specify a flattened list of packages that must be installed, EDOS allows packages to depend on either component X or component Y. Furthermore, packages may specify conflicts, thereby forbidding such components to coexist. Like RPM, EDOS considers two components with the same name but different versions to be a conflict.

Not only does EDOS propose to fix several problems with RPM, the project also gives a formal specification of the package deployment process. There are several desirable properties of configurations that can be formalized using this specification.

EDOS still has a limited predicate language. For instance, EDOS assumes the conflict relation to be binary and here is no way to specify that a component cannot be deployed when both of a pair of other components are present, yet it will work with either component in separately. There is also no policy mechanism: every valid configuration is considered equivalent (although such mechanism can be added to EDOS).

Finally, it is still very much a *package* management system; it cannot say anything about how packages interact with the kernel, drivers, or existing system files.

7.4. Nix

Our work is strongly inspired by Eelco Dolstra's work on Nix [6, 8, 9], a purely functional software deployment tool. Nix has a principled approach to policy-free software configuration management and a mature implementation that can handle non-trivial Linux package installations.

An important difference is our focus on the model of the final configuration constrained under first-order logic predicates, which makes it easy to reason about properties of a configuration. In constrast, Nix has a complex language with a focus on specifying component functions. Since the language is side effecting, and can for example execute scripts at any point, it is much harder to reason about its properties, or the final configuration.

8. Implementation

We have developed a prototype system illustrating our approach to software deployment. Our system uses the purely functional language Haskell [16]. Besides implementing the core resolution algorithm, we developed a simple GUI for browsing and installing software components using wxHaskell [13].

The core resolution algorithm is surprisingly succinct. Although the problem of finding a valid bound configuration is NP-complete, we make great use of Haskell's *lazy evaluation* to simplify the resolution process. Instead of generating *all* possible valid resolutions, we generate a list of resolutions and pick the first valid choice. Rather than compute the whole list, we use lazy evaluation to only compute the list until the first valid configuration is found.

We must be careful when generating all possible bound configurations. To profit the most from lazy evaluation, we must not sort the list of all configurations: this would rely on computing the entire list. As different policies affect the order in which configurations are generated, we must generate the configurations in descending preference.

Besides lazy evaluation, we also use Haskell's ability to pass functions as arguments. The resolution algorithm abstracts over the policy and predicates that are used. We have implemented several simple policies, such as preferring newer versions over older ones. It would be interesting to expose an API for writing policies: this would let system administrators customize the deployment process to suit their needs.

On top of the core resolution algorithm, we have implemented a GUI to install components. Figure 6 has a screenshot illustrating the installation process.

A user typically opens files containing a list of components, which they are then free to browse. Once a user initiates the installation process, they are asked to select the names of the components they wish to install. Using these names, we resolve all the dependencies and compute a proposed resolution. This resolution is then presented to the user, who has a final opportunity to add additional constraints. For instance, the user may choose to require a different version number for a certain component. Once the user is satisfied, the installation begins: files are actually written to disk and suitable environments are created.

We have tested the implementation using a selection of representative DLLs from Microsoft Windows, including substantial DLLs that define thousands of procedures, and import hundreds of procedures from other components. Our component browser is capable of handling such complicated components.

The Achilles' heel of our prototype implementation is of course the resolution algorithm. While we have tried to minimize the amount of unneccessary work, it is clear that the search space could explode. We believe, however, that most components have fairly straightforward requirements. The potentially huge search spaces only occurs when a large number of components have complex interactions, far more involved than simple dependencies. End users who are interested in simply installing stand-alone applications should rarely experience performance issues, but this is an area of future work.

9. Conclusions and further work

We have described a general model to reason about deployment in a purely declarative way. We presented a clear description of a configuration and can reason about its properties. A configuration can be composed from component functions, where general predicates constrain valid configurations. Deployment is parameterised by policies, which are partial orders on components, and used to select a best configuration.

There are still many open questions that warrant further research. In particular, we have assumed that the contents of a component are fixed. Components may depend on other components, but the content they deliver cannot. Yet there are plenty of examples where a component is configurable. Also, allowing general predicates and policies is potentially very expensive, and the interaction between predicates and policies is unclear. It would be very interesting to find specific subsets of these where solutions can be found more efficiently.

Acknowledgements

The authors would like to thank Andres Löh for his valuable comments and Eelco Dolstra for his inspirational work on Nix.

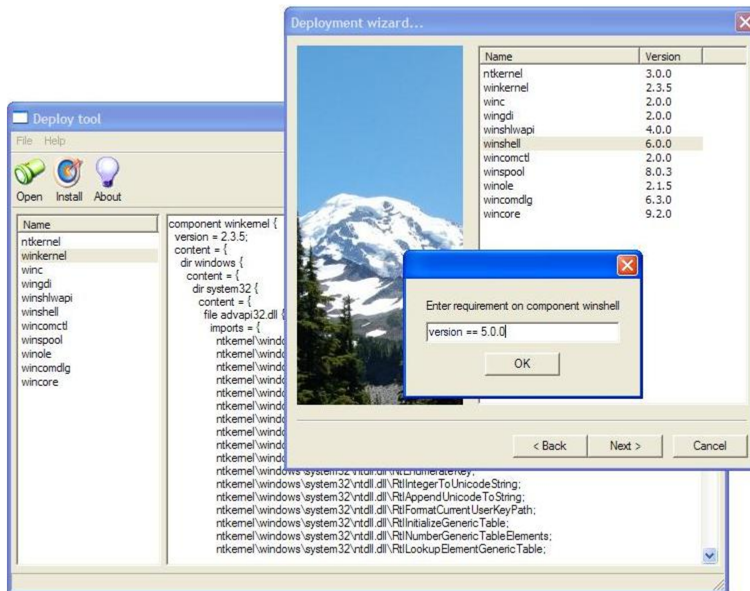

Figure 6. Specifying arbitrary predicates

References

[1] Windows Installer, `http://msdn.microsoft.com/library/`.

[2] FreeBSD Ports Collection, `http://www.freebsd.org/ports`.

[3] M. Burgess. Cfengine: a site configuration engine. In *USENIX Computing systems*, volume 8, 1995.

[4] A. L. Couch and M. Gilfix. It's Elementary, Dear Watson: Applying Logic Programming To Convergent System Management Processes. In *LISA '99: Proceedings of the 13th USENIX conference on System administration*, pages 123–138, Berkeley, CA, USA, 1999. USENIX Association.

[5] R. Di Cosmo, B. Durak, X. Leroy, F. Mancinelli, and J. Vouillon. Maintaining large software distributions: New challenges from the FOSS era. In *Proceedings of the 1st International EASST-EU Workshop on Future Research Challenges for Software and Services*, 2006.

[6] E. Dolstra. Efficient upgrading in a purely functional component deployment model. In *CBSE*, pages 219–234, 2005.

[7] E. Dolstra. *The Purely Functional Software Deployment Model*. PhD thesis, Faculty of Science, Utrecht, The Netherlands, jan 2006.

[8] E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In *LISA '04: Proceedings of the 18th USENIX conference on System administration*, pages 79–92, 2004.

[9] E. Dolstra, E. Visser, and M. de Jonge. Imposing a memory management discipline on software deployment. In *ICSE*, pages 583–592, 2004.

[10] E. Foster-Johnson. *Red Hat RPM Guide*. Red Hat, 2003.

[11] J. Hart and J. D'Amelia. An Analysis of RPM Validation Drift. In *LISA '02: Proceedings of the 16th USENIX conference on System administration*, pages 155–166, Berkeley, CA, USA, 2002. USENIX Association.

[12] A. Heydon, R. Levin, T. Mann, and Y. Yu. *Software Configuration Management Using Vesta*. Springer, 2006.

[13] D. Leijen. wxHaskell – a portable and concise GUI library for Haskell. In *ACM SIGPLAN Haskell Workshop (HW'04)*. ACM Press, Sept. 2004.

[14] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Ingineering (ASE'06)*, Tokyo, Japan, Sept. 2006. IEEE Computer Society Press. To appear.

[15] K. Manheimer, B. A. Warsaw, S. N. Clark, and W. Rowe. The Depot: A Framework for Sharing Software Installation Across Organizational und UNIX Platform Boundaries. In *LISA '90: Proceedings of the 6th System Administration Conferencs*, pages 37–46, 1990.

[16] S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.