A Formal Framework for Component Deployment

Yu David Liu Scott F. Smith

Department of Computer Science
The Johns Hopkins University
{yliu, scott}@cs.jhu.edu

Abstract

Software deployment is a complex process, and industrial-strength frameworks such as .NET, Java, and CORBA all provide explicit support for component deployment. However, these frameworks are not built around fundamental principles as much as they are engineering efforts closely tied to particulars of the respective systems. Here we aim to elucidate the fundamental principles of software deployment, in a platform-independent manner. Issues that need to be addressed include deployment unit design, *when*, *where* and *how* to wire components together, versioning, version dependencies, and hot-deployment of components. We define the *application buildbox* as the place where software is developed and deployed, and define a formal Labeled Transition System (LTS) on the buildbox with transitions for deployment operations that include build, install, ship, and update. We establish formal properties of the LTS, including the fact that if a component is shipped with a certain version dependency, then at run time that dependency must be satisfied with a compatible version. Our treatment of deployment is both platform- and vendor-independent, and we show how it models the core mechanisms of the industrial-strength deployment frameworks.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.2.7 [*Software Engineering*]: Version Control

General Terms Languages, Design, Theory

Keywords Application Buildbox, Deployment, Component, Version

1. Introduction

In [44], Szyperski presents the three defining properties of software components; his first property is that they serve as units of *independent deployment*. This aspect has attracted far less attention from the research community than his other two properties (that components are units of composition and units of state encapsulation). Make no mistake, component deployment is a complex process which must be carefully thought out if correct software behavior is to be achieved: an Enterprise JavaBean [17] might be tested in one container and then deployed in a different container that is incompatible with the bean; the same Java application might display different run-time behaviors on sites with different settings of CLASSPATH to locate dependency components; updating a Windows DLL when installing an application might cause other applications dependent on it to exhibit erratic behaviors; and, with dynamic plugins, components can also be deployed into a running application.

The manner in which industry has responded to these problems is a barometer of real-world need for elegant and rigorous models of deployment: numerous tools [18, 40, 12, 25, 10, 11, 43] have been designed to support component deployment, and the .NET, Java and CORBA platforms define how software components should be deployed [32, 17, 36]. These industrial solutions provide workable systems that users can directly use, but along with this strength comes a weakness: to ensure correct functionality on a particular platform, these solutions are highly embedded in platform-specific details, such as file system structures, environment variables, scripts, middleware architectures, and programming language models. The specifications are also informal, and so no properties can be rigorously guaranteed. For tool-based approaches, non-trivial issues such as dependency resolution are submerged in the code. The best way to understand them in some cases is to run the tools and see what happens! So, we believe there is a real need for the formal study of the component deployment problem.

1.1 This Work

In this paper, we elucidate the fundamental principles of software deployment, in a platform-independent manner. Despite its complexity, software deployment is known to follow a lifecycle where common activities are shared [22, 37]: release from the development site (*shipping*), unpacking at the deployment site (*installation*), reconfiguring at the deployment site in response to changes (*update*), and activating the application into a usable state (*execution*). Is there a common "core" that a good deployment system should have regardless of the choice of the platform? Is there a fundamental and precise notion about what a shipment package should contain, what it means to be a good software update, and what deployments will not violate version compatibility? We answer these questions by coming up with an abstract formalization. Our goal is for these results to give researchers a basis to reason about deployment, and guide future deployment tool developers to come up with systems with well-defined operations that have provable correctness properties.

This paper also indirectly sheds light on component design, from the perspective of what kind of component model is best-suited for deployment. By focusing primarily on this property here, this paper extends our past studies, from different perspectives, of component dependency and linking [42, 27, 28]. Linking – especially its theoretical foundations and type properties – has been extensively studied and well-understood [6, 19, 20, 14, 13]. However, previous studies do not adequately take into account

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-348-4/06/0010...$5.00

the *when* and *where* of linking: Does it happen at build time, installation time, or run time? Does it happen on the development site or the deployment site? If the software is first linked at the development site, and then some of its dependencies are relinked at the deployment site, are these new dependencies compatible with those at the development site?

1.2 Goals and Contributions

The main contributions of our framework can be grouped into four categories, each of which addresses a principal design goal of the framework:

- *Expressiveness*. The framework addresses the entire lifecycle of application evolution, spanning both the development site and deployment site, and covering both how applications evolve statically as well as dynamically. A more detailed discussion on the framework features is given in Sec. 2.
- *Generality*. Our treatment of component deployment is both platform- and vendor-independent, and models the core mechanisms of existing deployment models found in Microsoft products (DLLs, COM[33], CLI Assemblies[32]), Unix/Linux (package installers[18, 12, 40]), Java (EJB[17], Classloaders) and middleware infrastructures (CORBA CCM[36]). Sec. 5 details how our framework can be related to different platforms. Although our framework does not capture every feature in every existing deployment model, it offers a more principled way to implement their core features.
- *Correctness*. We give the first known proof of version compatibility in a software deployment context, which states that if a component is shipped with a certain version requirement for its dependencies, then at run time the dependency must be satisfied with a compatible version. Our framework also provides a formal notion of application well-formedness, and we prove the framework is robust enough to preserve well-formedeness during application evolution. This topic is elaborated in Sec. 4.
- *Simplicity*. Our framework provides a foundational model of the core issues of deployment such as evolution, naming and versioning. Because it is simple, it will be possible to use it as a basis for studying more advanced deployment features such as security and transaction control.

2. Informal Overview

In this section, we informally explain the core ideas of our framework. A few design decisions are also discussed.

2.1 Core Concepts

2.1.1 The Component Model: Assemblages

Software components in our framework are called *assemblages*. Each assemblage is a named, versioned, independently shippable and independently deployable code unit. This abstraction can be mapped to any form of real-world deployment unit, for example an EJB component, a Java `.class` file, an Eclipse plugin, or a C dynamic linking library. (A non-example is the typical use of the C `.o` file on Unix platforms: it is not an independent unit since it relies on linking at the development site before shipment and deployment.) Our goal is to construct a model which elegantly captures the key features of the widely known deployment models; after presenting the technical details we will make a detailed comparison with these models to show how we meet this goal.

Each assemblage may be equipped with two kinds of interfaces, *mixers* and *pluggers*. Both mixers and pluggers are named interfaces which can not only *export* features defined inside the assemblage implementation body to other assemblages, but also

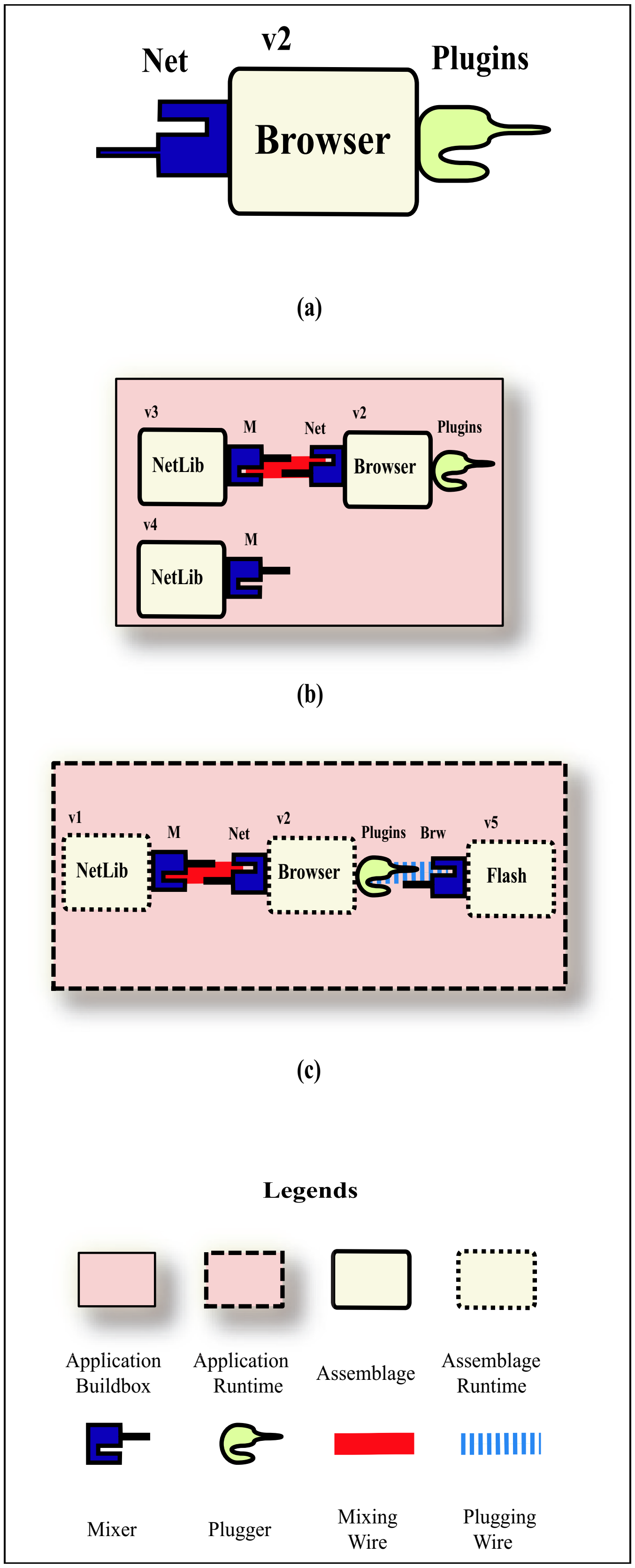


Figure 1. Core Concepts (a) An Assemblage (a) A Snapshot of the Application Buildbox (b) A Snapshot of the Application Runtime

can *import* features from other assemblages. Our framework does not make assumptions on the format of the features themselves; they may be functions, classes, or other structures. For convenience here, we often will use functions as examples. We will explain the difference between mixers and pluggers soon, in Sec.2.1.2. A simple assemblages example is given in Fig. 1(a), showing a browsing tool that is wrapped up as an independent deployment unit. It has the name `Browser`, and version identifier `v2`.

The component model introduced here is based on the module construct of our previous *Assemblages* project [27]. Despite the shared structural similarity, the goal of this work is very different from the original calculus, where type safety was the focus. Since we have previously addressed typing issues, we ignore them in this paper for simplicity.

2.1.2 The Component Wires: Mixing and Plugging

Component-based applications are commonly viewed as a number of individual components wired together [44], where wiring indicates name binding. What matters in a deployment context is *when* the wires are established. The watershed event in a component lifetime is *load time*, when the component turns from a piece of stateless code to a potentially stateful and executable runtime form. We call a wire established before the two wired components go through this watershed event a *mixing wire* (or sometimes simply a *mixing*), and for a wire established after that a *plugging wire* (or sometimes simply a *plugging*). In Fig. 1(b), there is a mixing wire between the networking library `NetLib` assemblage and the `Browser` assemblage, and its nature indicates the wire is established *before* the browser is up and running[1]. In Fig. 1(c), there is a plugging wire between the `Browser` and the `Flash` dynamic plugin. Such a wire is established *after* the browser has been launched.

From an assemblage writer's perspective, declaring a mixer signifies a willingness to establish a wire before it is loaded, while declaring a plugger signifies a desire to establish a wire after loading. A mixing wire binds together a pair of mixers, one from each wired party, so that the imports from one mixer can be satisfied by the exports from the other. A plugging wire is the same as a mixing wire, except it binds together a plugger of an assemblage already loaded and a *mixer* of an assemblage to be loaded as a result of plugging. The use of a mixer here may sound odd, but it indicates the desire of the assemblage being plugged in: `Flash` as a plugin in fact desires its names inside `Brw` interface be resolved the minute it is loaded, not after.

2.1.3 Evolution and Application Buildbox

The study of deployment leads to the study of application evolution. The components forming the application are first deployed one by one, then the wires between them are formed, and then perhaps some components will be updated independently, and eventually the application is launched into the runtime form.

In our framework, this "application in flux", *i.e.* a snapshot of the evolution process where some components of the application are yet to be deployed and some wires are yet to be established, is evolving in an imaginary "box" called an *application buildbox*, or simply a *buildbox*. The buildbox is illustrated in Fig. 1(b). Three assemblages are currently in the buildbox, with two of them being two different versions of the same named assemblage `NetLib`. When the buildbox contains all the components the application needs and all the wires are established, we say *an application is formed*. The buildbox naturally captures the evolving nature of component-based software development and deployment, and serves as a useful mechanism for modeling the process.

After the application is formed, it can be launched and be turned into a stateful and executable *application runtime*. Fig. 1(c) shows a snapshot of an application runtime, which contains a collection of running assemblages, called *assemblage runtimes*. Because plugging wires may be established after application runtimes are established, application execution continues to support limited application evolution.

2.2 The Deployment Lifecycle

A primary goal of our framework is to model all of the fundamental actions involved in the deployment process. To demonstrate this, we now describe how the development and deployment of a simple browser is modeled in our framework. The focus of the framework is decidedly on deployment, but to model deployment we must also model fundamental actions at the development stage, since they have a major impact on deployment. The important development and deployment actions of our framework are illustrated in detail in Fig. 2.

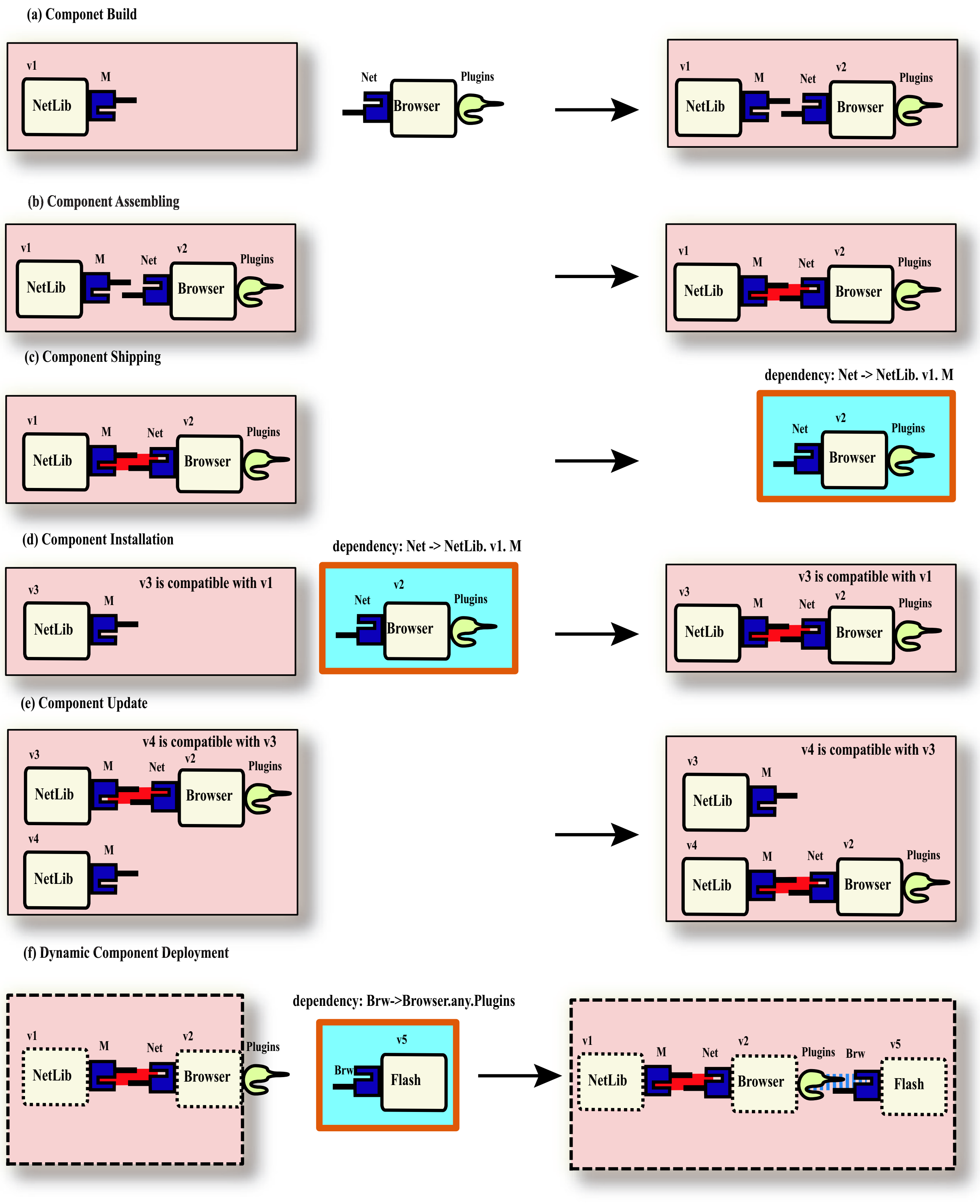
Component Build Component-based software development starts by preparing the components themselves. There are two ways a developer can obtain a component: either by building one from scratch, or by using off-the-shelf components shipped by third parties. The first case is modeled by an action called *component build*, in Fig. 2(a). For a concrete analogy, this step can be viewed as the creation of a Java `.class` file. After this build step, the `Browser` component is now in the buildbox. A new browser version `v2` assigned to the component and uniquely identifies it. The second way in which a component can be obtained (an off-the-shelf component like `NetLib`) will be explained below when component installation is discussed.

Component Assembling In Fig. 2(a), the two components are in the buildbox, but the namespaces have yet to be wired together. The component assembling process achieves this, by wiring together the two mixer interfaces, one from each component. Note that the mixing wire is between two *versions* of components, not just two components. For instance in Fig. 2(b), the framework is not just aware that a `NetLib` component is wired to `Browser` component, but also it is that `v1` of `NetLib` is wired to `v2` of `Browser`.

Component Shipping When development of the browser is finished, the `Browser` component needs to be packed up and shipped. This *component shipping* process is modeled in Fig. 2(c). The result of shipping is the packed form of `Browser`, a data structure called *packaged assemblage*. When a component is shipped, the shipper is given the option of specifying whether each version dependency should be matched when the component is installed elsewhere. In the example here, let us assume the `Broswer` shipper does indeed care about the version of the network library, so that it will not be installed on a machine whose `NetLib` component has an incompatible version that *e.g.* might crash the browser. The resulting packaged assemblage ends up containing a *dependency constraint* saying "my mixer `Net` at development time is wired to the `M` mixer of version `v1` of `NetLib`. I will never let myself be wired to a version of `NetLib` that is not compatible with `v1`". On the other hand, if the shipper decides all that matters is that `NetLib` has a mixer `M` containing the features it needs, but not the versions of them, he/she could also choose to leave out this constraint.

Component Installation The browser enters the deployment site by the action of component installation, illustrated in Fig. 2(d). The installation unit is a packaged assemblage. Note that in the example, the deployment site doesn't happen to have a copy of

[1] Our term "mixing" is sometimes called "static linking" in the literature. We are cautious in using that term, however, because it is overloaded in the C context, where it refers to the linking of `.o` object files. Mixing is analogous to the case in C where an `a.out` file is linked to its dynamic linking library (`.so` files), an action – due to the technicalities of the platforms – called "dynamic linking" in C terminology.



Legends 🟦 Packaged Assemblage

Figure 2. Software Development and Deployment Lifecycle

version v1 of NetLib, only a compatible version. It results in a mixing wire connecting v3 of NetLib and v2 of Browser.

Earlier when we explained component build, we raised the question of how NetLib was put into the buildbox. The Browser developer certainly does not want to build it from scratch, and most likely it is prepared by some other library developer. In our framework, the way a third-party component such as NetLib is put in the buildbox is via *component installation*. Indeed, the *development site* of a browser is in fact the *deployment site* of the network library. This explains from one angle the inseparable relationship between development and deployment, and why a deployment framework also covers the actions in development.

Component Update At some point, there may be a new, improved release of the NetLib, and the system administrator wishes to update the browser to use the newer version. This updating action is modeled in Fig. 2(e), where the wiring to the Net mixer of Browser is now switched to the new NetLib library of version v4. A component update is only successful if the update does not violate version compatibility. Our framework does not explicitly support component removal, as non-referenced components (such as version v3 of NetLib here after the transition) can be garbage collected easily.

Dynamic Component Deployment With the browser now installed, it will eventually be launched. And, once the browser is launched, a user might download a dynamic plugin such as Flash. It will be desirable for the browser to use Flash without the need to terminate and restart. This action is modeled in Fig. 2(f). It is similar in spirit to component assembling, but is an action that happens only at run time. Note that a Flash plugin here is also a packaged assemblage, which should come as no surprise since it is also a well-encapsulated, independently deployable unit. This makes the action of enabling a dynamic plugin equivalent to deployment a component dynamically, an action sometimes also called *hot deployment* in the deployment community.

Formalism and Its Design Choices In our formal framework, all actions represented in Fig. 2 except the last one are represented as transitions in a Labeled Transition System (LTS). Only well-defined transitions are allowed during deployment. Dangerous operations such as manipulating low-level files and system registries arbitrarily are not allowed. In the spectrum of formal models, the LTS approach lies between purely declarative approaches and full-fledged procedural languages. A purely declarative approach has the strength of being more mathematically concise, but it is too weak to model the dynamic evolution of the deployment process as the latter is inherently dynamic. A full-fledged procedural approach is maximally expressive, but it comes with the price of complexity and greater difficulty in verification. We believe an LTS serves as a good balance between formal verification and expressiveness: the important operations of deployment are faithfully and intuitively modeled, and at the same time formal properties of the framework can be proved. In this paper we proved deployment site well-formedness preservation and version access compatibility properties. In fact, out of the traditional strength of LTS, stronger framework-specific temporal properties can also be expressed and checked, such as the fact that component update must happen before component installation.

All LTS transitions are independent of the component implementation and are completely language-neutral. As for dynamic component deployment, since it is fundamentally an operation after the application is executed, it is modeled as a reduction step within a minimally defined programming language. Indeed, since dynamic deployment fundamentally relies on dynamic linking/loading, some execution model must be assumed. Modeling it as a language expression makes it programmable, so that the Browser developer has the freedom of trigger dynamic deployment at any point in the program.

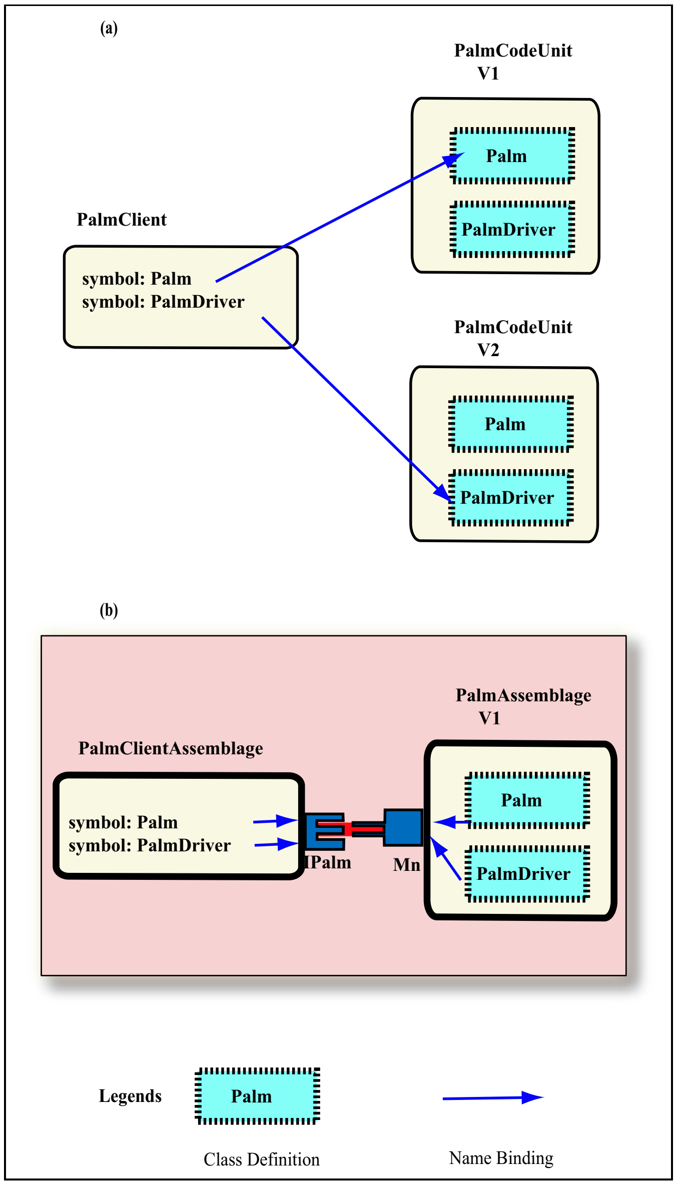


Figure 3. The Need for Interfaces (a) The Problematic Case (b) The Assemblage Solution

2.3 Design Issues

This section attempts to elucidate a few design decisions we made, and also introduces some advanced features of the framework. Among them, an important decision any deployment framework needs to make is the format of the deployment unit, *i.e.* the component model. In Sec. 2.1, we described our component model; you may ask, why does it look the way it is? The beginning subsections here will answer this question from the perspective of how such a model can facilitate deployment.

2.3.1 Why are Interfaces Needed for Deployment?

From the perspective of encapsulation, there is universal consensus that components should be designed with explicit interfaces. We

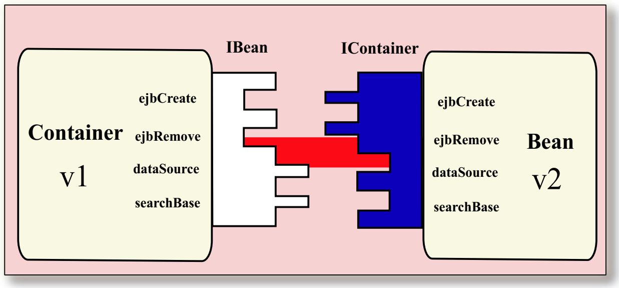


Figure 4. The Container-Bean Interaction

argue that user-definable interfaces *for the purpose of deployment* are also important, for several reasons.

First, an interface for a deployment unit is useful to define the atomic unit of versioning, *i.e.*, all dependencies in one interface must be satisfied *with the same version* of the component where the dependencies are defined. Fig. 3 demonstrates this point. It is common that a software component, say, `PalmClient`, contains two symbolic references, say, `Palm` and `PalmDriver`, defined in a different component, say, `PalmComp`. A version of `PalmDriver` should only operate upon a `Palm` of the same version. A careless design of a deployment unit could result in the dangerous binding of Fig. 3(a), and the application may exhibit erratic behaviors. This problem can be avoided by equipping the component with explicit interfaces, illustrated in Fig. 3(b). Here, dependencies are satisfied on the level of *interfaces*, and not on the level of a single symbolic reference, so that all imports in the same interface can only be satisfied by one version of another component. Equipping a mixer with an assemblage expresses the desire to define an atomic unit for versioning dependencies.

Second, explicitly declaring interfaces on deployment units also presents a cleaner solution for namespace management. This is especially true when a component will be deployed in a different site, and the component needs to rebind its dependencies. In Fig. 3, the namespace of the two mixers are merged together only when a mixing dependency is present, and there is no global binding of names. Since the development site and deployment site will often differ, global binding of names can easily lead to accidental capture/clash of names, and also make deployment sensitive to platform-specific details such as the `CLASSPATH`. Our approach here is similar to Units [19] and Knit [41] (Units' adaptation to C-like languages).

Third, interfaces also specify a component's capability for deployment. Such a declarative construct is in fact common in existing deployment frameworks, including the package specification used by Unix/Linux package managers[18, 40, 12], and the Deployment Descriptors of EJB.

2.3.2 Why are Interfaces Bidirectional?

Bidirectional dependencies between software components are a natural result of software decomposition. For instance, the CORBA CCM model allows users to define four types of interfaces where two types of them can be analogously thought of as all-export mixers in our framework and two as all-import mixers[2]. In CLI Assemblies, although no explicit bidirectional interfaces are defined, it is *de facto* equivalent to an assemblage with each defined class representing a one-export mixer, and each class dependency as a one-import mixer.

A more important question to answer is how bidirectional interfaces help model *the process of deployment*. We will use the example of EJB deployment to show the prevalence of bidirectional dependency. The primary task of an EJB deployer is to establish the interaction between the to-be-deployed bean and its container located at the deployment site. Such an interaction is fundamentally bidirectional: the bean depends on the container to provide container-specific *environment entries* to customize its behavior, while the container depends on the bean to provide call-back lifecycle-related methods [3]. Currently, EJB treats these as special cases, where environment entries are declared in an auxiliary data structure called the *deployment descriptor* with special tags that can be recognized by the container, and call-back methods are supported by adding syntactic restrictions on what a bean class has to satisfy.

Our framework can model such a case more directly, by allowing the bean to declare a *bidirectional* interface signifying how the bean will interact with the container. Fig. 4 demonstrates such an interaction. In the figure, the `Bean` can access its imports to obtain environment entry information set by the `Container`, such as the `dataSource` and `searchBase` here. The `Container` can access its imports to invoke the lifecycle-related methods of the `Bean` such as `ejbCreate` and `ejbRemove`.

The merit of our approach is our core model is smaller, environment entry access and call-back methods are not handled as special cases, and the interface for a bean's interaction with its container is structurally similar to many other EJB business interfaces. When the above `Bean` is deployed, the only thing that must be done is to set up a mixing wire between the `Bean`'s `IContainer` mixer and `Container`'s `IBean` mixer[4]. This more unified treatment gives the deployment model a more rigorous basis, and can help deployers to focus on the core aspects of the deployment task. In addition, version control can also be naturally supported, as explained in Sec. 2.3.1. For instance, a bean can specify that it must be deployed in some particular version of container. The current EJB model does not support version control explicitly, and so such a task would need special handling by the programmer.

Since `IContainer` being defined as a distinct mixer, the deployment logic and business logic, defined on other mixers, are naturally separated. With this separation in place, an interface such as `IContainer` could be written by the EJB deployment descriptor writer role.

2.3.3 Why Put Pluggers in a Deployment Unit?

For frameworks supporting hot deployment, issues such as which party should initiate the deployment and when it should be initiated are often a vendor-specific decisions. For instance, WebLogic's implementation of EJB [4] supports hot deployment via a utility `weblogic.deploy`. Indeed, since WebLogic is a container vendor, they know all the details of their container implementation, and so writing a tool of this flavor is possible. Our approach is to make minimal assumptions on the underlying software architecture, and provide a language construct which can facilitate the coding of such hot-deployment tools.

[2] The interfaces of concern are called *ports* in CORBA, and the two types corresponding to all-export mixers are Facet and EventConsumer, and the other two corresponding to all-import mixers are Receptacles and EventPublisher.

[3] Different kinds of beans need to implement different sets of call-back methods. For an entity bean for instance, they include fixed name/signature methods `ejbCreate`, `ejbPostCreate`, `ejbActivate`, `ejbPassivate`, `ejbStore`, `ejbLoad`, `ejbRemove`, `setEntityContext`, `unsetEntityContext`.

[4] Some vendors of EJB supports hot-deploying. In that case, the example will demonstrate a plugging wire; the argument stays the same in terms of bi-directional dependency.

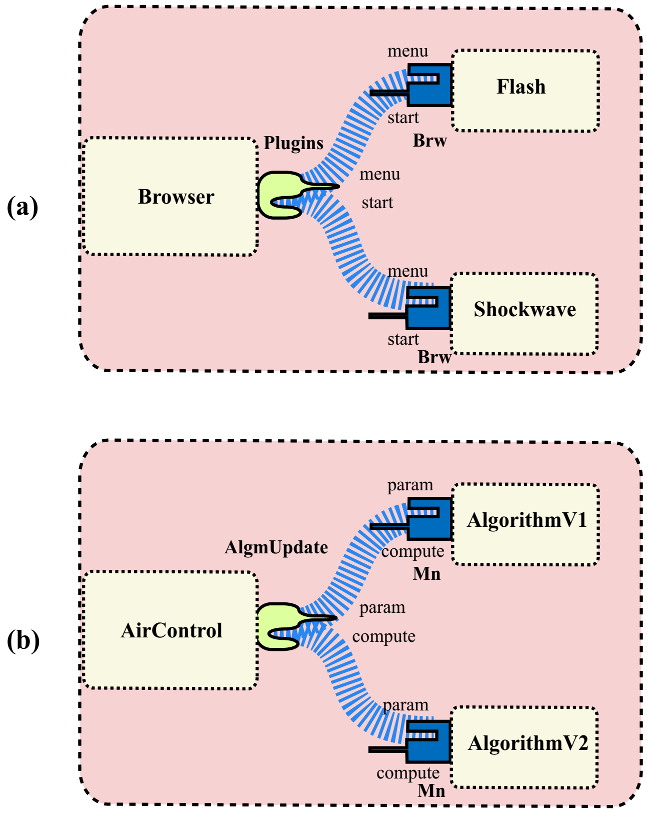


Figure 5. Plugger Rebindability (a) Two Plugins (b) Dynamic Software Updating

At a high level, placing a plugger on an assemblage indicates its ability to initiate dynamic deployment. Since each component in component-based software development is independently deployable, such an interface declaration gives the deployer a better idea what hot-deploying ability the to-be-deployed component can have. For instance, when a `Browser` component is deployed, the deployer knows the buildbox the component will enter will have some capability for hot-deployment, and can thus prepare accordingly for other orthogonal and yet important issues, such as security. The import and export declarations of the plugger can further aid the deployer by describing how the hot-deployed component needs to interact with the rest of the system, and thus what consequences the potential hot-deploying might have. For instance, if the `Plugins` plugger of the `Browser` is going to export a method called `fileWrite`, the deployer of `Browser` may have to be careful about security issues. Applets are an analogous situation if we model applet download as a hot deployment.

2.3.4 Plugger Rebindability and Dynamic Component Update

Pluggers are *rebindable* in our framework: a plugger can be wired to multiple hot-deployed components at the same time. In Fig. 2(e), we presented an application runtime with only one plugin, `Flash`. But as the application runtime evolves, it can hot-deploy another plugin, say, `ShockWave`, which would result in the situation captured by Fig. 5 (a). Since there is no way to predict in advance how many dynamic plugins a browser might hot-deploy, there is no way one can enumerate all pluggers statically. Rebindability provides a solution to this problem.

From the `Browser`'s perspective, it needs a way to interact with these two plugins individually. Since hot deployment is modeled by a language expression, we can distinguish different plugins by allowing the expression to return a *handle* that uniquely identifies the specific hot-deployed component. For instance, the setup of Fig. 5 (a) can be achieved by the following code fragment:

//hot-deploy Shockwave
$\texttt{h1} = \textbf{plugin } W_{\texttt{shockwave}} \textbf{ with } \texttt{Plugins} >> \texttt{Brw}$
//hot-deploy Flash
$\texttt{h2} = \textbf{plugin } W_{\texttt{flash}} \textbf{ with } \texttt{Plugins} >> \texttt{Brw}$
//invoke a function on Shockwave
`h1..start();`
//invoke a function on Flash
`h2..start();`

The handles `h1` and `h2` can `start` the `Shockwave` and `Flash` plugins, respectively.

The rebindability of pluggers gives our framework *de facto* support for *dynamic* software updating. Different handles to the same plugger could be viewed as different versions of the same component. For instance, suppose we have an air traffic control component enclosed in assemblage `AirControl`. The software must stay running, but it is useful to allow some of the core algorithms to periodically be tuned up and replaced with better versions. For this purpose the `AirControl` assemblage can declare a plugger `AlgmUpdate` that imports the main algorithmic function say `compute`. Invocation `recent..compute()` will always refers to the most recent algorithm if we declare a mutable field `recent` in `AirControl`, and define the update logic as follows:

//hot-deploy a new algorithm
$\texttt{p} = \textbf{plugin } W_{\texttt{newAlgm}} \textbf{ with } \texttt{AlgmUpdate} >> \texttt{M}$
// store the handle to the field
`recent = p;`

If a hot-deployed component can not be reached from all live variables it can be automatically garbage collected, removing the component from the application.

2.3.5 Singleton Mixers and Rebindable Mixers

A mixer is a *singleton* if there is at most one other mixer wired to it at any point in time. Singleton mixers are the most common form of dependency between components: for example if a Java `.class` file contains a symbolic reference to a class name, this name clearly refers to a single class definition. Most of the examples we have shown thus far use singleton mixers. The one exception is the `IBean` mixer in Fig. 4. In the EJB infrastructure, all beans in an application in fact exchange information with their container. For example, if we have a simple application involving two beans, one `Customer` and one `Agent`, both of them will need to read environment entries. The container-bean interaction thus forms a picture illustrated in Fig. 6. Note that in this case, although the import `ejbCreate` of `Container` is currently associated with multiple exports from different Beans, there is no confusion as to which one a container should call at any given time. This is because they are *call-backs* which are *never* proactively invoked by the `Container`. Our framework supports this style of mixer, and we call them *rebindable mixers*. We place the aforementioned restriction on their use: the imports are never proactively invoked by the rebindable-mixer-owning party. A precise definition of its use will be detailed in Sec. 3.4.

One way to help understand the difference between the two styles of mixers is they represent different forms of dependency.

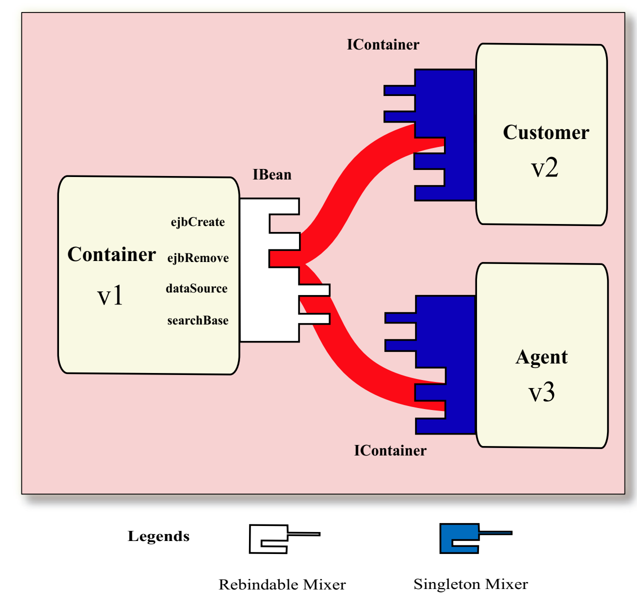


Figure 6. Rebindable Mixers: the Container-Bean Interaction Revisited

Singleton mixers represent the standard one-import-satisfied-by-one-export dependency, and rebindable mixers represent *parametric* dependency. Indeed, parameterized modules such as ML functors express parametric dependencies and they are well-known to be useful. However, existing deployment frameworks almost exclusively disregard this form of dependency. The consequence is that parameterized modules are not independent deployment units and thus cannot be updated independently in those frameworks.

2.3.6 On Hierarchical Code Composition

One design decision a typical component/module system needs to answer is: should code composition be hierarchical? In other words, does the system support operations such as `A = B + C` where `B` and `C` are components, and `A` can also be viewed as a component used for further composition, such as `E = A + D`? Hierarchical composition is in fact very common in modern component/module systems, such as the compounds in Units [19] and MJ [9].

Do assemblages allow for hierarchical code composition? The answer is yes and no: assemblages can be hierarchically composed, very much in the same way as other related work, but only the composed assemblage is considered an independent deployment unit. The idea is that on the development site, if hierarchical composition is needed, developers can freely use static linking of assemblages (a topic we have left out of this paper for brevity—interested readers can refer to [27]) to achieve this goal, and ship the compound as one deployment unit. Assemblages on the *deployment site* are not hierarchically composable.

The main reason behind this design decision is that hierarchical composition will introduce complexity if every participant is individually versioned. Take the above example. If we allow hierarchical composition on the deployment site, would we give both `A` and `E` new versions if `B` was updated? Indeed, some solution along these lines would probably be possible, but we feel the expressiveness gained is outweighed by the complexity that must be added.

It is worth pointing out that even without hierarchical code composition on the deployment site, some typical benefits of hierarchical composition are nonetheless preserved in our framework. For instance, one situation hierarchical composition is useful for is for the case that a parametric component is used more than once in forming a full application. For instance,

```
MyApplication = E + F
E = A + B
F = A + C
```

If one insists on deploying every component of `MyApplication` as an independent unit, a naive non-hierarchical treatment will have will have to deploy two distinct versions of `A`, a solution which will not be efficient or elegant. Note that this problem can be addressed in our framework through the use of rebindable mixers, in which case only one copy of `A` needs to be deployed.

2.3.7 On Sub-versioning

Our framework does not have a fixed definition of when two versions of a component are considered semantically compatible. The only constraints our framework assumes for sub-versioning are minimal structural compatibility, such as if a version has an interface named m, then any subversion should have an interface of the same name. We will formalize this notion in Sec. 3.3.1.

Refraining from giving sub-versioning a stronger definition is in fact a feature of the framework: it gives our framework more generality. In the real world, what is considered a compatible dependency in the deployment context varies: in Java dynamic class loading, it means satisfying type constraints, while in other critical systems, it means more restrictive invariants such as pre-conditions and post-conditions. Previous approaches to answering what constitutes compatible versions include behavioral subtyping [26, 30] and refactorability [2]. In our framework we assume compatibility of versions is declared by some means, *e.g.*, based on analysis results from the aforementioned systems.

3. The Formal System

In this section we make rigorous the ideas of the previous section by presenting a formal Labeled Transition System (LTS) that captures the key component interactions during the deployment process. In the following section we establish correctness properties of our formal framework.

We first define basic notations. $\overline{x}$ denotes a set $\{x_1, \ldots, x_p\}$, with empty set $\emptyset$. $|S|$ is the size of set S and $A - B = A \cap \overline{B}$. $\overline{x \mapsto y}$ is used to denote a mapping function Mp mapping x_1 to $y_1, \ldots,$ x_p to y_p where $\{x_1, \ldots x_p\}$ is the domain of the function, denoted as $\mathsf{dom}(Mp)$. We also write $Mp(x_1) = y_1, \ldots Mp(x_p) = (y_p)$. We use $\emptyset_M$ to denote an empty map. $\mathsf{dom}(\emptyset_M) = \emptyset$. We write $Mp[x \mapsto y]$ as a standard map update. Mp and $Mp[x \mapsto y]$ are identical except that $Mp[x \mapsto y]$ maps x to y.

3.1 Definitions

Fig. 7 defines the grammatic structure of the buildbox, and Fig. 8 gives auxiliary functions that are used in the framework. We now work through these definitions step by step.

Buildbox A buildbox (Abb) is represented as a graph, where the nodes (N) are components and the edges (K) are the wires between them. This is in sync with the general notion of component-based software development: applications are components wired together. Each node might contain multiple versions (V) of components of the same name (a), and each version of the component is identified by a version identifier (α). Instead of giving a concrete representation of a version identifier, we only require that version identifiers are unique to a degree that accidental clash can be avoided. In real-world systems, a version identifier could be realized as a meaning-

Abb	$::=$	$\langle N; K\rangle$	*application buildbox*
N	$::=$	$\overrightarrow{a \mapsto \langle V; R\rangle}$	*buildbox nodes*
K	$::=$	$\overline{a.\alpha.m \smile b.\beta.n}$	*buildbox wires*
V	$::=$	$\overrightarrow{\alpha \mapsto \langle A; C\rangle}$	*component versions*
R	$::=$	$\overline{\alpha \leq \beta}$	*compatibility relation*
A	$::=$	$\langle M; L\rangle$	*assemblage*
C	$::=$	$\overline{m \smile b.\beta.n}$	*dependency constraints*
M	$::=$	$\overrightarrow{m \mapsto \langle kd; I; E\rangle}$	*interfaces*
I	$::=$	$\overline{k}$	*imports*
E, L	$::=$	$\overrightarrow{k \mapsto B}$	*exports, locals*
B			*code block, see Fig. 10*
W	$::=$	$\langle a; \alpha; A; R; C\rangle$	*packaged assemblage*
a, b, c			*component name*
m, n			*interface name*
k			*feature name*
α, β, γ			*version identifier*
kd	$\in$	$\{$ `smixer`, `rmixer`, `plugger` $\}$	*interface kind*

Figure 7. Definition: Buildbox

ful number appended to a verifiable signature or hash. Other concrete examples along the same lines are the GUID's of COM and the strong names of CLI Assemblies.

Assemblage Recall from earlier discussion that components in our framework are realized as *assemblages*. Each assemblage is formally represented as a list of interfaces M and local internal definitions L. Interfaces can either be singleton mixers (`smixer`), rebindable mixers (`rmixer`) or pluggers (`plugger`). Each interface is bidirectional, with its imports (I) and exports (E) declared. Note that for now, we abstractly represent items imported and exported through interfaces as named "features"; the concrete definition of a code block B is not given until Sec. 3.4. Before then, all our discussion is independent of the actual component implementation, so it is language-neutral and platform-neutral.

Wire A wire is represented as $a.\alpha.m \smile b.\beta.n$ (see definition of K in Fig. 7). It means that component a version α's interface m is wired to component b version β's interface n. Note the wiring information is precise as to what version of the node is wired. We equate $a.\alpha.m \smile b.\beta.n$ and $b.\beta.n \smile a.\alpha.m$. On the syntactical level, we do not distinguish mixing wires and plugging wires. This task can be easily achieved given the kinds of interfaces of m and n.

Packaged Assemblage A packaged assemblage (W) is the shipped form of component that can be deployed at a deployment site, and so it serves as the bridge between the development site and the deployment site. It contains the name (a), version identifier (α) of the assemblage to be shipped (A), together with its dependency constraints (C) and compatibility information (R). We will next explain dependency constraints and postpone the elaboration of R until Sec. 3.3.1.

Dependency Constraint A dependency constraint $m \smile b.\beta.n$ in C indicates that the current assemblage's m interface should be satisfied by an interface named n of assemblage named b of version β. Note that C is both part of a packaged assemblage (W) and part of an already installed node (V). One might be tempted to think that dependency constraints should be checked against at installation time and declared satisfied or failed, so that installed nodes do not need to carry such information any more. But, this is not realistic because assemblages may be mutually dependent: solving all constraints might end up leading to neither assemblage being installed. In fact, since the buildbox at deployment site is an evolving system, there is nothing wrong with recording some of the unsatisfied constraints, the C found in the definition of V. In addition, recording the constraints can also help with future updates, ruling out those operations violating the dependency constraints.

3.2 Buildbox Well-formedness and Closure

One important property of our framework is that no matter how the buildbox evolves, it stays "well-formed" (see Sec. 4). We define the notion of well-formedness in Fig. 8. A buildbox Abb is well-formed (wffAbb) if all versions of all nodes, wires and version compatibility settings are well-formed, represented by wffVer, $\mathsf{wffWire}$ and wffR respectively. A component of a specific version is well-formed if there is no conflict between its dependency constraints and the rest of the buildbox ($\mathsf{noconflictC}$), and no import in a singleton mixer is satisfied by more than one export ($\mathsf{noconflictImp}$), as explained earlier in Sec. 2.3.5. Wire well-formedness ($\mathsf{wffWire}$) depends on the correct matching of interfaces: on a per-wire basis, imports in the interface on one end of the wire must be satisfied by exports on the other end. A well-formed wire also should not connect two rebindable mixers, because otherwise neither party would initiate any invocation and is *de facto* futile (recall the callback nature of rebindable mixers, explained in Sec. 2.3.5). The last constraint precludes two pluggers from being connected together: pluggers only declare what plugins may be plugged into the current runtime. We will explain wffR in Sec. 3.3.1 below.

Note that buildbox well-formedness itself does not require all imports of installed assemblages to be satisfied at deployment time, nor does it require all dependency constraints associated with a node version to be satisfied, as we explained in Sec. 3.1. These extra conditions are captured by definition $\mathsf{closedAbb}$.

3.3 The Labeled Transition System

The evolution of a buildbox is defined as a series of transitions in the LTS of Fig. 9. This LTS is the formalization of the software development and deployment cycle, and represents the general and formal version of Fig. 2. All rules excepting (**ship**) are of the form $Abb \xrightarrow{l} Abb'$, denoting a buildbox transition from state Abb to Abb', via operation l. Each LTS rule defines a legal operation that can be performed by a deployer (or by a role appropriate for the operation), with values in the label being the information provided by the role. The pre-conditions of each rule define what a "good" operation is. Multi-step evolution $Abb \xrightarrow{l}_* Abb'$ is defined as the transitive closure over the single-step transition. The (**ship**) rule is slightly different because we only care about its output (packaged assemblages).

The focus of this section is on how individual components are developed and deployed (Sec. 3.3.1). At this point we assume for simplicity that there is only one buildbox deployed in the whole universe. In Sec. 3.3.4, we show how a universe with multiple buildboxs can be modeled in terms of this simpler notion. Sec. 3.3.2 discusses the impact of our framework on distributed applications.

$$
\begin{array}{rcl}
\mathsf{depends}(a, \alpha, m, \langle N; K \rangle) & \stackrel{\text{def}}{=} & \{ b.\beta.n \mid \; a.\alpha.m \smile b.\beta.n \in K, \; \mathsf{interface}(a, \alpha, m, Abb) = \langle kd; I; E \rangle, \; I \neq \emptyset \} \\
\mathsf{interface}(a, \alpha, m, \langle N; K \rangle) & \stackrel{\text{def}}{=} & M(m) \text{ where } N(a) = \langle V; R \rangle, V(\alpha) = \langle A; C \rangle, A = \langle M; L \rangle \\
\mathsf{interfaceKind}(a, \alpha, m, Abb) & \stackrel{\text{def}}{=} & kd \text{ where } \mathsf{interface}(a, \alpha, m, Abb) = \langle kd; I; E \rangle \\
\mathsf{newer}(R, \alpha) & \stackrel{\text{def}}{=} & \{\alpha\} \cup \{ \beta \mid (\alpha \leq \beta) \in R \} \\
\mathsf{newer}(R, \mathtt{any}) & \stackrel{\text{def}}{=} & \{ \beta \mid (\alpha \leq \beta) \in R \} \cup \{ \alpha \mid (\alpha \leq \beta) \in R \} \\
\mathsf{newest}(\Theta, R, \alpha) & \stackrel{\text{def}}{=} & \beta, \text{where } nset = \mathsf{newer}(R, \alpha) \cap \Theta, \; \beta \in nset, \; \forall \beta' \in nset.(\beta' \neq \beta \implies (\beta' \leq \beta) \in R) \\
\mathsf{newest}(\Theta, R, \mathtt{any}) & \stackrel{\text{def}}{=} & \beta, \text{where } nset = \mathsf{newer}(R, \mathtt{any}) \cap \Theta, \; \beta \in nset, \; \forall \beta' \in nset.(\beta' \neq \beta \implies (\beta' \leq \beta) \in R) \\
\mathsf{newest}(\Theta, R, \alpha) & \stackrel{\text{def}}{=} & \text{undefined otherwise} \\
\mathsf{bestChoice}(\langle N; K \rangle, b, \beta) & \stackrel{\text{def}}{=} & \beta \text{ if } N(b) = \langle V; R \rangle, \beta \in \mathsf{dom}(V) \\
\mathsf{bestChoice}(\langle N; K \rangle, b, \beta) & \stackrel{\text{def}}{=} & \beta' \text{ if } \;\; \beta' = \mathsf{newest}(\mathsf{dom}(V), R, \beta), N(b) = \langle V; R \rangle, \beta \notin \mathsf{dom}(V) \\
\mathsf{bestChoice}(\langle N; K \rangle, b, \beta) & \stackrel{\text{def}}{=} & \text{undefined otherwise} \\
\mathsf{wrap}(a, \alpha, \Delta, Abb) & \stackrel{\text{def}}{=} & \displaystyle\bigcup_{i \in [1..p]} \mathsf{wrapE}(a, \alpha, m_i, \Delta, Abb) \\
 & \text{if} & \;\; Abb = \langle N; K \rangle, \; \{m_1, \ldots, m_p\} = \{ m \mid (a.\alpha.m \smile b.\beta.n) \in K \} \\
\mathsf{wrapE}(a, \alpha, m, \Delta, Abb) & \stackrel{\text{def}}{=} & \{ m \smile b.\beta.n \} \\
 & \text{if} & m \in \Delta, \; \{b.\beta.n\} = \mathsf{depends}(a, \alpha, m, Abb), \; \mathsf{interfaceKind}(a, \alpha, m, Abb) = \mathtt{smixer} \\
\mathsf{wrapE}(a, \alpha, m, \Delta, Abb) & \stackrel{\text{def}}{=} & \{ m \smile b.\mathtt{any}.n \} \\
 & \text{if} & m \notin \Delta, \; \{b.\beta.n\} = \mathsf{depends}(a, \alpha, m, Abb), \; \mathsf{interfaceKind}(a, \alpha, m, Abb) = \mathtt{smixer} \\
\mathsf{wrapE}(a, \alpha, m, \Delta, Abb) & \stackrel{\text{def}}{=} & \emptyset \text{ otherwise} \\
\langle M_1; L_1 \rangle <:_{\mathtt{A}} \langle M_2; L_2 \rangle & \stackrel{\text{def}}{=} & \forall m \in \mathsf{dom}(M_1).(m \in \mathsf{dom}(M_2) \wedge M_1(m) <:_{\mathtt{M}} M_2(m)) \\
\langle kd_1; I_1; E_1 \rangle <:_{\mathtt{M}} \langle kd_2; I_2; E_2 \rangle & \stackrel{\text{def}}{=} & (kd_1 = kd_2) \wedge (\forall k \in \mathsf{dom}(E_1).k \in \mathsf{dom}(E_2)) \wedge (\forall k \in I_2.k \in I_1)
\end{array}
$$

Well-formedness and Closure Related Predicates

$$
\begin{array}{rcl}
\mathsf{wffAbb}(\langle N; K \rangle) & \stackrel{\text{def}}{=} & (\forall a \in \mathsf{dom}(N).(N(a) = \langle V; R \rangle \wedge (\forall \alpha \in \mathsf{dom}(V).\mathsf{wffVer}(a, \alpha, Abb)))) \wedge \\
 & & (\forall a \in \mathsf{dom}(N).(N(a) = \langle V; R \rangle \wedge \mathsf{wffR}(V, R))) \wedge \\
 & & (\forall lk \in K.\mathsf{wffWire}(lk, Abb)) \\
\mathsf{closedAbb}(\langle N; K \rangle) & \stackrel{\text{def}}{=} & (\forall a \in \mathsf{dom}(N).(N(a) = \langle V; R \rangle \wedge (\forall \alpha \in \mathsf{dom}(V).\mathsf{closedVer}(a, \alpha, Abb)))) \\
\mathsf{wffVer}(a, \alpha, Abb) & \stackrel{\text{def}}{=} & \mathsf{noconflictC}(a, \alpha, Abb) \wedge \mathsf{noconflictImp}(a, \alpha, Abb) \\
\mathsf{closedVer}(a, \alpha, Abb) & \stackrel{\text{def}}{=} & \mathsf{satisfiedC}(a, \alpha, Abb) \wedge \mathsf{satisfiedImp}(a, \alpha, Abb) \\
\mathsf{noconflictImp}(a, \alpha, Abb) & \stackrel{\text{def}}{=} & \forall m.\exists I.\exists E.(\mathsf{interface}(a, \alpha, m, Abb) = \langle \mathtt{smixer}; I; E \rangle \\
 & & \qquad \implies \mid \mathsf{depends}(a, \alpha, m, Abb) \mid <= 1) \\
\mathsf{satisfiedImp}(a, \alpha, Abb) & \stackrel{\text{def}}{=} & \forall m.\exists I.\exists E.((\mathsf{interface}(a, \alpha, m, Abb) = \langle \mathtt{smixer}; I; E \rangle) \wedge (I \neq \emptyset) \\
 & & \qquad \implies \mid \mathsf{depends}(a, \alpha, m, Abb) \mid = 1) \\
\mathsf{noconflictC}(a, \alpha, Abb) & \stackrel{\text{def}}{=} & \forall (m \smile b.\beta.n \in C) \\
 & & \;\; a.\alpha.m \smile b'.\beta'.n' \in K \implies N(b') = \langle V'; R' \rangle \wedge \beta' \subseteq \mathsf{newer}(R', \beta) \wedge n = n' \wedge b = b' \\
 & & \text{where } Abb = \langle N; K \rangle, N(a) = \langle V; R \rangle, V(\alpha) = \langle A; C \rangle \\
\mathsf{satisfiedC}(a, \alpha, Abb) & \stackrel{\text{def}}{=} & \mathsf{noconflictC}(a, \alpha, Abb) \; \wedge \forall (m \smile b.\beta.n \in C).\exists \beta'.(a.\alpha.m \smile b.\beta'.n \in K) \\
 & & \text{where } Abb = \langle N; K \rangle, N(a) = \langle V; R \rangle, V(\alpha) = \langle A; C \rangle \\
\mathsf{wffWire}(a.\alpha.m \smile b.\beta.n, Abb) & \stackrel{\text{def}}{=} & (I_1 \in \mathsf{dom}(E_2)) \;\wedge\; (I_2 \in \mathsf{dom}(E_1)) \wedge \\
 & & (\neg\, (kd_1 = \mathtt{rmixer} \wedge kd_2 = \mathtt{rmixer})) \;\wedge\; (\neg\, (kd_1 = \mathtt{plugger} \wedge kd_2 = \mathtt{plugger})) \\
 & & \text{where } \mathsf{interface}(a, \alpha, m, Abb) = \langle kd_1; I_1; E_1 \rangle, \; \mathsf{interface}(b, \beta, n, Abb) = \langle kd_2; I_2; E_2 \rangle \\
\mathsf{wffR}(V, R) & \stackrel{\text{def}}{=} & \mathsf{partialOrder}(R) \wedge \\
 & & \forall (\alpha \leq \beta) \in R.((V(\alpha) = \langle A_1; C_1 \rangle) \wedge V(\beta) = \langle A_2; C_2 \rangle) \implies A_1 <:_{\mathtt{A}} A_2)
\end{array}
$$

Figure 8. Definition: Auxiliary Functions and Predicates

(build)

$$\frac{\alpha \text{ is fresh} \qquad Abb \xrightarrow{\textbf{install } \langle a;\alpha;A;\emptyset;C\rangle} Abb'}{Abb \xrightarrow{\textbf{build } a,A,C} Abb'}$$

(assemble)

$$\frac{\begin{array}{c} Abb = \langle N; K\rangle \\ Abb' = \langle N; K \cup K'\rangle \\ \forall (a.\alpha.m \smile b.\beta.n) \in K'.(\mathsf{wffWire}(a.\alpha.m \smile b.\beta.n, Abb') \wedge \mathsf{wffVer}(a, \alpha, Abb') \wedge \mathsf{wffVer}(b, \beta, Abb')) \end{array}}{Abb \xrightarrow{\textbf{assemble } K'} Abb'}$$

(ship)

$$\frac{\mathsf{closedVer}(a, \alpha, Abb) \qquad Abb = \langle N; K\rangle \qquad N(a) = \langle V; R\rangle \qquad V(\alpha) = \langle A; C'\rangle \qquad C = \mathsf{wrap}(a, \alpha, \Delta, Abb)}{Abb \xrightarrow{\textbf{ship } a,\alpha,\Delta}_S \langle a; \alpha; A; R; C\rangle}$$

(install)

$$\frac{\begin{array}{c} Abb \xrightarrow{\textbf{addN } \langle a;\alpha;A;R;C\rangle}_G Abb'' \\ Abb'' \xrightarrow{\textbf{assemble } K'} Abb' \\ K' = \{a.\alpha.m \smile b.\beta.n \mid m \smile b.\beta'.n \in C, \beta = \mathsf{bestChoice}(Abb'', b, \beta')\} \end{array}}{Abb \xrightarrow{\textbf{install } \langle a;\alpha;A;R;C\rangle} Abb'}$$

(update)

$$\frac{\begin{array}{c} Abb = \langle N; K\rangle \qquad N(b) = \langle V; R\rangle \\ \beta' \in \mathsf{dom}(V) \qquad \langle N; K - K_0\rangle \xrightarrow{\textbf{assemble } K_1} Abb' \\ K_0 = \{a.\alpha.m \smile b.\beta.n \mid a.\alpha.m \smile b.\beta.n \in K \text{ for some } a, \alpha, m, n\} \\ K_1 = \{a.\alpha.m \smile b.\beta'.n \mid a.\alpha.m \smile b.\beta.n \in K \text{ for some } a, \alpha, m, n\} \end{array}}{Abb \xrightarrow{\textbf{update } b,\beta,\beta'} Abb'}$$

(set-compatible)

$$\frac{N(b) = \langle V; R\rangle \qquad N' = N[b \mapsto \langle V; R \cup_{\mathtt{rm}} \{\beta \leq \beta'\}\rangle] \qquad \mathsf{wffR}(V, \{\beta \leq \beta'\})}{\langle N; K\rangle \xrightarrow{\textbf{setComp } b,\beta,\beta'} \langle N'; K\rangle}$$

(auxiliary - add node)

$$\frac{a \notin \mathsf{dom}(N) \qquad V = \alpha \mapsto \langle A; C\rangle}{\langle N; K\rangle \xrightarrow{\textbf{addN } \langle a;\alpha;A;R;C\rangle}_G \langle N[a \mapsto \langle V; R \cup_{\mathtt{rm}} \emptyset\rangle]; K\rangle}$$

(auxiliary - add version)

$$\frac{N(a) = \langle V; R\rangle \qquad V' = V[\alpha \mapsto \langle A; C\rangle] \qquad \mathsf{wffR}(V', R' \cup_{\mathtt{rm}} \emptyset) \qquad \alpha \notin \mathsf{dom}(V)}{\langle N; K\rangle \xrightarrow{\textbf{addN } \langle a;\alpha;A;R';C\rangle}_G \langle N[a \mapsto \langle V'; R \cup_{\mathtt{rm}} R'\rangle]; K\rangle}$$

Figure 9. Application Evolution: the LTS Rules

3.3.1 Component-Level Deployment

Building a Component Building a component from scratch is covered by transition rule (**build**). It is an operation typically performed by a role different from a deployer. For instance in the EJB specification, a role named Bean Provider is defined for this responsibility. It first involves specifying the component with a name (a), and its interfaces and the implementation (A). Note that a component almost always depends on other components, since libraries are themselves components. Here we allow a component to specify how such dependencies should be satisfied, in dependency constraints C. Such a mechanism is not absolutely necessary, since a separate rule (**assemble**) handles the wiring of components together, but we feel it is closer to real-world development practice when library component dependencies are immediately resolved at build time. A fresh version identifier α is assigned to each build.

Assembling Components To form an application, components need to be assembled together. This process in the LTS is modeled by (**assemble**). The parameter of the operation is a set of wiring specifications (K'). The assembling process can only succeed if the well-formedness of the buildbox is not undermined. For instance, wiring two mixers with unmatched import-export pairs will fail to transition. As another example, if one component contains a constraint saying its mixer m can only be satisfied by mixer n of component b version β or a compatible one, the transition will fail if the attempt is to wire it with component b version β' where β' is not compatible with β.

The (**assemble**) operation is typically performed by the application developer: in the specification of EJB, this is mostly the responsibility of the Assembler role. Some component dependencies can only be solved at deployment time, for example EJB environment entries; so, these operations can also be performed by the Deployer.

Shipping a Component Shipping a component is defined in **(ship)**, and involves specifying the name a and version α of the component to be shipped. Users can also specify a set of interface names of the component (Δ), expressing the desire that the version information of the dependency to that interface be recorded as a dependency constraint in the packaged assemblage (the C part). Such a mechanism gives users the freedom to selectively record version information. For those interfaces not included in Δ but still dependent on other components, a special version identifier any is recorded. Dependency constraints will affect the version checking when the component is later deployed.

The key task of shipping is to create a packaged assemblage, in the form as defined in Fig. 7. Preparing the dependency constraints is the core of this process; this is modeled by the wrap function defined Fig. 8. What makes wrap a non-trivial task is that not every wire to an assemblage interface is necessarily a dependency; for instance an all export mixer does not depend on assemblages wired to it (it is the other way around). The essence of the wrap function is to precisely identify the real dependencies. Specifically, it only garners constraints from the wires hooked up to its singleton mixers' imports. No wire to a rebindable mixer will generate a dependency constraint: rebindable mixers by nature is *passive*, *i.e.* the assemblage with the mixer can never *proactively* access the code defined in the assemblage it is wired with. (Otherwise there would be ambiguity as to which dependency is to be used when an import is accessed.) In terms of dependency, it is the other party that depends on it but not *vice versa*.

Setting Component Version Compatibility As we have explained in Sec. 2.3.7, our framework provides a user interface for declaring two component versions are backward compatible. In our transition system, this is modeled by rule (**set-compatible**), and operation **setComp** b β β' sets version β' of the component named b to be backward compatible with version β. To explain this rule, we first need to explain how backward compatibility information is recorded in an buildbox, the R of Fig. 7. R is a set of version identifier pairs of the form $\beta \leq \beta'$, meaning version β' is backward compatible with version β. Sometimes we also say β' is *newer* than β.

We now give a few formal definitions related to R, including "merging". R is well-formed, denoted as $\mathsf{wffR}(V, R)$ (see Fig. 8), iff R is a partial order (the definition of $\mathsf{partialOrder}(R)$ is standard) and each pair of versions declared as compatible by R also satisfies minimal structural compatibility (the definition of $<:_{\mathtt{A}}$), as explained in Sec. 2.3.7. We define merging $R_1 \cup_{\mathtt{rm}} R_2$ as the transitive closure over relation set $R_1 \cup R_2$, and it is undefined if R_1 or R_2 is not a partial order, or the result of transitive closure is not a partial order. When R is a partial order, function $\mathsf{newer}(R, \alpha)$ computes the set of versions (identifiers to be precise) that are backward compatible with α, including α itself. When α is the special value any, the function degenerates into enumerating all version identifiers that appear in R. Similarly, partial function $\mathsf{newest}(\Theta, R, \alpha)$ finds out a version included in Θ that is the "newest" subversion to α according to R.

(**set-compatible**) simply records the newly declared compatibility information. It fails if the merging results in a non-partial order (the $\cup_{\mathtt{rm}}$ part), or β' does not meet the minimal structural requirement to be compatible with β (the wffR part).

Installing a Component Component installation is modeled by the (**install**) rule. It has two key tasks: 1) adding the component itself to the buildbox (the **addN** auxiliary transition) 2) adding wires (the **assemble** transitiion). The core part of the (**install**) rule is to select a compatible version that resolves dependencies. This is achieved by a partial function $\mathsf{bestChoice}$, with its self-explanatory definition found in Fig. 8.

The installation rule reflects a "best-effort" strategy: it tries its best to wire the installed component up with the rest of the buildbox, based on the information it has on what dependencies are expected (C). However, due to reasons such as cyclic dependency (see Sec. 3.1), it is not always possible to find a total order where all dependencies have already been present when a component is installed. What (**install**) can achieve, in an intuitive way, is *whenever it sets up a wire, it is guaranteed to be a "good" one*, *i.e.* without undermining the well-formedness of the buildbox. In the cyclic dependency case where components a and b depend on each other, the unresolved constraints when installing a will be satisfied later when b is installed. This late satisfaction will not result in dangling dependencies at run time, since every execution will start with a check to make sure the buildbox is closed; see Sec. 3.4 for details.

Explicitly declaring compatibility relations in the fashion of (**set-compatible**) can be a labor for deployers, so our transition system also allows packaged assemblages to carry the compatibility relation accumulated on the development site over to the deployment site. At the deployment site, the compatibility relation carried over from the development site is "merged" with the relation recorded by the node on the deployment site, an operation realized by the (**auxiliary - add version**) rule of Fig. 9. Note that we do not require all version identifiers appearing in R to have their corresponding assemblage installed in the node, and this is in fact crucial to support compatible installation: an installation may indicate it depends on a component of version α, but a deployment site may not have the component of the same version. Now, as long as the deployment site can recognize version identifier α and find some compatible version of it available there, installation can still proceed.

Updating a Component The (**update**) rule models the case where a component b would like to update itself from version β to β'. This process is modeled by first removing the existing wires and then add new wires in. Note that such a rule depends on (**assemble**), which in turn contains well-formedness checks to ensure updating does not sacrifice environment well-formedness. The update operation does not require the updating version to be a subversion of the updated version. Indeed, all that matters is switching from one version to another would not violate any dependency contraint for any involved party, which is guaranteed by (**assemble**). This more relaxed treatment is in sync with real-world scenarios: not all updates are upgrades.

Removing a Component Components are automatically garbage collected when a version is not wired to the rest of the buildbox. The criterion for garbage collection is a very simple wire-counting based on the following predicate:

$$\mathsf{collectable}(a, \alpha, \langle N; K\rangle) \quad \overset{\text{def}}{=} \quad (a.\alpha.m \smile b.\beta.n) \notin K \text{ for all } m, b, \beta, n$$

3.3.2 Distributed Deployment

Software components of a buildbox are not necessarily located on one physical network node. Distributed deployment has been a focus of CORBA, and in the EJB case, all beans that can be accessed by its naming service JNDI may be distributed across different network locations.

This framework does not attempt to solve all issues related to distributed deployment, but we point out that the framework itself can serve as a basis for distributed deployment. The buildbox represented as a graph $\langle N; K\rangle$ may have nodes in N located in different places, and the K can represent the distributed wiring amongst nodes. The idea is that when an application is developed and shipped, all its components may end up being deployed in different locations, and their version compatibility is still preserved in a distributed manner. For instance, when one component is updated via (**update**), the wires being updated could very well link to some component in a different location, so that when components in other locations later have access to the updated component, the new (and compatible) version will be used.

3.3.3 Deployment in Batch Mode

Up to now our discussion has focused on how deployment can be performed *at the component level*. This is because software components by definition are the deployment units, and atomic operations should be defined at this level. In addition, some operations such as update are fundamentally component-level operations. In the real world, some operations might be more commonly used on the application level, such as shipping and installation. They do not introduce extra difficulty however: when shipping an application, it is equivalent to shipping its components one by one by repeatedly applying the (**ship**) rule, and when installing an application, it is equivalent to installing its components one by one by (**install**). Note that because our framework does not require all dependencies are satisfied all at once at installation time, the ordering of which assemblage should be installed first is not important.

3.3.4 Multiple BuildBoxes

The LTS rules address how an assemblage operates in a single buildbox scenario. A software environment composed of multiple buildboxes can easily be built up on top of this. Let us consider component installation, for example. In a multiple buildbox scenario, when an assemblage private to a buildbox is to be installed, the LTS **install** operation specific to that buildbox will be triggered and only this buildbox will evolve. When an assemblage functioning like a library is to be installed, it can be conceptually viewed as installing the same assemblage in all buildboxes. Since the majority of the transitions for the multiple buildbox scenario is to delegate them to a single buildbox scenario, we do not present these rules in this presentation. Interested readers can refer to the long version for details [29]. The principle behind this treatment is a conceptual level of buildbox isolation, which simplifies our formal framework without loss of generality.

3.4 Execution

Thus far we have made no assumptions about the code inside an assemblage, but the aim of deployment is to eventually run the application. In this section, we construct a very simple assemblage realization, by taking code blocks to be functions, illustrated in Fig. 10.

Since we are only interested in how names are linked and how hot deployment is accomplished, the expressions e in Fig. 10 are very simple. Besides the **plugin** expression explained earlier, it supports $m :: k(e)$ to invoke a function k defined in singleton mixer m, $k(e)$ to invoke a callback function k defined in the current rebindable mixer, $:: k(e)$ to invoke a local function, and $e..m(e)$ to invoke a function defined in the plugger. Values v are either constants, or plugin handles: $a.\alpha.m \rhd b.\alpha.n$ is a first-class value denoting the plugging from plugger m of node a's version α, to the mixer n of node b's version β. We leave out features unrelated to deployment from the expressions, for instance, mutable state. Readers interested in a more complete language specification can refer to [27].

Execution starts by applying rule (**execute**). It simply looks for a mixer called `Main`, and invokes an export function called `main`. Predicate $\mathsf{closedApp}$ disallows dangling imports and makes sure all dependency constraints are satisfied. Fig. 10 defines the small-step reduction relation $App, Stk, e \rightarrow App', Stk', e'$, which executes the expressions inside an assemblage. Stk keeps track of the function invocations, based on where the current function is defined. A top element $\langle a; \alpha; m\rangle$ means the current expression is defined in a function inside mixer/plugger m of assemblage named a, version α. m can also be given a special value `local`, in which case the function is defined as a local function: here we are only interested in the scope up to the level of mixer/plugger (or whether it is a local function), but not which function is defined. The definitions of expressions, values and stacks are given in Fig. 10. $\mathsf{top}(Stk)$ returns the top element of the stack. Multi-step reduction $App, Stk, e \rightarrow_* App', Stk', e'$ is defined as the transitive closure over the small-step reduction.

3.4.1 Dynamic Plugins and Hot Deploying

The process of hot deploying is illustrated by (**plugin**). Note that a dynamic plugin in a deployment framework represents an independent deployment unit, and hence it is represented as a packaged assemblage (W), with its dependency information packed up. In our framework, built-in version control for dynamic plugins is provided, and it is unified with version control for non-hot deployment: notice the structural form of a dynamic plugin is no difference from regular packaged assemblages used for non-hot deployment.

A plugging wire is established between the initiating assemblage runtime's plugger and the mixer of the plugee (see the **assemble** part of the rule). The **assemble** rule will also make sure all imports from one party are satisfied by exports from the other party: no dangling import is possible. The expression returns a plug handle which exactly records the information of the plugging wire.

As the rule suggests, hot deploying needs to *install* the dynamic plugin (see the **install** part of the rule). This is obvious; hot-deploying is installation at runtime. The not-so-obvious issue is that depending on the plugger to export to the dynamic plugin all functionalities is unrealistic in the real world. Take EJB for in-

App	$::=$	Abb	*application runtime*
B	$::=$	$\lambda x.e$	*code block*
e	$::=$	$i \mid m :: k(e) \mid k(e) \mid :: k(e) \mid e..m(e) \mid e; e$	*expression*
	$\mid$	$\textbf{plugin}\ W\ \textbf{with}\ m >> n \mid \textbf{return}\ e$	
v	$::=$	$i \mid a.\alpha.m \rhd b.\beta.n$	*value*
$\mathbf{E}$	$::=$	$[] \mid m :: k(\mathbf{E}) \mid :: k(\mathbf{E}) \mid k(\mathbf{E}) \mid \mathbf{E}..k(e) \mid v..k(\mathbf{E})$	*evaluation context*
	$\mid$	$\mathbf{E}; e \mid v; \mathbf{E} \mid \textbf{return}\ \mathbf{E}$	
Stk	$::=$	$s \diamond \mathit{Stk} \mid s$	*stack*
s	$::=$	$\langle a; \alpha; m\rangle \mid \langle a; \alpha; \texttt{local}\rangle$	*stack frame*
i			*integer*

(execute)

$$\frac{\mathsf{closedAbb}(\mathit{Abb}) \qquad \mathsf{interface}(a, \alpha, \texttt{Main}, \mathit{Abb}) = \langle \texttt{rmixer}; \emptyset_M; \texttt{main} \mapsto \lambda x.e\rangle \qquad \mathit{App} = \mathit{Abb}}{\mathit{Abb}, \textbf{execute}\ a, \alpha \xrightarrow{\text{exe}} \mathit{App}, \langle a; \alpha; \texttt{main}\rangle, e}$$

(plugin)

$$\frac{\begin{array}{c} \mathsf{top}(\mathit{Stk}) = \langle a; \alpha; m'\rangle \qquad W = \langle b; \beta; \langle M; L\rangle; R; C\rangle \\ \mathit{App} \xrightarrow{\textbf{install}\ W} \mathit{App}' \\ \mathit{App}' \xrightarrow{\textbf{assemble}\ \{a.\alpha.m \smile b.\beta.n\}} \mathit{App}'' \\ \mathsf{interfaceKind}(a, \alpha, m, \mathit{App}) = \texttt{plugger} \qquad \mathsf{closedVer}(b, \beta, \mathit{App}'') \end{array}}{\mathit{App}, \mathit{Stk}, \textbf{plugin}\ W\ \textbf{with}\ m >> n \to \mathit{App}'', \mathit{Stk}, a.\alpha.m \rhd b.\beta.n}$$

(handle invoke - 1)

$$\frac{\mathsf{interface}(a, \alpha, m, \mathit{App}) = \langle \texttt{plugger}; I; E\rangle \qquad k \notin I \qquad E(k) = \lambda x.e}{\mathit{App}, \mathit{Stk}, (a.\alpha.m \rhd b.\beta.n)..k(v) \to \mathit{App}, \langle a; \alpha; m\rangle \diamond \mathit{Stk}, e[v/x]}$$

(handle invoke - 2)

$$\frac{\begin{array}{c} \mathsf{interface}(a, \alpha, m, \mathit{App}) = \langle \texttt{plugger}; I; E\rangle \\ k \in I \qquad \mathsf{interface}(b, \beta, n, \mathit{App}) = \langle \texttt{smixer}; I'; E'\rangle \text{ or } \langle \texttt{rmixer}; I'; E'\rangle \\ E'(k) = \lambda x.e \end{array}}{\mathit{App}, \mathit{Stk}, (a.\alpha.m \rhd b.\beta.n)..k(v) \to \mathit{App}, \langle b; \beta; n\rangle \diamond \mathit{Stk}, e[v/x]}$$

(local invoke)

$$\frac{\mathsf{top}(\mathit{Stk}) = \langle a; \alpha; m\rangle \qquad \mathit{App} = \langle N; K\rangle \qquad N(a) = \langle V; R\rangle \qquad V(\alpha) = \langle A; C\rangle \qquad A = \langle M; L\rangle \qquad L(k) = \lambda x.e}{\mathit{App}, \mathit{Stk}, :: k(v) \to \mathit{App}, \langle a; \alpha; \texttt{local}\rangle \diamond \mathit{Stk}, e[v/x]}$$

(import invoke)

$$\frac{\begin{array}{c} \mathsf{top}(\mathit{Stk}) = \langle a; \alpha; m'\rangle \qquad \mathsf{interface}(a, \alpha, m, \mathit{App}) = \langle \texttt{smixer}; I; E\rangle \qquad k \in I \\ \mathsf{depends}(a, \alpha, m, \mathit{App}) = \{b.\beta.n\} \qquad \mathsf{interface}(b, \beta, n, \mathit{App}) = \langle \texttt{smixer}; I'; E'\rangle \text{ or } \langle \texttt{rmixer}; I'; E'\rangle \qquad E'(k) = \lambda x.e \end{array}}{\mathit{App}, \mathit{Stk}, m :: k(v) \to \mathit{App}, \langle b; \beta; n\rangle \diamond \mathit{Stk}, e[v/x]}$$

(export invoke)

$$\frac{\mathsf{top}(\mathit{Stk}) = \langle a; \alpha; m'\rangle \qquad \mathsf{interface}(a, \alpha, m, \mathit{App}) = \langle \texttt{smixer}; I; E\rangle \qquad k \notin I \qquad E(k) = \lambda x.e}{\mathit{App}, \mathit{Stk}, m :: k(v) \to \mathit{App}, \langle a; \alpha; m\rangle \diamond \mathit{Stk}, e[v/x]}$$

(rmixer import invoke)

$$\frac{\begin{array}{c} \mathsf{interface}(a, \alpha, m, \mathit{App}) = \langle \texttt{rmixer}; I; E\rangle \\ k \in I \qquad \mathsf{interface}(b, \beta, n, \mathit{App}) = \langle \mathsf{smixer}; I'; E'\rangle \text{ or } \langle \mathsf{plugger}; I'; E'\rangle \qquad E'(k) = \lambda x.e \end{array}}{\mathit{App}, \langle a; \alpha; m\rangle \diamond \langle b; \beta; n\rangle \diamond \mathit{Stk}, k(v) \to \mathit{App}, \langle b; \beta; n\rangle \diamond \langle a; \alpha; m\rangle \diamond \langle b; \beta; n\rangle \diamond \mathit{Stk}, e[v/x]}$$

(return)

$$\mathit{App}, \langle a; \alpha; m\rangle \diamond \mathit{Stk}, \textbf{return}\ v \to \mathit{App}, \mathit{Stk}, v$$

Figure 10. Application Execution: Definitions and Reduction Rules

stance,a hot-deployed bean might not only need to interact with its container, but also interact with other beans, and also system libraries, to make it work. This is supported by the **install** part of the rule, since all dependency requirements C that W contains will be solved by **install**. Some elements in C can certainly specify how some of the mixers of the dynamic plugin can be satisfied by other assemblages in the application App.

3.4.2 Name Binding Rules

All the other rules handle some form of name binding; studying linking in the context of component deployment is a goal of this paper. Readers should pay attention to the (**import invoke**) rule, where it needs to read from the application graph to find the correct function to invoke. Due to the structural choice of our formalism, this lookup process might look inefficient, but in reality all the functions in the rule can be precalculated: the assemblage of concern can be associated with a symbol table and the function entry can be obtained without indirect lookup.

4. Formal Properties

One important evaluation of any formal framework is what properties may be proven rigorously. In this section, we state three important formal properties. Informally, they are that any buildbox created, deployed and any application runtime executed in our framework will stay well-formed, and any well-formed application runtime will access features without violating version compatibility. The proofs of these theorems are detailed in the long version [29].

THEOREM 1 (Well-Formed Evolution over LTS). *If* $\mathsf{wffAbb}(Abb)$ *and* $Abb \xrightarrow{l}_* Abb'$*, then* $\mathsf{wffAbb}(Abb')$.

THEOREM 2 (Well-Formed Run-time Evolution). *If* $\mathsf{wffAbb}(App)$ *and* App*,* Stk*,* $e \rightarrow_* App'$*,* Stk'*,* e'*, then* $\mathsf{wffAbb}(App')$.

Thm. 1 states that all operations defined by the LTS – including component building, assembling, installation, update – do not turn a well-formed buildbox into an ill-formed one. Note that since each labeled transition can be analogously thought of as a command issued by component deployers, providers and assemblers, such a theorem ensures that our framework is robust enought to fend off misuses of the commands (such as providing special parameters) to undermine buildbox well-formedness.

Thm. 2 states that running an application also does not affect application well-formedness; for instance, hot-deploying will not change a well-formed application into an ill-formed one.

Together with the trivial fact that the bootstrapping process (see the (**execute**) rule in Sec. 3.4) does not affect well-formedness and $\langle \emptyset_M ; \emptyset \rangle$ is trivially well-formed, we know that the buildbox created and deployed, and the application runtime launched from it in our framework are always well-formed at any given point of its evolution.

We now study version compatibility. Before we state the main theorem, we first define a relation $\hookrightarrow$ to capture where a feature implementation will be looked for at run time.

DEFINITION 1 (Run-time Feature Access). $App, a, \alpha, m, k \hookrightarrow App', b, \beta, n$ *holds iff*

- $\mathsf{wffAbb}(App)$.
- $App = \langle N; K\rangle, N(a) = \langle V; R\rangle, \alpha \in \mathsf{dom}(V)$ *for some* N, K, V, R.
- $App, \textbf{execute}\ a_0, \alpha_0 \xrightarrow{\text{exe}} App_0, Stk_0, e_0$ *for some* a_0, α_0, App_0, Stk_0, e_0.
- App_0, $Stk_0, e_0 \rightarrow_* App'', \langle a; \alpha; p\rangle \diamond Stk'', \mathbf{E}[m :: k(v)]$ *for some* App'', Stk'', p, v, $\mathbf{E}$.
- App'', $\langle a; \alpha; p\rangle \diamond Stk'', \mathbf{E}[m :: k(v)] \rightarrow App', \langle b; \beta; n\rangle \diamond Stk', e$ *for some* e.

THEOREM 3 (Compatible Code Access). *Given*

A1: $App_0 \xrightarrow{\textbf{ship}\ a, \alpha, \Delta}_S W$
A2: $m \in \Delta$
A3: $\langle \emptyset_M ; \emptyset \rangle \xrightarrow{l_1} \ldots \xrightarrow{\textbf{install}\ W} \ldots \xrightarrow{l_p} App'_0$
A4: $App_0, a, \alpha, m, k \hookrightarrow App_1, b, \beta, n$
A5: $App'_0, a, \alpha, m, k \hookrightarrow App'_1, b', \beta', n'$
A6: $App'_1 = \langle N'; K'\rangle, N'(\beta') = \langle V'; R'\rangle$

then $b = b'$*,* $n = n'$ *and* $\beta' \in \mathsf{newer}(R', \beta)$

This is the main theorem addressing the correctness of our framework in terms of version compatibility. It is not trivial because it spans the lifecycle of the component, from component development time (shipping) to its deployment (installation) to the *run-time access to its features*. It states that if a component is shipped from the development site (**A1**), and its mixer m is specified by the shipper to consider version compatibility (**A2**), then no matter where the packaged component is installed (**A3**) and then executed, any access to the features inside m (either as an import or an export) at run time will always locate a version of the component (**A5**) compatible with the version located at the development site if the same application is test-run (**A4**).

5. Discussion

An important aspect of any formal framework is how well it models the problem domain. This section is aimed at elucidating how our platform-independent framework can be mapped onto different platforms, how it relates to existing real-world deployment approaches, and/or what insights it can provide for their improvement. It serves both as a validation of our framework and as a summary of related work.

5.1 Deployment on Microsoft Platforms

Over the years several different software component models have been defined on Microsoft platforms, and each of them addresses the issue of deployment.

5.1.1 Dynamic Link Libraries (DLLs)

An application on earlier Windows platforms is deployed as a set of files in `.exe` and `.dll` formats. The `.dll` files are commonly shared by multiple applications, and many of them are provided by the operating system itself. The notorious problem [39] related to DLL deployment is *updating without version control*: when the installation of one application involves installing a new version of an existing `.dll` file, the new copy will overwrite the old copy and affect all other applications that depend on the library. If the new copy happens to be backward incompatible, the affected applications might behave erratically. This can happen weeks after the damage was done and, and is thus very hard to track down the cause. The DLL model is a very weak model of deployment that we do not feel a need to contrast with.

5.1.2 COM

COM [33] as a component model strongly promotes the use of interfaces, a feature also shared by assemblages. COM components are typically deployed in an environment, called *context* in which they run. A COM component developer can specify how a particular component should behave by modifying specific attributes that define the component's behavior within a context, a feature commonly known as "attribute-based programming". Such a feature can be modeled in our framework by using bidirectional interfaces, in the same way as modeling EJB environment entries, elaborated in Sec.2.3.2.

COM addresses versioning by using immutable interfaces: after one publishes an interface in a COM component with a universal ID, it should never be changed. Newer versions of the same component should create additional interfaces to the component. Such a solution can easily solve the problem of version backward compatibility, but in reality is prone to interface proliferation [21].

COM also inherited some older problems of the Windows platform, such as the requirement that all component metadata information be stored in a centralized repository, the Windows Registry. The principle of application isolation is not supported.

5.1.3 CLI Assemblies

CLI Assemblies [16] define a vendor-independent component standard, which also had as a major goal the solution of the deployment problem with DLLs on the Windows platform. The best known implementation of it is .NET CLR Assemblies [32] by Microsoft.

Each assembly as a deployment unit contains version-aware dependency information in a manifest file, which is precisely matched against at deployment time. Since solving the DLL problem was a primary aim, only shared libraries are versioned, and they are always referred to by *strong names*, which contain rich version information to avoid accidental name matches. Since different versions of the same assembly do not have the same strong name, they can co-exist (so-called *side-by-side deployment*). Updating a dependency to a newer version is allowed by adding a *version policy* file to an assembly, which claims to redirect the dependency to the new version. We share these features with Assemblies.

Since application-specific assemblies are not accessed by globally unique strong names, the correctness of locating the correct assembly is still subject to the lookup path setting, an issue very similar to the `CLASSPATH` issue of Java. Assembly lookup paths happen to be the least stable part of CLI specification, and different vendors have chosen different strategies (for instance .NET CLR and Mono [34] have different priorities on which path should be looked at first). In our framework, all assemblages are consistently accessed via its version identifier, so no name confusion can arise.

Assemblies are known to have difficulty in handling cyclic version dependencies. This does not look that bad in the specific domain Assemblies aim to have an impact on: Assemblies historically have a strong focus on solving DLL-related problems, so components *within* an application are not versioned and their dependencies – where cyclic dependencies are most likely to happen – are not considered. In general component-based software development, however, each individual component might be subject to version control. In fact, even on the CLI platform, two core library DLLs are still known to be cyclically dependent: there is a cyclic dependence between `System.dll` and `System.Xml.dll` that requires special handling. Interested readers can refer to the source code of an open source CLI implementation, Rotor [35].

The Assemblies framework does not consider the deployment site as a running evolvable system, and does not precisely specify the deployment action as a process.

5.1.4 Installers

On the Windows platform, many programs exist to address application shipping and installation. These tools are typically not more than a compression tool with a friendly user interface (the "wizard"), and they heavily depend on platform-dependent environment variables and scripts to configure packages inside. A better installer in this category is InstallShield. Its recent releases have conformed to the CLI Assemblies standard.

5.2 Deployment on the Java Platform

Java's component model is JavaBeans. The deployment model on the Java platform is mostly specified for a variant of the model, EJB. In this section, we will also consider how `.class` files are deployed.

5.2.1 EJB

The J2EE solution for component deployment is detailed in the specification for Enterprise JavaBeans (EJB) [17]. It revolves around a data structure called the *Deployment Descriptor* [5] associated with each to-be-deployed bean, which describes the interactions the bean may engage in at the deployment site. The primary task of a deployer is to establish the interaction between the to-be-deployed bean and its container. The way in which this interaction can be modeled in our framework was explained in Sec. 2.3.2. In addition, a bean can rarely achieve a task without collaborating with other beans, and it supports inter-bean dependency by introducing extra syntax (<ejb-ref>). In our framework, this is analogous to declaring a bidirectional mixer. EJB does not explicitly model versioning, and also does not define the build evolution process that the application buildbox models.

5.2.2 Deploying Java Programs

In a Java context, each `.class` file can serve as an independent deployment unit. It is common that a Java program is tested on the development site with one version of the class as the dependency, and then is installed on the deployment site with another version of the class. This is because `CLASSPATH` settings may be different between the development site and the deployment site, and so program behavior is vulnerable to the settings of the `CLASSPATH`. This problem does not arise in our framework, since components are referred to by their unique ID, not just by their name.

This issue also affects the way type safety is ensured. Consider a simple Java program where class `A` refers to class `B`. Suppose class `B` at the deployment site is different from the version used when `A` was compiled. Type safety would be jeopardized if the `B` with an incompatible type was loaded at run time. Java addresses the issue by performing load-time re-typechecking as part of bytecode verification. In our framework, if we make sure the `B` referred to by `A` is the same version (or a compatible version) of the one used by `A`, such a re-typechecking process will be unnecessary.

5.3 Deployment in CORBA

OMG has released a Deployment & Configuration (D & C) specification [36] for the CORBA Component Model (CCM). In Sec. 2.3.2, we have demonstrated how to model its bidirectional interfaces, and its deployment time attribute-rebinding is similar to EJB environment entries. D & C allows components to be assembled together to form an assembly. This can be modeled in our framework by the (**assemble**) rule.

One focus of this paper, versioning, is left out of D & C. Since a CCM component does depend on other CCM components (attested by its <dependsOn> tag), it is unclear what will happen when different versions of the same components are deployed. This will especially affect component update, which is left out in the current specification. The CORBA naming mechanism is also challenged in [44].

Since it is aligned with the general rationale of CORBA, the D & C framework focuses on distributed deployment, where different CCM components in one assembly can be deployed on different

[5] In EJB 3.0 (the latest version), metadata annotations are used to record deployment information, but its difference from deployment descriptors is only syntactic.

network nodes. Our framework as an abstract study does not explicitly model distribution, but it can be soundly re-interpreted if different components of the application lie on different nodes. This topic was discussed in Sec. 3.3.2. CORBA relies on its underlying bus to look up distributed components. We do not have such a mechanism, but note that the globally unique version ID ensures name confusion will not be an issue in the distributed context.

5.4 Deployment on Unix/Linux Platforms

5.4.1 Package Managers

Modern operating systems come with *package managers* and *installers* to assist in installing and upgrading software. On the Linux platform, well-known examples include RedHat's RPM [18], Debian 's Dpkg [12] and Gentoo 's Portage [40]. Despite differences in usability, the underlying goal of the three systems is the same: to install packages with the correct dependencies satisfied. How dependencies are created, checked and resolved is too specific in these systems. For instance, RPM is known to be complicated and weak in its support for dependency resolution. For cyclic dependencies, a tool like Dpkg will rely on a special specification from the package developer to break the cycle (declaring one of the dependencies to be a post-depends). In Portage, cyclic dependencies induce failure, and the resulting packages will not be installed. Broadly, package managers do not take into account the execution phase of the deployment lifecycle: the content in packages might not be executable code at all. The resulting model is weak to treat the deployment of code components: the declared package dependencies can be arbitrary, and not necessarily reflect inherent code dependencies; satisfying all package dependencies does not guarantee any correct behavior when code is executed. The popular archive distribution systems CPAN [10] (for Perl programs) and CTAN [11] (for TeX documents) can also be included in this category; these systems have a strength on how to locate packages on the Internet. There are also package managers for specific language environments, including RubyGems [43] and Scala Bazaar [3].

5.4.2 Deploying C Programs

C compilers on modern platforms almost always use dynamic linking for library access. A direct consequence is when a binary application is created (the one with default name `a.out`), it rarely forms a closure in terms of name binding. When shipping a C application using function `strlen`, the implementation of `strlen` is only defined by the library at the deployment site.

If applied to our framework, a deployment unit for C programs can either be the non-closed application code (with default name `a.out`) or the shared library code (the one defining `strlen`, commonly represented as files with suffix `.so` on Unix-compatible platforms). What our framework can help avoid is to create an application depending on one version of the library with `strlen`, and then being deployed at a deployment site with a different (and incompatible) version of the library. Our framework can also make sure multiple versions of the same library can be installed, with some applications use one, and other applications use the other.

5.5 Hot Deployment

A number of research projects address the ability to dynamically load/link/update software components, which in various degrees overlap with our discussion on dynamic component deployment.

Dynamic linking/loading of code fragments provides a foundational layer of this problem. Existing approaches comparable to our **plugin** expression are those programmable dynamic linking/loading mechanisms. Well-known examples include Java's `ClassLoader`, Assemblies' `Assembly.Load`, and Unit's invoke [19] expression. From a language construct perspective, our **plugin** expression differs from these approaches as it respects the *bidirectional* contract of the interactions between the link/load initiating component and the linked/loaded component, where both of the parties provide an interface (a plugger or mixer) to specify what it needs and what it requires. Declaring the plugger on the link/load imitating component gives the component a more declarative specification on its ability to perform dynamic linking, and in the context of component deployment, it provides an explicit declaration of the fact that a form of dynamic deployment is supported. MagicBeans [7] provides a library to help develop dynamic plugins with the use of the Observer pattern, and our **plugin** covers that model. Typing issues of dynamic linking have been well studied, such as linking of assembly code [20] and C# and Java's dynamic linking [13]. In our previous work [27] a deployment unit must be type-safe, and so for simplicity we have left out typing issues here.

EJB hot-deployment is not defined the EJB specification, and it is supported only in a vendor specific manner. For example, Weblogic [4] supports hot deployment via several custom tools.

A well-known example of plugin-based software is Eclipse [15]. The recent release (3.0) has also included the ability to add plugins at run time, *i.e.* dynamic plugins, based on the Open Services Gateway Initiative (OSGi) [37] specification. OSGi only provides limited support for handling dependencies. Their consistency checking is optional and not rigorously defined, especially for the versions of dependencies.

Some projects have addressed issues related to dynamic updating and reconfiguration of software. The Dynamic Software Updating system [23] focuses on correctness, usability and type safety. It does not focus on software deployment, and does not consider version control. OpenRec [24] is a software architecture effort focusing on how to design dynamically reconfigurable systems. In [8], a few more practical issues related to dynamic reconfiguration of distributed systems are considered, such as the handling of stubs. Our abstract framework does not address this issue.

5.6 Deployment-Related Formalisms

Formal frameworks that address component deployment in a vendor-independent manner are rare. CLI Assemblies' name binding mechanism was recently formalized by Buckley [5]. The focus of that work was to demonstrate the use of strong and simple names in the framework. No formal property was proved, and the framework does not address application evolution in the deployment lifecycle. The linking of DLLs was formalized in [14], where the focus was on type safety.

A conceptual framework for component-based software deployment was proposed in [38]. As a result of a highly conceptual treatment, inter-component dependency – arguably the central issue in modeling complex process of deployment – was not modeled. No formal rules were given to describe the deployment process or application execution.

5.7 Complementing Module/Component Systems

Assemblages, the abstract component construct used by our deployment framework, was first described in [27]. This previous work had a different goal: equipping mixin-like modules with interfaces to directly support Internet-era concepts, such as distributed communication and dynamic linking. Deployment and versioning were not considered in that paper. In this work, we have reused the assemblage construct itself, but almost every structural choice was rejustified in a deployment context, such as why we need bidirectional interfaces and why we need pluggers for hot-deployment. Put together, they paint a complete picture to justify the need for an assemblage-like construct in component design: 1) it is fit for model modern programming concepts such as distributed communication and dynamic linking (the thesis of [27]); and 2) it is fit for component deployment (this work).

Most of the structural assumptions we have made about our deployment units are common to many modern module/component systems. The generality makes our deployment framework useful for studying deployment in a variety of next-generation module/component systems. Module systems with bidirectional interfaces include Units [19] and Jiazzi [31]. In MJ [9], the Java classloader is replaced with a compile-time notion of module, in which module dependencies are separated into two categories: static dependencies (`import` and `export`) and dynamic dependencies (`dynamic export`). These two notions can be modeled as mixing and plugging in our framework, respectively. In Fortress [1], program fragments are organized into components with interfaces of explicit `import` and `export` declarations, and are organized into a persistent store called a *fortress*, where a few pre-defined library operations (called *targets*) are defined such as `link`, `upgrade`, and `execute`. A fortress can analogously be thought of as a deployment site in our framework, where the targets can mapped to our LTS operations. The Fortress specification only specifies signatures for the targets and allows different vendors to provide their own implementations. Our framework can be thought of as providing a guideline for what a "well-formed" fortress should be, and how different implementations should abide so that good properties of a fortress will not be undermined. In Fortress, because there is no support for multiple interfaces, clashing of names are common at `link` time, and Fortress has to resort to special rules to handle them.

6. Conclusion

In this paper we showed how the complex, ad hoc software deployment cycle could be reduced to a calculus with a small set of platform-independent, vendor-independent operations that define how the deployment site should evolve. It elucidates the subtle relationships between pre-runtime application buildbox evolution and run-time application evolution, and proves formal properties about application well-formededness and version compatibility throughout the evolution process. It also serves as a study of component design from the perspective of the deployment unit, where component dependency in terms of timing and location is studied. Expressive forms of dependency such as parametric and cyclic dependency between components are also addressed.

With this foundational framework defined, we would like next to investigate how more advanced features important in component deployment can be expressed on top of it, for instance security, distribution, and transaction control. A formal treatment of these issues within the deployment lifecycle should give us deeper insights into these hard problems.

References

[1] Allen, E., Chase, D., Luchangco, V., Ryu, J. W. M. S., Steele, G., and Tobin-Hochstadt, S. The Fortress Language Specification (Version 0.618), April 2005.

[2] Balaban, I., Tip, F., and Fuhrer, R. Refactoring Support for Class Library Migration. In *OOPSLA '05* (2005), pp. 265–279.

[3] The Scala Bazaar System, `http://scala.epfl.ch/downloads/sbaz.html`.

[4] BEA. BEA WebLogic Server Enterprise JavaBeans 1.1, `http://www.weblogic.com/docs51/classdocs/API_ejb/`.

[5] Buckley, A. A model of dynamic binding in .NET. In *Proceedings of 3rd International Working Conference on Component Deployment* (2005), pp. 149–163.

[6] Cardelli, L. Program fragments, linking, and modularization. In *POPL'97* (1997), pp. 266–277.

[7] Chatley, R., Eisenbach, S., and Magee, J. Magicbeans: a platform for deploying plugin components. In *Second International Working Conference on Component Deployment* (2004), vol. 3083, pp. 97–112.

[8] Chen, X., and Simons, M. A component framework for dynamic reconfiguration of distributed systems. In *Lecture Notes in Computer Science, Volume 2370* (Jan 2002), vol. 2370.

[9] Corwin, J., Bacon, D. F., Grove, D., and Murthy, C. MJ: a rational module system for java and its applications. In *OOPSLA'03* (2003), pp. 241–254.

[10] Comprehensive perl archive network, `http://www.cpan.org`.

[11] Comprehensive tex archive network, `http://www.ctan.org`.

[12] Debian package management, `http://www.debian.org`.

[13] Drossopoulou, S., Lagorio, G., and Eisenbach, S. Flexible models for dynamic linking. In *Proceedings of the 12th European Symposium on Programming* (2003).

[14] Duggan, D. Type-safe linking with recursive DLLs and shared libraries. *ACM Transactions on Programming Languages and Systems 24*, 6 (2002), 711–804.

[15] Eclipse, `http://www.eclipse.org`.

[16] ECMA. Standard ECMA-335: Common Language Infrastructure, 2002.

[17] EJB 3.0 Expert Group. JSR 220: Enterprise JavaBeans Version 3.0, June 2005.

[18] Ewing, M., and Troan, E. The RPM packaging system. In *Proceedings of the 1st Conference on Freely Redistributable Software* (1996).

[19] Flatt, M., and Felleisen, M. Units: Cool modules for HOT languages. In *PLDI'98* (1998), pp. 236–248.

[20] Glew, N., and Morrisett, G. Type-safe linking and modular assembly language. In *POPL'99* (1999), pp. 250–261.

[21] Gordon, A. *The .NET and COM Interoperability Handbook*. Pearson Education, Inc., Upper Saddle River, NJ, USA, 2003.

[22] Hall, R. S., Heimbigner, D. M., and Wolf, A. L. Evaluating software deployment languages and schema. In *ICSM '98: Proceedings of the International Conference on Software Maintenance* (Washington, DC, USA, 1998), IEEE Computer Society, p. 177.

[23] Hicks, M. W., Moore, J. T., and Nettles, S. Dynamic software updating. In *PLDI'01* (2001), pp. 13–23.

[24] Hillman, J., and Warren, I. An Open Framework for Dynamic Reconfiguration. In *ICSE'04* (2004), pp. 594–603.

[25] Installshield, `http://www.installshield.com`.

[26] Liskov, B., and Wing, J. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems 16*, 6 (Nov. 1994), 1811–1841.

[27] Liu, Y. D., and Smith, S. F. Modules With Interfaces for Dynamic Linking and Communication. In *ECOOP'04* (2004), pp. 414–439.

[28] Liu, Y. D., and Smith, S. F. Interaction-based Programming with Classages. In *OOPSLA '05* (2005), pp. 191–209.

[29] Liu, Y. D., and Smith, S. F. A Formal Framework for Component Deployment (Long Version), `http://www.cs.jhu.edu/~yliu/deploy/`. Tech. rep., The Johns Hopkins University, Baltimore, Maryland, March 2006.

[30] McCamant, S., and Ernst, M. D. Early identification of incompatibilities in multi-component upgrades. In *Proceedings of the 18th ECOOP* (2004), pp. 440–464.

[31] McDirmid, S., Flatt, M., and Hsieh, W. Jiazzi: New-Age Components for Old-Fashioned Java. In *OOPSLA'01* (2001), pp. 211–222.

[32] Meijer, E., and Gough, J. Technical Overview of the Common Language Runtime, 2000.

[33] MICROSOFT. Component Object Model Technologies, `http://www.microsoft.com/com/`.

[34] Mono, `http://www.mono-project.com`.

[35] MSDN. Shared Source Common Language Infrastructure 1.0 Release, `http://msdn.microsoft.com/net/sscli/`.

[36] OBJECT MANAGEMENT GROUP. Deployment and Configuration of Component-based Distributed Applications Specification, July 2003.

[37] OSGI. Open services gateway initiative service platform, release 4 core, available at `http://www.osgi.org`, 2005.

[38] PARRISH, A., DIXON, B., AND CORDES, D. A conceptual foundation for component-based software deployment. *Journal of Systems and Software 57*, 3 (2001), 193–200.

[39] PIETREK, M. Avoiding DLL hell: Introducing application metadata in the microsoft .NET framework. *MSDN Magazine, available at* `http://msdn.microsoft.com` (2000).

[40] Portage, `http://www.gentoo.org`.

[41] REID, A., FLATT, M., STOLLER, L., LEPREAU, J., AND EIDE, E. Knit: Component composition for systems software. In *Proc. of the 4th Operating Systems Design and Implementation (OSDI)* (October 2000), pp. 347–360.

[42] RINAT, R., AND SMITH, S. F. Modular internet programming with cells. In *Proceedings of the 16th ECOOP* (2002), pp. 257–280.

[43] Rubygems, `http://rubyforge.org/projects/rubygems/`.

[44] SZYPERSKI, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.