Dynamic Reconfiguration of Component-Based Applications in Java

by

Ziqiang Tang

B.S., Electrical Engineering and Computer Science
U.C. Berkeley, 1998

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2000

Author .....
Department of Electrical Engineering and Computer Science
August 17, 2000

Certified by .....
Barbara Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Committee on Graduate Students

Dynamic Reconfiguration of Component-Based Applications in Java

by
Ziqiang Tang

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2000, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

In this work, we present mechanisms for transparently evolving component implementations in an application while preserving instance consistency. We also describe a runtime system based on the Java programming language that allows an application to be dynamically reconfigured by evolving instances of underlying components.

Our mechanism for component evolution allows us to convert instances of a component type from one implementation to another while meeting the same specification. The evolution occurs online, and preserves on-going computation and all relevant run-time instance state. Previously existing systems that dynamically upgrade software place severe programmatic or performance constraints on the upgrade process, such as requiring the use of transactional semantics or application termination. Our work differs by specifying additional properties that support instance evolution without placing unrealistic constraints on the implementation. These properties are defined in the component interface and constrain implementations of the component such that necessary instance state can be mapped between differing implementations.

Thesis Supervisor: Barbara Liskov
Title: Ford Professor of Engineering

Acknowledgments

I would like to thank my thesis advisor, Barbara Liskov, for helping me accomplish this wonderful task. Without her timely advice and critical comments, I would not be as happy with this work as I am today.

I would also like to thank my faculty advisor, Daniel Jackson. Daniel's enthusiasm for all things was infectious, and his insightful comments and suggestions have helped carry this project far.

Chandra Boyapati played a wonderful role in continually challenging me with intellectual exercises, both in the scope of this work and outside... very far outside. Without long discussions with Chandra on possibilities and held assumptions, I would have turned back long ago.

This thesis would not have made it to its current form without the help of Sameer Ajmani, who volunteered significant amounts of his own busy time in order to read and comment on early drafts of this document.

Much thanks must go out to all of my lab-mates that have managed to put up with me for the past two years. This includes Sarah Ahmed, Miguel Castro, Kincade Dunn, Kyle Jamieson, Nick Mathewson, Rodrigo Rodrigues, Jon Whitney, and Yan Zhang.

No man survives in isolation, and I want to thank all of my friends for making my life what it is. Special thanks go out to Q and Laura for making sure I was properly nourished, both physically and mentally. Special everythings go out to Eileen.

Nothing I have now could have been achieved without the continued support of my parents. The more I seem to learn, the more I recognize the extent of their wisdom. Thank you for understanding the difference between pressure and encouragement so well. I can't wait to see both of you on the golf course.

Contents

List of Figures

Chapter 1

Introduction

The future of network-based pervasive computing requires fundamental modifications to the way applications are implemented and deployed. Most existing applications are designed monolithically for a particular deployment environment, constraining them for use on a single machine architecture and operating system. These applications directly interface with and use resources that are statically assumed to be available within its deployment environment. Thus, the application can only be deployed in an environments where all of its expected resources are available, and other available resources that were not expected are not utilized.

Network-based computing is premised on the idea of flexibility. The network makes available third-party resources that can be shared by multiple applications. Even as the set of resources available over the network changes dynamically, applications designed for this environment should behave as well as possible. They should take advantage of new resources as they become available over the network, and then smoothly degrade their quality of service as resources leave the network. For example, a file-system relying on network-based storage could automatically compress data to reduce demand as available servers leave the network.

Pervasive computing has a related goal of providing connected computing applications to the user at all times. This suggests that applications must remain available even as the actual physical environment and platform changes. For example, an application's graphical user interface should adapt to use the visual display closest to the user as she walks down the hall. Thus, the application must be able to adapt according to changes in the physical environment. The mobile computing devices that applications in such an environment execute on introduce other challenges; available hardware resources such as bandwidth, memory, and battery life can fluctuate quickly on such devices. Ideally, applications using these platforms would adapt for optimal operation as resource levels change [13].

This work focuses on mechanisms for dynamic adaptation of applications for their changing execution environment. We do not tackle the issue of how this dynamic change should be specified, or managed; instead, we focus on the actual process of adaptation. Previous work in this domain have generally made two key assumptions that render them impractical for real-world usage. The class of applications we support don't share these assumptions.

1. One common assumption that we do not share is that applications are developed with

full knowledge of all potential future adaptations. In a heterogeneous network environment with constantly evolving third-party resources, applications adaptations should not be constrained to particular implementation changes. We do not want to limit the flexibility of an adaptive application by supporting only those changes that the original developer might have conceived.

For example, imagine the design of an application that uses a caching system to store large amounts of data. The developer might recognize that different platforms manage memory differently, and implement the application to adapt behavior by changing cache size as needed. However, the developer might not have realized that the application should also adapt according to the user's changing security profile, and cache data in encrypted form as needed. The application should be aware that it requires a caching system, but little else; the particular implementation to be used should be determined dynamically. This allows the application's flexibility to be unconstrained by the developer's imagination.

2. Other systems also commonly assume that adaptations are relatively rare, and thus can be treated as exceptional conditions. We remove this restriction and instead allow the execution environment for applications in this domain to change quickly and at any time during the execution of the application. For example, as hand-held devices move about in the physical world, the execution environment can change as quickly as the locally available network connection changes. The application should adapt its implementation of the connection layer as soon as possible; waiting for application termination or direct user intervention is not an option.

Our approach focuses on making adaptations to part of the application that are transparent to the rest of the application. Our system provides for the construction of applications using independently implemented software modules. These modules are frequently called **components**; we define the precise meaning of this term in our work in the next chapter. Components are self-contained software packages implemented by third-party developers that can be integrated to form a complete application. The run-time system automatically adapts applications by bringing in alternative component implementations as needed in response to changes in the execution environment. Here, we define the execution environment informally as the set of available resources (whether local or network-based) and application parameters that affected application behavior and software implementation.

We would like the adaptation process to be transparent in two ways. First, we desire *implementation transparency*: the application need not be aware of the exact component implementation used to provide certain behavior. In addition, there is *change transparency*: the application should not be aware of when the adaptation happens. Transparency properties such as these are needed to develop flexible applications that are not burdened with the substantial complexity of managing adaptation. To achieve these properties, we propose a system design that preserves consistency and application invariants during the evolution process.

The remainder of this chapter elaborates on the motivations and primary contributions of this work. Section 1.1 presents a brief introduction to the role abstractions and component frameworks have played in designing flexible applications. Existing frameworks supporting online adaptation for active applications are heavy burdens in performance and implementation costs. This renders them inadequate for our application domain. Section 1.2 discusses some existing mechanisms, and our general approach for designing a practical system. Section 1.2.3 summarizes our primary contributions. Finally, Section 1.3 provides an overview of the remainder of this thesis.

1.1 Adapting Applications with Abstractions

Preserving application consistency and implementation transparency relies upon the use of *abstractions*. Abstractions frame interactions that occur across them, allowing different implementations and clients to inter-operate across such a boundary as long as certain constraints are respected. This allows for freedom in any dimension not constrained by the abstraction, meaning the implementation can be specialized for a specific set of operating parameters.

Components are hidden behind abstractions that constrain implementation behavior. An application relies on the behavior described by the specification of the abstraction to complete certain tasks, and component implementations fulfilling the specification can then be used interchangeably to actually provide the behavior. Components can be implemented for specialized execution environments (such as different machine architectures), making it possible for the application to be available in a wide range of execution environments by switching between components as needed.

For example, an application could rely on a `LookupTable` abstraction to provide the behavior of mapping a key to a previously inserted value. One component implementation fulfilling the `LookupTable` abstraction might be specialized for reduced memory usage; it uses data compression and encoding schemes (such as Huffman codes) for use in an execution environment where storage is at a premium. Another component implementation fulfilling the same abstraction could choose to tradeoff speed versus size by using a hash-table implementation. The application can use one of these two implementations depending on its available storage capacity.

Applications create and use instances of the component exclusively through its' abstractions. The run-time system transparently maintains one-to-one mappings between each component abstraction and a component implementation selected to operate in the current execution environment. This mapping can be dynamically adjusted, possibly in response to changes in the execution environment. Each instance of the component abstraction used within the application is matched by an instance of a component implementation.

We call this process of adapting component implementation mappings within an application a *reconfiguration*. Reconfigurations leverage the strength of abstractions; a reconfiguration replaces an active implementation hidden behind an abstraction with another implementation similarly hidden. To achieve our goal of transparency for clients of the component, *all* references between components and their clients must occur through an abstraction layer to allow dynamic component replacement. Consistency generally requires that execution traces for component instances meet invariants declared as part of the component abstraction, before and after reconfiguration occurs.

1.1.1 Component Abstractions

The notion of separating abstract interface and implementation is well represented in object-oriented programming languages (such as C++ and Java). Interface abstraction and subtype polymorphism allow code to operate on objects without being immediately aware of the underlying implementation; all accesses to the object are constrained by the declared interface. In this way, C++ and Java classes can theoretically be independently developed (and maintained) before being integrated into a single application for deployment or use.

Component implementations are similar to classes in a number of ways. Functionally, they are similar in that both provide concrete implementations for object instances, including state and behavior. Component abstractions are used to hide implementation detail just as interface abstractions can be used to hide classes, promoting modularity and flexibility in application design. From an implementation perspective, components are often constructed out of language-level classes.

One distinction between the component programming model and classes lies in an increased emphasis on *deploying*, and not just developing, applications in a truly modular fashion. In practice, applications built using C++ and Java classes often tightly bind abstraction and implementation. The separation based on abstraction between different implementations is strengthened with the component model. All interactions between components *must* occur through a firm abstraction layer that loosens the connection between code implementations, in effect introducing true implementation transparency. The component model is essentially a restricted form of object-oriented programming that limits the way clients and classes interact through abstractions. It is unlikely that an arbitrary implementation class in an application can be replaced by an entirely different class after the application has been deployed; such changes would require access to the source code and recompilation of the application. In contrast, applications built with components are intended to support precisely this kind of change.

Applications developed using classes do not have this kind of flexibility after deployment for a few reasons. For example, in Java and C++ instances of classes are created by specifically naming the implementation; there is no notion of creating the instance of an abstraction or virtual type. Even the class factory design pattern [14] requires the factory to name the implementation to be created. As [12] points out, Java lacks the support for *assembly-line programming* necessary for easy re-binding between abstraction and implementation [1].

In addition, a class's object layout is constrained by fields declared in a super-type. An object's fields are read from and written to directly from clients of the object, based on a field's expected placement in the presentation. In C++/Java, all classes inheriting such fields must implement a consistent object layout that allows such access.

Components generally prohibit this type of direct access. Instead, access to instance state variables (often called properties in the case of component architectures) is available through method invocations to abstract accessors/settors that allows the component to choose any object layout it desires; this means a component can implement `foo` in a completely different manner, for example as a combination of two other fields.

The growing importance of component-based application reconfiguration is well described in [31]:

> Modern software systems are increasingly required to be open and distributed. Such systems are open not only in terms of network connections and interoperability support for heterogeneous hardware and software platforms, but, above all, in terms of *evolving and changing requirements.* [We must] escape from traditional models of software development that assume system requirements to be closed and stable. We argue that open systems requirements can only be adequately addressed by adopting a

[1] Although Java reflection could be used to add this form of flexibility, the associated overhead assures this rarely occurs in practice.

> component-oriented ... approach, by shifting emphasis away from programming and towards generalized software composition.

Component frameworks such as CORBA [32], COM [5], and JavaBeans have expanded upon the original object-oriented programming paradigm and make it possible to implement very flexible applications consisting of third-party components. Changing the underlying behavior of an application requires a simple adjustment of the bindings between various component implementations, but do not require changes to using code.

The distinction between component implementations and classes is exemplified by a JavaBeans vs. Java comparison. Although the JavaBeans component framework is based strictly on the Java programming model and implemented using Java classes, it provides a structured environment for application design where use of components is bound exclusively to abstract interfaces and not implementations. A JavaBean implementation can be easily replaced in an application with another implementation meeting the same interface.

However, existing component frameworks are insufficient for our needs. The abstractions existing component frameworks support are generally limited to declarations of method signatures and properties (which are desugared as method invocations). These basic interfaces only allow the rewiring of applications such that methods and constructors might be invoked on multiple different concrete component implementations. In order to enable transparent online adaptation of *instances*, we also require component interfaces that provide abstractions for on-going computation at the level of a single component instance.

1.2 Online Reconfigurations

During the reconfiguration process, a component implementation currently used by the running application to fulfill an abstraction is replaced dynamically with a new component fulfilling the same abstraction. New instances of the component abstraction will be created using constructors on the current component implementation.

Existing instances of the component are adapted to use the new implementation while preserving all of the instance state needed for consistent behavior, including its identity. Moreover, the run-time type of an existing instance of the component, which specifies all of the agreed upon abstractions of the underlying component implementation, remains unchanged from the perspective of its clients. This form of change has the effect we desire: the application is adapted to exclusively use the new implementation transparently while preserving consistency.

For example, a graphical user interface that renders in great detail but consumes high amounts of battery power might be converted to another implementation that meets an identical interface but renders in less detail and saves power. Since we desire the change to be entirely transparent to the client application, which treats both implementations equivalently as an abstract GUI toolkit, the application should not have to re-create the visual interface with the new GUI toolkit component. Instead, transparency requires that the new GUI automatically gain the render state of the previous GUI instance. Transparency would also require that any on-going computation in the component, such as long-running animations, be preserved even as the underlying component changes.

The timing of the reconfiguration is also an important consideration. The most basic mechanism for component replacement simply waits for application termination before integrating new implementations (e.g., dynamically linked libraries). At restart, application behavior is usually re-initialized and begins anew. There is no requirement for instance or state consistency between different application executions. For some environments, waiting for termination before adapting the application is not a serious restriction. In this work we focus on an application domain where applications must be reconfigured while remaining online. Resource levels defining the current operating environment can change frequently, and the application must be able to adapt for operation in the new environment.

1.2.1 Application Quiescence

The work in [24] by Kramer and Magee proposes the condition of *quiescence* in a running application as a key part of dynamic change management. By their definition, an application reaches quiescence when all of the instance nodes to be evolved are not involved in on-going computation. Clearly, the effort involved in replacing a node becomes substantially easier if computation does not need to be preserved. With this approach toward quiescence, the problem reduces to one of adapting object representations as components change. This problem is also well known as *schema evolution* due to its origin in the database community.

The transaction-based dynamic change management system presented in [24] uses a straightforward approach to satisfy quiescence. It first finds the set of all computation nodes that can initiate a transaction affecting the node to be evolved; all of the nodes in this set are then marked *passive*. While passive, these computation nodes can not initiate new transactions. Once all potential clients of a node to be evolved have been made passive and the node itself has completed all active transactions, the node is considered *quiescent*.

Basic mechanisms for achieving quiescence in this manner have followed two primary approaches.

The first approach relies upon transactional semantics to abort active computation. This does not lend well to applications in the user-domain due to performance and implementation overheads related with state checkpointing and aborted computations. A variant of this first approach treats method invocations as the equivalent of transactions. Under this model, quiescence is achieved by forcing early completion of all on-going method invocations on the evolving instance using exceptions. However, this approach of equating method invocations and transactions remains unsatisfactory. Aborting computation with exceptions makes method implementation difficult. In addition, the use of exceptions increases complexity for clients of the component substantially, which must be prepared to catch exceptions across every component method invocation, and then re-invoke the method on the new implementation. Furthermore, discarding work already accomplished by the method invocation on the first implementation is rather wasteful.

The second approach is demonstrated by Kramer and Magee's system. After nodes capable of initiating computation have been set passive, quiescence is achieved by waiting as long as needed for all on-going transactions to complete [24]. If nested transactions are allowed, this potentially requires the completion of multiple transactions affecting multiple application nodes.

The Kramer and Magee approach can also be generalized as waiting for all method invocations on affected objects to complete before declared quiescence. However, we find this approach

unacceptable for our chosen application domain. Long running methods can delay application reconfiguration significantly; indeed, there are methods (such as event loops) that might never complete until application termination. Since every method incurs some overhead, applications can't simply be redesigned to use short-lived methods, pushing longer-lived loops higher up the call chain. Opportunities for quiescence should not be restricted to only the beginning and end of method invocations.

Other approaches toward evolving objects define a slightly different variant of quiescence where on-going computation is in some sense minimal, but not necessarily complete.

Reconfiguration points The MMLite system proposed in [17] uses labels to indicate clean points, which are statements placed into the original source to signify execution points in the code where local object state is consistent. Threads synchronize on clean points to assure that all computation is blocked while instance evolution occurs via a special mutator thread. The form of quiescence in this system only requires threads to be blocked at clean points, instead of total completion of all computation (method invocations). However, in their system mutator threads are only capable of making limited modifications, such as changes to the object representation of instances. More significant changes, such as modifications to the method implementation, would make it impossible for computation threads to continue execution consistently upon entering the critical region.

The work in [21] proposes the notion of *reconfiguration points*. The use of reconfiguration points is similar to the use of clean point synchronization in MMLite, but extends it by capturing a wider range of computation-related meta-data. The original source is translated to record relevant local meta-data such as call frames in a continuation-like form, which is then preserved and transferred over to the new instance when execution reaches a reconfiguration point. This mechanism preserves details of on-going computation, such as pending method returns, and maps them into a new context. Reconfiguration points allow a more powerful form of quiescence. The mechanism used for capturing call-frame meta-state allows a node to be considered quiescent even *before* method invocations have completed. In addition, by defining quiescence such that the state related with on-going computation can be mapped, more substantial changes to the application state are possible in this system than with MMLite. However, the system presented in [21] does not fit our definition of application reconfiguration; it only provides for migrating applications with identical implementations from one machine architecture to another.

1.2.2 Maintaining instance consistency

Intuitively, consistent instance evolution is only possible if all meaningful instance state is preserved. However, the underlying implementations and representations can be drastically different between existing classes, and the classes selected to replace them. There needs to be a mechanism for mapping all relevant state from an instance of one implementation to an instance of another implementation. These mappings may only be possible at certain points, when the instance to be evolved is *quiescent*.

We consider instance state to be held in two different regions in the application: the stack and the heap. Mapping the state on the heap requires mechanisms similar to those provided for schema

evolution. Mapping the state on the stack, which represents on-going method invocations, requires a novel approach based on the use of reconfiguration points. In this thesis, we provide mapping mechanisms that preserve consistency after evolution in both state regions.

Existing techniques for introducing quiescence use mechanisms that are heavy-weight, and do not allow on-going transactions or method invocations. A critical addition to our approach toward instance state mapping is the definition of rules for bringing about quiescence in the application that still allows the preservation of on-going method invocations after evolution. We show that by using reconfiguration points, we can separate our criteria for quiescence from method invocations.

Our solution assumes that component implementations share component interface specifications, but little else. Notably, components are not aware of the implementation details of the other components participating in the evolution process. Part of the component specification defines a shared interface that allows clients of the instance to treat all implementations of the component equivalently. Our approach extends the component specification to also include more information about method behavior during evolution.

1.2.3 Contributions

This thesis presents a new way of doing reconfigurations on the fly. The approach presented in this work differs from earlier work in that it is more flexible as to when a reconfiguration can occur. Rather than waiting until an application is not running at all, or at least is not using objects affected by a reconfiguration, our approach preserves the state and identity of objects whose implementation changes in the reconfiguration. Furthermore, our approach allows reconfigurations to happen even when methods are running inside the objects affected by the reconfiguration.

In more detail, the thesis proposes:

1. A component model for applications that allows reconfigurations in which one component implementation is replaced by another. The replacement is invisible from the perspective of any code that uses the component.

2. A way of specifying component abstractions that allows switching from one component implementation to another without the need for developers of the implementations to understand other implementations. The specification is extended with certain information that is of interest only to implementers of the component.

3. A way of implementing components to satisfy the extended specification. This takes the form of an extended Java programming language, which could then be translated via a preprocessor into normal Java, where the code produced by the preprocessor contains calls on the reconfiguration system discussed in the next point.

4. A description of a system that causes component replacement to occur. The system will cause objects belonging to replaced component implementations to be converted to the current component implementation one at a time. It allows objects to be converted even when methods are running inside them, provided every thread is at a "safe point" with respect to that object.

1.3 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 presents the component model used in this system, and details our fundamental mechanism for instance evolution. We introduce the way instance evolution occurs within our system, and describes the fundamental conditions that limit opportunities for evolution within our model.

Chapter 3 presents the Java-language extensions necessary to implement the component evolution system presented in Chapter 2. We discuss the language and system mechanisms used to declare and implement the component abstractions in our model. Using an example component representing a network stack, we step through the requirements associated with implementing a component.

Chapter 4 extends further the results from Chapter 2. It takes another look at the notion of quiescence and its relevance to our system. It describes the system level features needed to support entire application reconfigurations, rather than just isolated component instance evolution. We present key optimizations necessary to make application reconfiguration more timely. There can be a substantial burden in real-world performance associated with the need to achieve quiescence; we analyze several cases where this burden can be reduced.

Chapter 5 presents other works related to this thesis. Finally, chapter 6 concludes the thesis and proposes areas of future research.

Chapter 2

Component Evolution

This chapter discusses our strategy and design for supporting the adaptation of component instances. In this chapter, we introduce the component abstractions used throughout this thesis, and explains the motivation behind their design. Section 2.1 lays out the specific design goals our mechanism for component evolution must satisfy. Section 2.2 presents an overview of the component model supported by our system. Section 2.3 discusses the internal architecture of the applications and the component packages they use. Section 2.4 introduces the basic design of the component instance evolution mechanism based on the principle of preserving instance state.

In Chapter 3, we introduce a set of Java language extensions for implementing reconfigurable components based on work presented in this chapter.

The focus of this chapter is on the semantics of the evolution process for a particular component instance without regard to interactions with other component instances and the application at-large. Chapter 4 discusses reconfigurations from more of an application perspective.

2.1 Evolution Goals

The purpose of this system is to support dynamic adaptations of running applications. This system has five specific requirements that frame our ultimate goals.

Timeliness The application domain this work focuses on allows reconfiguration requests frequently enough that waiting for application termination is inadequate; this includes long-lived applications that might never terminate. The conditions that allow instance evolution should be specified aggressively.

Consistency Applications should behave in a consistent manner *from the perspective of the user.* Although we cannot guarantee implementations are implemented correctly, at least our mechanisms supporting change in implementation should not cause disruptions in application behavior that are perceptible to the user. This generic goal requires support at the component level; component specifications must be detailed enough to fulfill sufficient safety properties for consistent behavior.

Performance Over-all performance should not be substantially degraded when applications are designed to be adaptable; this also applies to application performance when reconfigurations are not occurring. For examples, transactional systems that use state checkpointing would affect performance even when the application is not adapting. Similarly, the use of mechanisms such as exceptions that abort and then waste on-going computation during adaptation should be minimized.

Programmability The system proposed here should be usable with minimal difficulty by existing development and deployment processes. The trade-off between detailed component specification versus over-constrained implementation must be considered when balancing *programmability* with *consistency*. The application model should be familiar to traditional application developers. This suggests that transactional semantics, software buses, and asynchronous message queues are undesirable compared to traditional method invocations. Programmers should have minimal responsibilities for assisting the runtime system in the evolution process.

Implementation-Independence Applications should be developed without any specific knowledge of component implementations. This offers more opportunity for component re-use. Components interact in two different ways that require implementation independence:

1. Using code should be independent of component implementation. This allows multiple component implementations, each with different implementation features and trade-offs, to be used interchangeably without requiring modifications in using code. Furthermore, dynamic changes in the component implementation used in the application should be completely hidden from using code. Clients of the component should not be forced to respond to implementation changes, or even be aware that a change has occur-ed. This would allow the run-time system to have complete control over when and how reconfiguration occurs.
2. The specific object rep and implementation of a component implementation should be independent of other implementations, despite the fact that object state is shared during implementation replacement for existing instances. This form of independence allows the growth in number of transform functions (code used to map object state between different implementations during component replacement) to be linear in the number of different component implementations, rather than quadratic. Interchangeable components only provide transform functions for a shared abstraction, rather than a different transform function for every known implementation.

Along with these goals, our work also includes a set of assumptions and restrictions. While some of these assumptions could be loosened to generalize our system for future work, they would also introduce substantial complexity into the system.

- Components communicate exclusively through interfaces provided by the runtime system. Special mechanisms would be required to preserve transparency for lower-level communication mechanisms, such as handles for open files.
- Components to be evolved are limited to those on the same host; applications requiring multi-host reconfigurations would require a distributed run-time that tracked and controlled all interfaces.

- The interface of the component is not allowed to change over time. If we allowed backward compatible changes that only added to the component interface, we would be making it impossible for the application to later replace this component with the original (older) component implementation. We desire implementation changes in our system to be reflexive.
- It is possible to motivate standards committees to generate component specifications that meet the requirements of component clients and implementors.
- Programmers are cooperative, not malicious, and components are implemented correctly. There are various application properties that cannot be enforced by the runtime system; for example, the component must not intentionally deadlock and purposely avoid reconfiguration. In fact, careful component implementation will be required to assure timely reconfiguration[1]. In our system, *starvation is possible* if threads do not behave cooperatively and allow instance evolution to occur.

2.2 Component Model

Applications in our system rely on software modules that can be replaced dynamically without affecting the behavior of the rest of the application. The term *component* refers to a particular abstract entity in the application that can be replaced.

In our system, a component is specified with two abstractions. The first abstraction is an abstract datatype, which includes the declarations of methods and constructors used by the clients of the component. We call this the *component type*. The second abstraction is used to manage instance state when the current component implementation is replaced with another.

We call the construct used to define the component interface containing these specifications a *component mold*. The use of this terminology is used to distinguish our approach from the traditional language-level interface, which only provides part of the abstraction required for a component. The mold construct interfaces *all* interaction between using code and the underlying implementation, including the interaction between two different implementations of the component during instance replacement. The elements of the component mold are discussed further in section 2.2.1.

A component is implemented by a class that satisfies the behavior specified in the component mold. We refer to this declared relationship between the class and the component mold as the *meets* relationship. There can be many different classes that *meet* the component; these classes can be used interchangeably. Every application has a one-to-one run-time mapping between each component and a class that implements the corresponding component mold. We refer to the class in use at a particular moment as the *mapped class*. This work presents mechanisms that allow the dynamic adjustment of this mapping from one class to another. These mechanisms are introduced in Section 2.4.

We call the adjustment in the mapping of component to class a *reconfiguration event*. A reconfiguration event can occur at any time and with any frequency at runtime, either due to direct request from the user or automatically as determined by specified system metrics. These events take the form of a pair consisting of a component and the replacing class, which then becomes the

[1]This assumption is similar to those made for cooperative multi-tasking

new mapped class. Note that under this model, a class can be mapped to a component more than once over time, allowing changes in implementation from class A to B and then back to A.

Application developers use the component type specified in the component mold as a concrete class would be used. The using code creates instances of the component by invoking a constructor declared in the component type. This causes a matching constructor on the mapped class to be invoked. The created object has known methods (declared in the component type) which can then be invoked. These invocations cause the corresponding method implementations in the mapped class to be called.

The client code instantiating a component type receives a reference to a special *component object* that is not identical to the object being constructed by the underlying mapped class. We call the object constructed by the mapped class the *underlying object*. The component object provides a level of indirection to the underlying object. This separation between the component object and the underlying object is necessary; when reconfiguration events occur, existing component objects are adapted to use instances of the new class as the underlying object. These objects are described in more detail later on in this section.

We call this process of adapting existing component objects *evolution*. During evolution, the existing underlying object transfers all of its state to an instance of the new mapped class, which uses this state to create its own equivalent concrete state. After evolution, the new mapped class instance becomes the underlying object, and provides the state and functionality of the component object. The identity of the component object remains unchanged during evolution; only the underlying object has been replaced.

As soon as each component object reaches a state that allows it to be evolved correctly, it will evolve independently of other objects of the same component. Much of the rest of this thesis focuses on mechanisms for determining when evolution for an object is possible.

Our system is multi-threaded. Any on-going method invocation made by a thread on the component object is preserved after evolution. The thread is able to continue execution at an appropriate point in the new underlying object. Trivially, this could mean that all on-going method invocations must terminate before evolution could occur. By preserving method invocations consistently, evolution of a component object are transparent to clients of the object.

Underlying objects interact exclusively through shared abstractions during evolution, as described in Section 2.4. This means that these objects are unaware of the particular class of the object they are being replaced with. This allows a new reconfiguration event for the same component to occur before a previous reconfiguration event has *completed* (meaning all component objects have evolved). Component objects can always be evolved to the mapped class of the most 'recent' reconfiguration; out-dated reconfiguration requests can be discarded. Arbitrarily many different implementation classes might be in use at a given time for a particular component if existing objects can't achieve timely evolution. There is only one mapped class at any given time, however, which is used when new component objects are created.

The component object is an entity with seemingly contradictory semantics. It is a concrete object in that is created, has an identity, has state that can be modified, and can be referred to and invoked. At the same time, the component object also has abstract characteristics in the sense that its representation (which is inaccessible to clients) and behavior is entirely abstract and relies

on the underlying object representation, which could change over time. Section 2.2.2 explores this relationship in more detail.

Often, a component implementation requires several classes in addition to the one that directly meets the specification given in the component mold. Section 2.3 describes these classes and the way they are used within the application.

2.2.1 Component Mold

The component mold describes two separate interfaces to provide different abstractions, as shown in Figure 2-1.

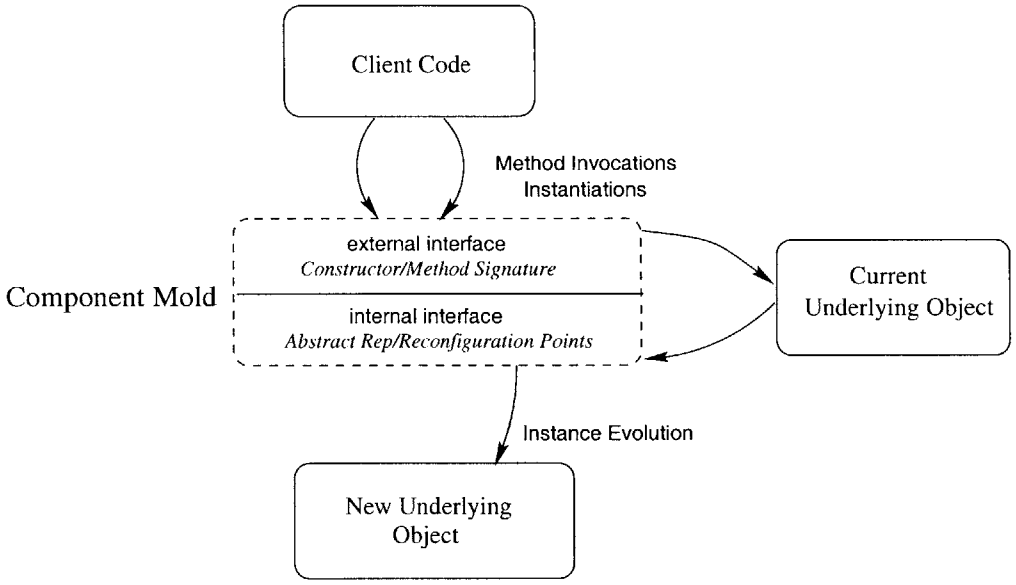

Figure 2-1: Component Abstraction

We call the first interface the *external interface.* This is the interface that gives us the component type. The external interface hides the implementation class from the client, allowing the class to change over time. This interface consists of a set of declared methods that can be invoked by the client. The concrete implementation used to handle method invocations from the client is selected transparently and changed by the run-time system as needed. Constructor declarations are also included, allowing clients to instantiate objects through the external interface.

Unlike some interface constructs in object-oriented programming languages, the external interface does not allow concrete field declarations. Declared concrete fields are usually compiled into direct accesses to some precise memory location in the referred to object; this constrains a class implementing the component to use an object representation that places this field at that fixed location. Instead, fields can be emulated while preserving implementation flexibility by using paired abstract gettor/settor methods; this allows the class to implement the semantics of the field using any representation it prefers.

The exact form of method signatures differ from language to language. Generally, method signatures can specify argument types, typed return values, thrown exceptions, and synchronization properties. Arguments can be further specified with names, orderings, reference kind, type, and marshalling properties. Our interface is intended to be general and should be usable with any form of method signatures if mechanisms for evolution are provided. In this section, we focus on simple method signatures with typed argument passing and return values. We present a Java-specific version of our abstractions in Chapter 3.

The second interface is the *internal interface*. It is used to abstract from implementation details related to evolution so that component implementations can be written independently. The details of the internal interface are irrelevant (and made inaccessible at run-time) to the client.

The internal interface frames the mapping of existing object state between component implementations without requiring knowledge of each other's object representation. The internal interface does this by declaring an *abstract rep* for the component. The abstract rep is the external reference representation for a particular component, which is shared during evolution by both classes implementing the component. The way the abstract rep is used during evolution is discussed further in Section 2.4.1.

The internal interface is also used to abstract away method implementation detail and allow the mapping of on-going method invocations between different implementations. The internal interface specifies *reconfiguration points*, which are labels associated with each method. Reconfiguration points are used to specify the abstract behavior of a particular method by naming expected points in the computation of the method. Section 2.4.2 describes reconfiguration points in more detail.

The component mold has a type-name that uniquely identifies the component it is specifying within the application; this type-name remains static during the life-time of the application in order to assure type safety. In a well-typed program, all client invocation on a component object will satisfy the declared method signatures in the external interface. Similarly, a class implementing the component is well-typed if it satisfies the internal interface by providing appropriate definitions for reconfiguration points and the abstract rep. A compiler could be implemented to confirm that component and application implementations meet these requirements.

In our system, the component mold specification for the external and internal interfaces is only syntactic in nature. The component mold could also be extended to declare semantic constraints for these interfaces. For the external interface, the abstract data-type could be defined by a specification (eg, [27]). Similarly, the abstract rep and reconfiguration points could be specified further to capture semantic meaning; how this could be done is an open research question.

For the purpose of this work, the need for capturing precise component semantics must be balanced with the complexity the extra information introduces into the application design process. The modifications proposed here should not be a burden for existing commercial component developers, but are also insufficient for enforcing properties such as reliability and security on interactions between components.

2.2.2 Component Object

Code using the component do not interact directly with the class implementing the component mold, but instead uses component objects that are abstract in representation and behavior. From the perspective of the application, the state of the component object is fundamentally abstract, and equivalent to an abstraction function applied to the underlying object. The methods that can be invoked on the component object are also abstract, and satisfied by concrete implementations from different mapped classes over time.

The notion of instantiating an abstract object is unfamiliar, since it is not allowed with other abstraction mechanisms such as Java/C++ interfaces and virtual classes. However, it is a necessary addition to our component model in order to preserve *instance transparency.* Instance transparency is the requirement that any changes in implementation remain transparent to clients of *a particular instance.*

Instance transparency means that even though the component object uses different underlying objects over time, we will preserve fundamental properties associated with the identity of the abstract object. For example, at the very least references to the current instance must remain correct after evolution. Other constraints on object identity also exist. In Java, synchronized locks are held on a particular object, which represents another kind of reference on the object that must remain correct. Java also associates object identity with the methods `Object.hashCode()` and `Object.equals()`; consistent responses from a component object to these kinds of methods are critical.

The component object helps *multiplex instances with different concrete types on to the same abstract instance.* This differs from traditional language-level interfaces that are intended to *multiplex concrete types on to the same abstract type.* A Java interface allows a variable declared to be of this type to refer to objects of any concrete type that defines a subtype of the interface. Over time, the variable could point to *different* instances with differing concrete class types. However, any particular object has a single consistent concrete type including object representation and implementation.

Semantically, the underlying object is separate from the component object. When using code calls a constructor declared in the component type, the component object is (conceptually) created and allocated on a particular location on the object heap. The underlying object is then allocated elsewhere, and created using the corresponding constructor on the mapped class. As evolution occurs, the underlying object would be discarded while the component object retains its identity. The mapped class is not a sub-type of the component type; instances of this class cannot be cast and referred to in place of the component object directly.

This view of the component object does not necessarily suggest that it must be implemented as a separate surrogate object that proxies references between clients and the underlying object. For example, existing client references could also be preserved using an indirection table which points to the current underlying object. Other aspects of object identity could then be preserved by providing a small standard structure as a "cookie" object passed to the new underlying object during evolution, which would be used to emulate identity as needed (for example, the cookie could include the correct value to return for the Java `Object.hashCode() method`).

A small example based on human interaction helps to clarify the distinction between the abstrac-

tions provided by interfaces and those provided by the component object. Imagine Fred waiting to start a conversation; an interface abstraction might indicate Fred is prepared to speak with any human being who speaks English. This way, as long as the other conversationalist happens to speak English, Fred is prepared to initiate the discussion. Over time, Fred might have different conversations with different people, as long as they satisfy this basic interface. However, since Fred recognizes the difference between Bob and Mary, he would start a new conversation when Bob was replaced with Mary, even if both are English-speakers. Bob and Mary are both multiplexed, in time, on to the role of an English-speaker.

The component object abstraction is different. Imagine that Bob wears the mask and identity of Joe when speaking with Fred. When Bob grows weary, Mary takes over the mask, pretends to be Joe, and continues the conversation. Fred's perspective is that he is speaking directly with Joe at all times. The component object is equivalent to the mask and identity of Joe; Bob and Mary are multiplexed on to the identity of Joe.

2.3 Component Implementations

Multiple classes might be needed to implement a component. To distinguish among these classes, we refer to the one that meets the component mold as the *primary class*. Underlying objects are instances of the primary class. Other related classes are used by the primary class to fully implement the behavior of the component, e.g., to do special work for that particular implementation. Component implementations are deployed in a unit of deployment called the *component package*. Every component package includes at least a primary class, as well as other classes related to the implementation.

The types defined within a component package are isolated from the rest of the system for two reasons. First, this prevents type collisions between different packages if they offer different definitions of the same type. This might occur in the case of different 'versions' of component packages from the same source that retained the same type names. Second, this prevents classes in the rest of the system from explicitly naming and using classes deployed within a component package directly, and thus becoming "too friendly" (either due to laziness or maliciousness) with a specific component package.

Under the basic model for component implementation, the primary class is always loosely bound to its clients through component interfaces. The use of this abstraction assures that reconfigurations will be transparent to these clients, and permits replacement of the primary class over time. However, use of the component object introduces run-time overhead, and the component type restricts clients from using the fields and methods of the concrete primary class type directly. In practice, implementing the behavior of a component would be easier if we relax the use of abstraction (and sacrifice transparency) in certain contexts. This allows the implementation to trade off performance and ease of development versus transparency across reconfiguration.

In this section, we explore ways the use of abstraction can be relaxed within a particular component implementation, provided all of the affected code in the implementation evolves together.

2.3.1 Friendly Classes

Applications are built of two distinct kinds of classes.

First, there may be classes that are tied to a particular primary class, or equivalently, a particular component implementation. We call these classes *friendly classes.* A class is friendly if it uses a direct interface to the primary class that allows more efficient access. The relationship a friendly class has with a primary class is similar in intent to the `friend` binding in C++ and package classes in Java; it allows certain abstraction rules to be discarded in limited contexts for ease of implementation and performance.

We also use the term *primary-specific* to denote a tight binding relationship with a particular primary class and component implementation. Primary-specific constructs and definitions are only valid within the scope of a particular component implementation. Thus, a friendly class is also any class that is primary-specific.

The cost for this closer relationship is that friendly classes are directly affected by reconfigurations; they use primary-specific interfaces (i.e., interfaces that are not the component type) to access the primary class that would no longer be valid once a reconfiguration event caused the current primary class to be replaced. Friendly classes are not permanently part of the code-base of the application; they are removed along with the primary class when the component mapping changes.

There may also be classes that are not primary-specific. These are called *standard classes.* Standard classes remain available for the lifetime of the application since they are independent of any primary-specific code, and thus unaffected by reconfigurations. By independent we mean these classes do not directly access fields or invoke methods that are specific to any particular component implementation; standard classes also can not contain any fields with a primary-specific type that would allow these types of operations. This means that only standard class can be declared as the types for fields, method arguments, and return values for other standard types.

Examples of standard classes include the Java class libraries that are used throughout the application, as well as the classes that represent the "core" of the application. These core classes provide the fundamental behavior of the application that cannot be generalized into component molds, and will remain independent regardless of any application reconfiguration.

Only these standard classes can be used safely shared throughout the application, including use by friendly classes. For example, component types are considered standard, while the primary classes used to implement the component type are not standard.

A component package can also provide standard class implementations. These classes must provide behavior that remains relevant beyond the current component implementation, and avoid using primary-specific types. Standard classes are useful members of a component package because other classes in the component package can refer to standard classes with a primary-specific type, allowing more efficient direct access to fields and methods.

Standard classes are preserved in the application when the rest of the component package is removed; this allows instances of these standard classes to remain available even after the component package the standard class was deployed in is reconfigured away.

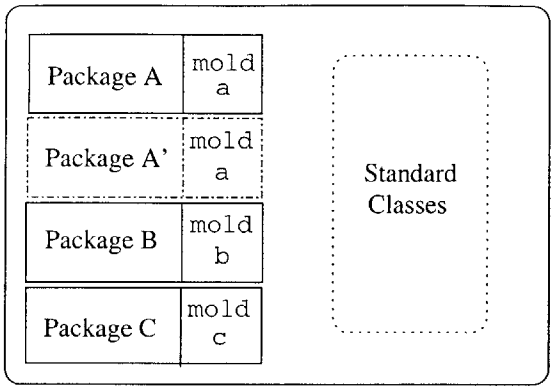

Figure 2-2: Application Code Architecture

For example, a component package might include a standard class used to represent exceptions. This class would expose an application-wide standard interface that makes it usable to other classes in the application. An object of this class might first be initialized and then accessed efficiently with primary-specific methods while it propagates through the friendly classes of the current component. As it continues to propagate into the rest of the application, it would remain relevant and useful to the rest of the application (although now only through a standard interface) even if the component package it originated in has been reconfigured away.

Figure 2-2 shows a snapshot of the application architecture with concrete packages brought in to fulfill component molds. Note that since component objects evolve at different times, two different component packages are simultaneously fulfilling the component mold a. The primary class provided by the more recent component package A is the mapped class, and thus used to construct all new instances of a.

2.3.2 Internal Instances

Instances in the executing application are divided along similar lines of whether they survive dynamic change. Just as certain primary-specific classes cannot be shared throughout the application because they violate the abstraction of the component type, there are also instances that cannot be referred to throughout the application because doing so violates the abstraction of the component object.

First, there are instances in the application that are tied directly to a particular underlying object; we call these *internal instances*. Internal instances break the layer of abstraction that component objects are intended to provide by referring directly to the underlying object. Thus, transparency during evolution is lost. When the underlying object is detached from the component object and loses its identity within the application, internal instances that hold references to the underlying object also become invalid. The same argument transitively tells us that any object that references an internal instance must also be internal.

Any instance of a friendly class is internal by definition; if the friendly class itself does not

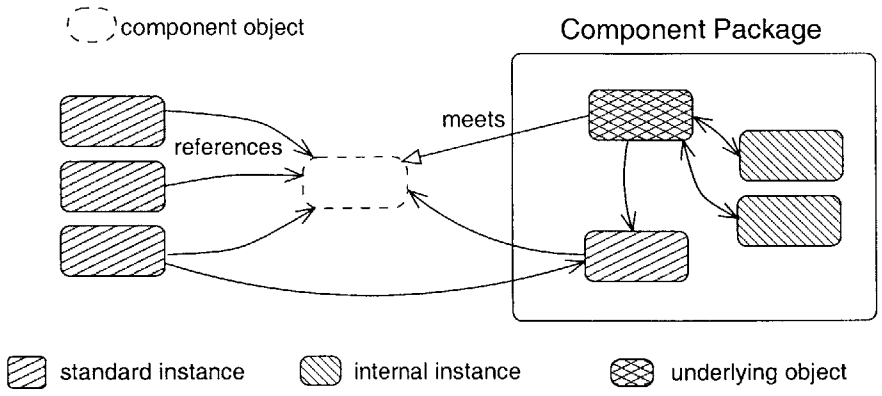

Figure 2-3: Component with Client

survive reconfiguration, it makes little sense for an instance of the class to survive evolution when the component object becomes implemented using classes from a different component package.

The critical property of internal instances is that *references to internal instances cannot escape the object-graph that is contained within the component object.* This means that internal instances cannot be used as method arguments or return values to other objects that could out-live the current underlying object in case of evolution. This restriction also applies to other component objects of the same component type, since each object evolves independently. In essence, internal instances are limited to being part of the primary class rep. Since no outside references are allowed to internal instances, their removal upon evolution is transparent to clients of the component object.

The second kind of application instances are those instances not removed by evolution; therefore, such instances can be used throughout the application. We call these instances *standard instances.* Standard instances by definition can not refer to internal instances. References to standard instances can be safely held by any object in the active application; this allows them to be used as method arguments or return values. Component objects, for example, are standard.

Standard instances can only be created from standard classes, since the type of the object must also survive reconfiguration if the object is to survive evolution. Note that standard instances *can* be referred to by internal instances.

Figure 2-3 shows the interaction between the different kinds of instances and classes in an application. The component package box contains the classes that were deployed together. All of the classes in the package can have primary-specific types that allow other classes in the package direct access to fields and methods that might otherwise be hidden behind the component type.

In this figure, two instances of friendly classes hold direct references to the underlying object , and vice versa. A standard instance deployed as part of the component package (shown in the box) is also referred to by the underlying object. However, the standard instance does *not* hold a reference to the underlying object. Instead, it uses a reference to the component object when invoking methods, although the methods will eventually be executed by the underlying object. As a standard instance, it can be passed to other objects in the application as a method parameter or return value; these other objects can then directly refer to the standard instance.

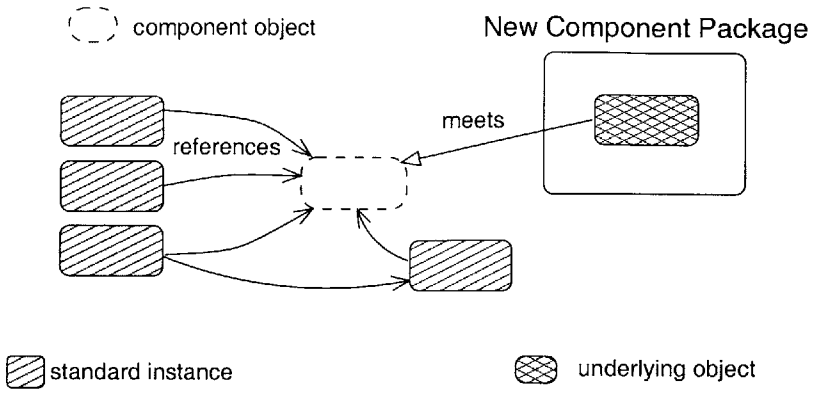

Figure 2-4: Component with Client after Evolution

Figure 2-4 shows the same component after evolution, contrasting the effect on standard versus internal instances. The underlying object is removed as part of the evolution process, with the component object re-directed to refer to an instance of the new primary class. The internal instances that referred to the previous underlying object are no longer valid, and therefore removed as well. The friendly classes used to implement the internal instances are also removed, along with the rest of the component package, from the application.

The standard instance, in contrast, did not refer to the underlying object. Its out-going reference was toward the component object, which remains unchanged through evolution. Thus, the standard instance remains a valid object in the application. The standard class used to implement the standard instance is not removed along with the rest of the component package.

If implemented naively, the distinction between standard and internal instances would be very expensive to maintain at run-time. The system would need to monitor assignments and method invocations to confirm that internal instances are not referred to inappropriately. However, a correct component implementation should be aware of this distinction, and take care not to share references to internal instances.

Static typing systems introduced in [8] and [4] represent a direction of research that might lead to a compile-time mechanism for assuring separation between different component objects. These systems use parameterized types to protect certain objects from being aliased outside a certain scope.

2.4 Preserving State for Instance Evolution

Application reconfiguration requires the instance of a primary class to be transparently evolved over time. Our definition of transparency requires that all relevant state held within the instance be preserved consistently into the new instance. Our approach relies upon the use of abstraction provided by the internal interface. All of the needed state from an existing instance is transformed into an abstract representation defined by the internal interface, which is then used to initialize an instance of the new implementation.

There are two different issues here. The first challenge is to define the specific elements needed in the interface to capture all of the relevant instance state while satisfying our system requirements. Since an appropriate function for mapping instance state to the abstract representation might not be available at all times, the second challenge is to find the conditions used to determine *when* instance state is consistent and evolution is possible.

Different abstractions in the internal interface are used to capture different forms of the instance state. We assume that all relevant instance state exists in only two different contexts: the per-instance context (section 2.4.1) and per-invocation context (section 2.4.2).

The state in the per-instance context consists of the object's concrete representation. An abstraction is needed to support mappings between different primary class instance representation. The per-invocation context includes the instance state associated with on-going computations in the form of method invocations. The threads currently executing within an instance require an abstraction for mapping on-going method invocations to the new instance.

One way of characterizing the difference between these two contexts is that state in the per-instance context is stored on the heap, while state in the per-invocation context is stored on the stack.

2.4.1 Per-Instance State

The state in the per-instance context consists of the object representation. Although language and system-dependent, we assume this representation is in the form of named and type member fields that store references to objects. This is analogous to traditional definitions of object state. The object representation is stored on the heap and has a static layout generated during compilation. Differing implementation will likely have very different object fields, resulting in incompatible heap state representations.

Distributed computing systems have used Abstract Data Types (ADT) for the mapping of object representations into external representations for transmission [18]. We propose the use of similar ADT-based abstractions. We use *abstract rep* declared as part of the internal interface to provide an implementation-independent abstract encoding of the instance state during evolution. Primary class implementation state is mapped to and from this abstract representation. Similar techniques have also been long applied in distributed computing for the mapping of object representations into external representations for transmission.

The abstract representation is declared as part of the component mold in a form that is language-dependent, but we again assume it consists of references to objects. For Java, the abstraction representation is in the form of named and typed fields. The declared types of these fields must be standard, which means a field can be typed as a standard class, a standard interface, or a component type. The external representation also allows structured types (such as arrays). The abstract rep is never made available to clients of the component object; fields in the rep are only accessible to primary class instances during the evolution process.

During evolution, the underlying object first *encodes* its current concrete representation into the abstract representation. Space is temporarily allocated elsewhere on the heap for the abstract representation. In the encoding process, the underlying object sets fields in the abstract rep to

refer to objects that capture current instance state. These objects may be created dynamically during the encoding process; this would likely occur if the concrete primary class representation is drastically different from the abstract representation. Alternatively, the abstract rep might refer to existing instances.

Only standard instances can be stored in the abstract representation. State transmission through the encode/decode clauses is no different than method arguments: internal instances that are tightly bound to the previous primary class instance cannot be used by other objects, including the new replacement primary class instance.

An instance of the new primary class completes the evolution process by providing a *decode* transform function that initializes its implementation-specific representation from the abstract representation. An invariant in the implementation of the decode and encode transform function is that if the new decoded primary instance is immediately encoded, it should create the same abstract rep used to generate this instance. In practice, this does not necessarily mean the recreated abstract rep must include references to identical objects, but rather just that the two abstract reps are equivalent with respect to the specifications of the component object. Figure 2-5 shows that as part of this process, references to instantiated objects or components can be preserved, or new objects can be created.

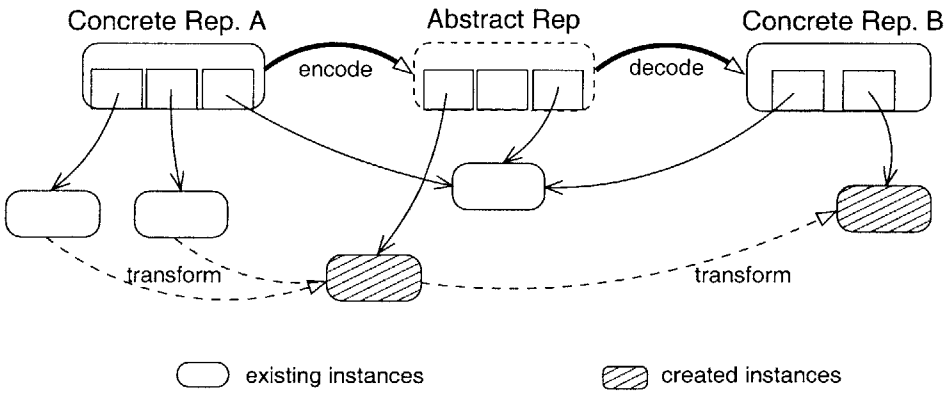

Figure 2-5: Abstract State Encode/Decode

One significant advantage of attaching the encode and decode clauses directly to the primary class implementation is that the transform functions for transforming the local object state into, and from, the abstract instance state are always available. This allows the reconfiguration event that leads to evolution to include only a reference to the newest implementation of the component, instead of also providing a specific transform function for evolving instances of the current implementation to the new one. With this simpler model of the reconfiguration event, older reconfiguration events can simply be discarded even if they have not yet completely evolved all objects. Any existing component object, regardless of current primary class implementation, is capable of evolving to the primary class specified in the most recent reconfiguration.

The encode transform function is not able to map current instance state to the abstract representation at all times. Parts of the state defined in the abstraction might not be available if on-going computation has left this instance's per-instance state inconsistent. This leads to our first condition for determining whether an instance is prepared for evolution:

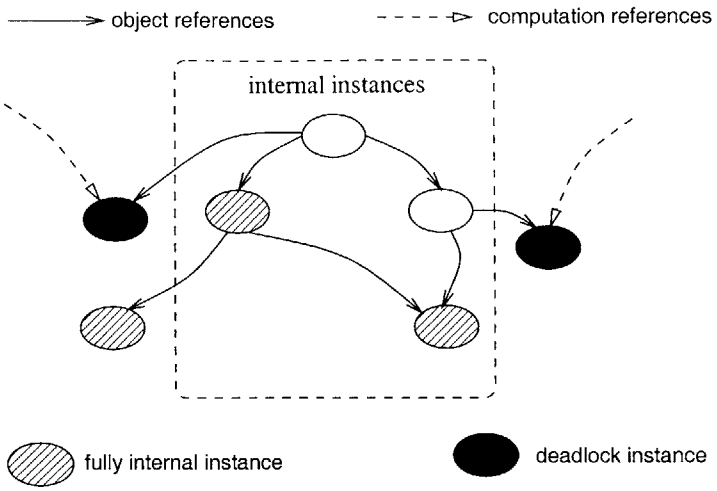

Figure 2-6: *Fully internal*, and instances that could cause deadlock

Condition 1A. Heap-state mappability: The implementation-specific representation on the heap must be consistent and mappable.

Fully internal instances

While an instance is in the process of being evolved, the object state can not be modified by other threads since it could render the object state inconsistent. As described later on in this chapter, application threads will block at appropriate points awaiting completion of evolution. Once the object has been completely evolved, these threads can continue execution.

Imagine that a thread blocked waiting for the object to evolve is holding synchronization locks on objects in the application heap. If at some point during evolution the system thread evolving the object (the *evolution thread*) tries to acquire the same lock, the application would become deadlocked. The evolution thread has become blocked waiting for a shared resource, which is in turn held by a thread waiting for the evolution thread to complete the evolution process. Deadlock occurs only if the evolution thread tries to acquire a lock already held by an application thread.

Note that only locks on objects outside of the component object (which are also standard instances) can be the source of deadlock. Locks on an internal instance would not be expected to survive evolution, meaning the lock can be discarded during evolution. Furthermore, only standard instances that are reachable from a current method frame can be locked. If the application thread can not refer to the object, than it certainly could not have acquired a lock on it. We call these objects that are both reachable by an application thread and outside of the component object *deadlock instances.*

Therefore, the evolution thread should never acquire locks on deadlock instances. This also means the evolution thread must avoid invoking methods with unknown implementations that *could* acquire a lock on a deadlock instance.

We introduce the term *fully internal instances* to denote the kind of objects that are safe for the evolution thread to invoke methods on. Figure 2-6 shows a collection of instances. The solid black ellipses represent deadlock instances, since the dashed arrows indicate that these objects can be reached and locked by an application thread. These are the objects that the encode and decode clauses should avoid locking (or invoking a method on). The clear ellipses indicate internal instances that may also be unsafe since they refer to deadlock instances. Methods on these instances should not be invoked either, since they might transitively invoke methods on a deadlock instance.

The patterned ellipses represent fully internal instances. These objects do not hold references to deadlock instances, and thus is in no danger of invoking an unsafe method or acquiring an unsafe lock. They are also not reachable from any application threads, which prevents them from being locked. The encode and decode clauses can invoke methods on these objects with no danger of deadlock. Note that a standard instance (outside of the dashed box) is also considered a fully internal instance in Figure 2-6. This is possible if the standard instance was just created within the encode and decode clauses, for example, and no application thread could possibly refer to it. Even if the standard instance became referred to by outside objects after evolution has completed, there is no danger of deadlock during evolution itself.

This restriction does not prevent the evolution thread from directly accessing the fields of instances that are not fully internal, or assigning references to instances that are not fully internal. For example, the encode transform function can still safely assign deadlock instances to fields in the abstract rep.

This restriction is expensive to enforce within the run-time system since it requires sweeping the entire object heap to determine whether a particular object is referred to elsewhere. In addition, such a run-time system may be over conservative if it prevented invoking methods on instances that could not have led to deadlock. Instead, we offer this as a guideline for correct component implementations. The developer should develop transform functions to meet this constraint, possibly by providing specialized method implementations for internal instances that only invoke methods on other objects that are fully internal.

2.4.2 Per-Invocation State

The per-invocation context includes all of the state related to on-going computations in the form of method invocations. Maintaining transparency and consistency for state in this context means that upon evolution, each thread that is participating in an on-going computation will be able to continue execution consistently in the new instance. An abstraction mechanism is used to map state associated with on-going computation for each of these threads from the old to the new implementation.

Like method signatures, the specific state within this context can vary from system to system. For purposes of discussion, we assume local variables, method arguments, return addresses, return variables, and program counters are included in the state in this context, and must all be held consistent after evolution. The per-thread stack is usually used to hold this state in the form of method frames. Registers are often used as well to store per-invocation state (generally used for argument passing and the program counter), but usually only as an optimization; this state could equivalently be preserved on the stack.

A thread is considered to be involved in an on-going computation on the instance if it is currently *active* in an instance method. This includes threads directly executing code within an instance method, as well as threads that are executing code in a method invocation made from an instance method. In both of these cases, the stack for this thread contains a method frame with primary-specific state that controls its interaction with the instance. The only distinction is whether the state related to the instance method is held at the top of the stack or not.

Some of the state in the per-invocation context is directly mappable across implementations. For example, state that is declared statically as part of the shared method signature declaration in the component mold is used equivalently by all primary classes meeting the mold. Method parameters can be mapped directly without need for any abstractions. The return address is also directly mappable since method invocations from clients are made without knowledge of the actual implementation, and thus all implementations return equivalently. Thread resources acquired outside of the current frame, such as locks, can also be transferred directly to the new execution environment as well.

Local variables

In most systems, state on the stack also includes local variables used to record temporary results from the current method invocation. An abstraction could conceivably be provided for mapping local variables from method implementation to method implementation via the use of declared *abstract rep*, just as was used for mapping per-instance heap state. However, it seems unlikely that meaningful abstractions can be created in the internal interface for local variables at the method level without seriously constraining the underlying implementations.

We instead assume that local variables are used only to represent transient state in the component object. If the state in some local variables is *not* transient and should be preserved for future instances of the component object, then this part of the local state must be explicitly stored within the concrete rep of the instance before the per-instance state of the object should be considered consistent. These local variables would then be encoded along with the rest of the concrete rep upon evolution.

Although method arguments are stored in the method frame the same way local variables are stored, they are treated differently upon evolution. Any changes the local method implementation makes to the argument parameters will be mapped as-is to future primary class instances. Method arguments are declared as part of the method header, and have meaning that is consistent across different method implementations.

Program counter

The most problematic area for evolution is in mapping the state represented by the program counter, which is used to track progress in the current computation. The difficulty lies in that control flow and method implementations can vary substantially between different component implementations.

The obvious and naive solution is to simply eliminate state of this form by waiting for all on-going method invocations involving the instance to complete, which could also mean throwing

exceptions to force early completion. By waiting for the per-invocation context to become empty, the system only needs to map the per-instance state as described previously. However, one key requirement of this system is that evolution must be completed in a timely manner. Furthermore, using exceptions to force termination would introduce additional complexity, since this implies each invocation must be prepared to exit at any point.

This work proposes the use of *reconfiguration points* as the abstraction used to map the program counter from one implementation to another. Reconfiguration point abstractions are declared alongside the method header in the component mold and act as well-defined points in the execution of that method. Essentially, reconfiguration points provide abstract names for places in the program flow where different implementations of the same method 'meet'. The program counters can be considered to be 'equivalent' at these points and are mappable from one implementation to another.

This suggests another condition for evolution based on the notion that only certain program counter values are mappable:

Condition 1B. Stack-state mappability: The program counters for on-going computation involving this instance must be at a well-defined reconfiguration point.

Both condition 1A and 1B can be merged and satisfied by a sufficiently broad definition of reconfiguration points. When these conditions are satisfied by computation threads, the state in the per-invocation context can be mapped from one primary class to another. We present the complete definition of reconfiguration points:

Reconfiguration point: an abstraction for a certain point in the computation of an abstract method. The concretization of a reconfiguration point takes the form of a specific point between two program statements in the method implementation. At this point in the method implementation, the computation must have left the instance object state consistent and mappable to the abstract representation.

The above definition makes a claim that is really unsupportable. The computation in the method implementation cannot ensure that over-all object state is consistent and mappable at the reconfiguration point. Other threads might be conducting other computations that have modified the object state. However, we assume that if the method implementation assures the current computation up to a reconfiguration point executed by a single thread has left the object state consistent, the union of all modifications by all threads at reconfiguration points will also leave the object state consistent. Thus, condition 1A is satisfied only once *all* of the threads executing on-going computation within the instance have reached reconfiguration points.

Figure 2-7 shows the evolution process. The reconfiguration points are shown as black bars in this figure. After method invocation, the program counter tracks the computation as it moves through implementation A (shown by the white arrow). Computation continues through the first reconfiguration point without interruption if evolution is not pending. The program counter might loop over the second reconfiguration point multiple times before a reconfiguration event occurs. If the program counter does not point at the reconfiguration point when the evolution event first occurs, the conditions 1A and 1B are not satisfied, and evolution is not possible immediately.

Under the rules we have proposed, condition 1B is satisfied for this thread once the program counter returns to the reconfiguration point. The program counter is then mapped (along with other relevant per-invocation state like method parameters and the return address) to the corresponding point in method implementation **B**, completing the computation as expected. The beginning and

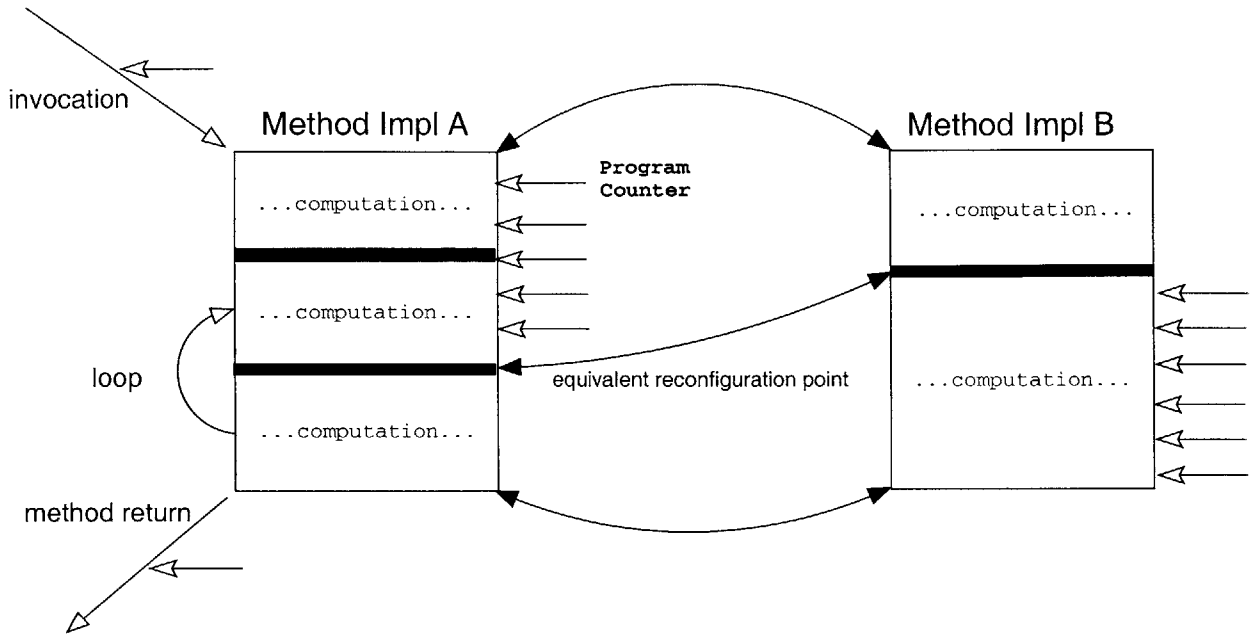

Figure 2-7: Method invocation with reconfiguration points.

end of methods can also be thought of as implicit reconfiguration points with labels `BEGIN_METHOD` and `END_METHOD`; they have equivalent abstract meaning in all method implementations.

2.4.3 Role of Reconfiguration Points

Reconfiguration points are placed within the methods (and constructors) of the primary class by the developer to satisfy the mapping requirements for state in both per-instance and per-invocation contexts.

First, reconfiguration points satisfy condition 1A by indicating object heap-state consistency. They play a role analogous to transaction commits by indicating that the current computation should be considered complete and all heap-state invariants hold, even if the method invocation is not yet complete. By placing a reconfiguration label at a certain point, the programmer is asserting that a correct mapping exists from the current object state (from the perspective of this method implementation) to the abstract rep at that point.

Second, reconfiguration points are used to capture the meta-state for each on-going computation. Their role in this process is in naming and identifying the specific point in the behavior the current computation has reached, by acting as an abstract label for the program counter. Just as method invocations across an abstract method declaration succeed regardless of the underlying implementation, reconfiguration points represent an abstraction that we can use to map an on-going computation from a reconfiguration point in one implementation to an equivalent point in a different implementation. The definition of reconfiguration points indicates that both implementations should have equivalent behavior at that point. A method invocation can complete partially in one

method implementation, become mapped over to an equivalent point in a different implementation with equivalent state, and then finish computation there. This allows evolution to occur without waiting for or forcing completion of all pending method invocations, including potentially very long-lived event loops.

Recall that abstract method declarations must already specify (informally) the behavior that must be supported by the underlying implementations. Reconfiguration points extends the extent of this specification only slightly by naming reasonable points in this behavior where evolution might be desirable. This should not represent a significant additional burden to the developer. The developer also has ultimate control over where and how to match their method implementations to the method behavior described abstractly with reconfiguration points. If the method implementation deviates significantly from the declared norm, than on-going method invocations could not be mapped successfully. In this case, declared reconfiguration points can always be placed at either the beginning or end of the method invocation where all method implementations share the same behavior. At worst, this would reduce timeliness since evolution must now wait for method completion.

An alternative method of segmenting computation to allow timely evolution is to implement multiple smaller methods. The role of reconfiguration points in indicating completed computation is now satisfied by method returns. However, there are two major flaws with this approach. First, it increases the complexity of development of component implementations. Simple constructs such as loops containing one of these computations would have to be 'pushed-up' an invocation level to the client method. Second, the number of method invocations would increase dramatically, which leads to significant performance loss.

There is a significant weakness in the reconfiguration point mechanism. As stated before, the reconfiguration point is placed *between two statements* in the program code. Thus, reconfiguration points can only be used to map state from the method frame at the top of the stack where the program counter is active. That means method frames lower on the stack, where the program counter is currently referencing a method invocation, can not be mapped.

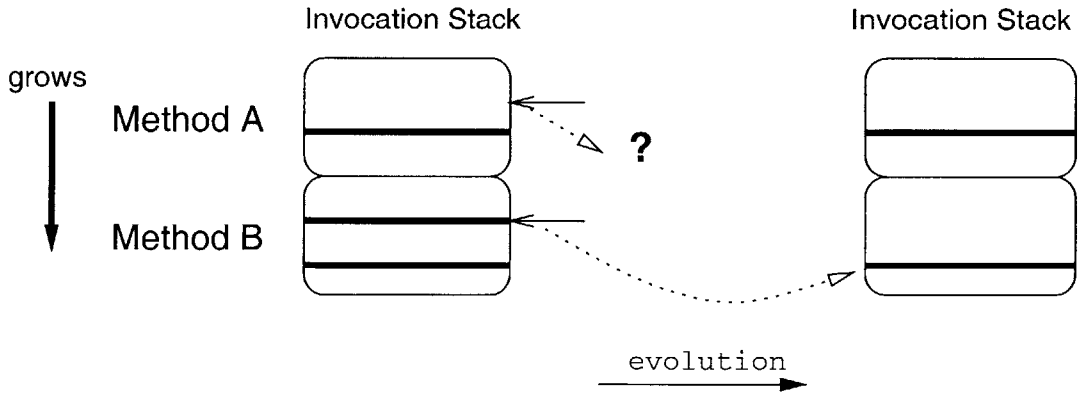

Figure 2-8: Blocked evolution.

Figure 2-8 shows this problem. Pending method invocations from a method A within an instance will block evolution for the instance if there is another method B on the stack. Even if method B is also a method on the same instance, and the program counter in the method frame for B is at a

reconfiguration point, evolution is still not possible. The program counter in B might be mapped over to a corresponding value in the new method implementation. However, the program counter in method A is still pointing at a pending method invocation (which led to B) that other primary class implementations might not have made. No mapping exists from this point in the method A to the new implementation.

Therefore, evolution for an instance is only possible if all relevant threads have only a single method frame on stack for that instance. We propose optimizations to improve upon this result in Chapter 4.

2.5 Summary

In this chapter, we have provided our description of a system that allows active component instances to evolve into differing underlying implementations. We have designed a framework for component mold specifications that allows for independent development of components. The reconfiguration point abstraction, in particular, allows evolution without requiring either heavy-weight transactional support or waiting for long-lived method invocations to complete.

Chapter 3

Language Framework

This chapter presents source extensions based on the Java platform [15] that support the abstractions and mechanisms introduced in the previous chapter. Java, the programming language, run-time system, and associated technologies are excellent examples of recent advances in commercial systems toward supporting the application domain this work is intended to support. The design presented in the previous chapter is not intended to be language or platform specific, and similar design extensions should be possible for any combination of object-oriented platforms and languages, including CORBA, COM, and C++.

The rest of this chapter focuses on additional mechanisms available to the developer for creating components. Section 3.1 gives a brief introduction of the way abstraction is used within Java, as well as other relevant Java system details. Section 3.2 uses a small sample implementation to introduce the language extensions used to describe and implement abstractions in our system. The grammar modifications introduced by these language extensions can be found in the appendix.

The grammar and language mechanisms introduced here could lead to a type-safe source-to-source translator that would generate classes with legal Java byte-code. The translated Java classes would use a standard set of Java libraries constructed on top of the standard Java environment to support our model of dynamic component evolution. The next chapter considers various real-world implementation issues involved in designing such a run-time system.

3.1 The Java Platform

Java is a network- and object centric system that can support independent development and deployment of components. It has stated goals of platform independence based on the notion of distributed and mobile computing across heterogeneous platforms. Java has existing mechanisms for separation of abstraction and implementation based on *interfaces* and *classes*. Java's support for subtype polymorphism and dynamic class loading [25] allows applications to extend themselves at run-time.

The Java virtual machine [26] includes an object heap where class instances are stored. Instances have member fields that can hold references to other objects on the heap. New instances are

instantiated by first allocating memory and then invoking a specially named static method ($\langle init \rangle$) included with the class, which then invokes the necessary constructor. The `java.lang.Object` class acts as the top of the type hierarchy and all other classes inherit basic behavior from it. Collection classes or other places where genericity is required use `java.lang.Object` as the reference type. Method dispatching in Java matches our chosen model; each thread has its own stack where method frames are pushed as they are invoked, and then popped as they complete.

In Java, abstract types are created using the `interface` keyword, which allows declarations of method headers. A class can `implement` multiple interfaces by providing method implementations with matching signature for each method declared in the abstract type. Interfaces do not include declarations of constructors since instantiation is bound to the actual implementation class. The method signature can be used to specify:

- the number and reference types of arguments,
- the type of returned value,
- whether the method invocation is synchronized on an object-specific lock,
- checked exceptions thrown by the method.

Some Java features strip away all language-based abstraction and are not supported by our system. For example, the Java Reflection API allows the interrogation of an instance for various implementation-specific details. The Reflection API allows clients of a class to peek behind the abstraction layer intended to hide implementation. Java-language abstractions also do not extend well into the JNI world, where native code does not necessarily conform to behavior specified in the Java environment.

3.1.1 Component Type vs. Java Types

There are several differences between the interfaces in the component mold and existing Java type constructs. The internal interface does not have a matching Java construct, and it does not have a run-time Java type. The most prominent functional difference between the external interface (which defines the component type) and Java interface types is that component objects can be instantiated directly via declared constructors on the external interface, despite the lack of a concrete binding to an implementation. On the opposite end of the spectrum, the external interface and Java class differ in that the external interface is only loosely bound to any particular class implementation.

Static methods and fields can not be declared in the external interface. In Java, static class methods and field declarations represent shared state and behavior for all instances of the type. If the component mold allowed the same declarations, it would suggest all component objects also must share state and behavior in some way. However, multiple primary classes providing implementations and fields might be available simultaneously, making the semantics of access to static fields or methods difficult to determine.

However, the component type is similar to the other Java type constructs in most other ways. Once created, the component object's observable behavior is similar to any other object instance in Java. For example, a component object has a single unique location on the object heap that can be

referred to by other objects. The component type implicitly sub-types `java.lang.Object`, which allows component object to be usable within Java collection classes. Methods such as `hashCode()` is defined to return a consistent value for an object instance, and a component mold instance follows the same rules regardless of evolution. The methods inherited from `java.lang.Object` have no associated reconfiguration points.

Method invocation behaves in identical ways whether invoked on a component object or standard object instance. Methods declared in the component type and shown to clients are identical in form to standard method signatures. Invocations are synchronous and accept arguments, throw exceptions, and return values just as instance methods on other Java objects.

3.1.2 Component Architecture in Java

Friendly classes are declared and implemented as Java classes. As described in the previous chapter, instances of these classes can not have lifetimes that out-last their containing underlying objects. Thus, Java inner classes could be used to implement friendly classes. There are strong similarities between the two structures. The Java inner class is intimately tied to the outer class implementation. Inner class instances can only be created after the enclosing top-level class instance has been created, and can not out-live their enclosing instance. Using Java inner classes for this purpose emphasizes to developers the limitations placed on their use of friendly classes, as well as the significant advantages (namely, the direct access to private implementation and instance state of the outer class). Similarly, standard classes provided as part of the component package are declared as static inner classes. Static inner classes have no dependencies on instances of the containing class, just as standard instances do not.

In addition, since the type-name of the inner class is automatically prefixed with the primary class name, other classes will not be able to access either friendly or standard class definitions (for example, to create an instance of such a class) unless they explicitly name the primary class directly. If we obfuscate the primary class name, this seems like an elegant mechanism for isolating the types provided within a component package.

These existing Java language mechanisms are insufficient for meeting all of our requirements for distinguishing between the different kinds of classes and instances in an application. For example, *internal instances* might be instances of standard classes defined outside of the scope of the primary class; additional analysis techniques would be needed to do any kind of reference flow restriction. Regardless, the properties these mechanisms do support can still represent helpful guides in providing component implementations, especially to developers already familiar with the Java programming language.

We consider Java classes available as part of the Java class library standard for the purposes of this work; their implementation is guaranteed to not evolve for the lifetime of an application. A truly aggressive implementation of this system could also hide such classes behind component molds to allow reconfiguration of the entire system.

3.2 Component Extensions

We introduce a new first-class type construct to the Java system for representing component molds with the keyword `component`. The component declaration declares both the external and internal interface, and thus provides all of the abstractions needed within our framework for transparent use and evolution.

Although these interfaces are declared within the same construct, elements of the internal interface are not part of the component type, and are therefore unavailable to clients of the component type. The component type is used by the standard Java compiler for assuring type-safety between clients and the component. Legal definition and use of the internal interface abstractions would have to be checked by a special pre-processor that understands abstract rep and reconfiguration points.

In this section, we present a simple component mold declaration and two primary class implementations of a basic network stack component written using our extensions to the Java programming language. This example helps introduce and motivate various language and system features.

The language extensions we provide does not include syntax used to declare whether classes are standard or friendly. This work has not focused on evaluating whether typing rules might be used to make this distinction clear, or if a more complicated run-time mechanism would be required. For now, the distinction between standard and friendly classes, as well as standard and internal instances, should be considered guidelines for safety implementing a component.

3.2.1 Sample Component

The simple component introduced here has a single method for transmitting byte-array buffers to different hosts. We do not constrain the component's send methods to complete before transmission is complete. We specify (arbitrarily) that the abstract rep of this network stack should include all pending buffers to be sent, the host they are destined for, and the progress that has been made in each transmission.

Although this description of the component type, behavior, and abstract state is very informal, it is representative of the level of specification often used for deployed software components (such as the Java class libraries). In future work, it might be preferable to describe more precise invariants on the operational behavior and instance state of the component since that would theoretically allows tighter and more correct integration between components.

3.2.2 Abstract and concrete representation

Figure 3-1 shows the abstract rep declaration in the component mold for the component `SimpleNetworkStack`. This code declares the abstract representation for this component type: a vector `currentConnections` containing objects of the nested Java class-type `CurrentPendingSends`.

The abstract rep declared here is part of the internal interface, and should not be confused with concrete field declarations made using similar syntax in Java classes. These fields are not

```
component SimpleNetworkStack
{   ...
   // this vector contains objects of type CurrentPendingSends
   java.util.Vector currentConnections;  // the abstract rep

   class CurrentPendingSends {
     byte[] buffer;
     String host;
     int bufferOffset;
   }
}
```

Figure 3-1: Declaration of component abstract rep.

```
class NetworkStackImplA fulfills SimpleNetworkStack
{
  // Implementation state.  maps String to String.
  java.util.Hashtable privateOpenSends;
  ...
}
```

Figure 3-2: Declaration for `NetworkStackImplA`.

part of the component type, and can not be accessed directly. Abstract fields can *not* be accessed or used during normal computation, and for all intents and purposes do not exist until evolution actually occurs. Therefore, the abstract rep declaration does not severely constrain component implementation representations, since the underlying implementation is not required to maintain consistency for these abstract fields until evolution. To satisfy the semantics of the internal interface, all that is required is that the implementing primary class must be able to present the unsent buffers and the host they're destined for at evolution.

The standard class `CurrentPendingSends` is used to capture the abstract state of a pending buffer transmission, which we declare to include the buffer to be transmitted, the destination host, and the current offset within that buffer.

No access modifiers are used for restricting access to the component type; all methods and constructors are implicitly public, final, static, and abstract. The type declaration can be placed in a package, so that the fully qualified type name would be `package-name.component-name`.

Figure 3-2 shows the declaration of the primary class `NetworkStackImplA`. The primary class is declared just as other Java classes, with the additional use of a special `fulfills` keyword to indicate the meets relationship between the class and the component mold. This keyword specifies that this class is intended to be used as the primary class for the named component declaration; for this to be a valid implementation, this class must now provide appropriate method implementations based on the method declarations in the component mold.

The declaration of the `NetworkStackImplA` network stack in Figure 3-2 shows the flexibility primary classes have in object representation by choosing to use a hashtable to map host names to the remaining buffer to be sent. The buffer representation used by this class is a String class, rather than the byte array specified in the component mold's internal interface. This representation should be sufficient in tracking pending buffer transmissions as required by the abstract rep.

```
class NetworkStackImplA fulfills SimpleNetworkStack {
  ...

  encode {

    Enumeration keys = this.privateOpenSends.keys();
    that.currentConnections = new java.util.Vector();

    while (keys && keys.hasElements())
      {
        String currentHost = (String)keys.nextElement();
        CurrentPendingSends newSend = new CurrentPendingSends();

        newSend.host = currentHost;
        String currentBuffer = (String)this.privateOpenSends.get(currentHost);
        newSend.buffer = currentBuffer.getBytes();
        newSend.bufferOffset = 0;

        that.currentConnections.addElement(newSend);
      }
  }
```

Figure 3-3: Encode clause for `NetworkStackImplA`.

3.2.3 Encode and decode clauses

The evolution process for abstract state depends on programmer-provided transform functions to map implementation representation to the abstract rep declared in the component mold. During evolution, space is allocated on the heap to temporarily hold this abstract state representation. As in standard Java, the special variable `this` is used to hold a reference to the local underlying object; `this` can then be used to refer to the concrete object representation. We use the special variable `that` to refer to the component object, and correspondingly it is used for accessing fields in the abstract rep.

The programmer implements an *encode clause* and a *decode clause* to transform the state between `this` and `that`. The fields held in the abstract representation referred to by `that` are only available within these encode and decode clauses. Within the encode and decode clauses, `that` has a type consisting of the abstract rep declared in the internal interface.

The syntax and form of these clauses are similar to the static and instance initializers used in Java. Any invariants on the abstract state specified by the component declaration must hold when the encode clause is executed; this allows the decode clause to assume that all rep invariants hold before restoring state into the local concrete representation.

The encode clause shown in Figure 3-3 contains a straightforward transformation from the hashtable representation held in `this.privateOpenSends` to the vector reference in `that.currentConnections`. The state that had been kept in each individual hashtable entry is now mapped into an instance of `CurrentPendingSends`. This implementation automatically adjusts the buffer in the hashtable to only contain the remaining string to be sent; as a result, the buffer offset in the abstract rep is 0.

The objects referred to in `that` is passed on for decode and use by the new implementation.

```
decode {
  this();

  Enumeration e = that.currentConnections.Elements();

  while (e != NULL && e.hasElements())
    {
      CurrentPendingSends currentSend = (CurrentPendingSends) e.nextElement();
      String currentBuffer = new String(currentSend.buffer, currentSend.bufferOffset,
                                currentSend.buffer.length - currentSend.bufferOffset);

      // store in hashtable.
      this.privateOpenSends.put(currentSend.host, currentSend.currentBuffer);
    }
  }
```

Figure 3-4: Decode clause for `NetworkStackImplA`.

Thus, these objects must be standard instances that will survive evolution. All of the classes included in the encoded representation: `java.lang.Vector`, `CurrentPendingSends`, and the underlying `String` and byte-array representations of the buffer are standard classes since they are not implementation bound to `NetworkStackImplA` in any way. This allows instances of these classes to be standard. Special care also must be taken during any method invocations made from these clauses that the object they are made on are *fully internal*. These considerations were discussed further in Section 2.3.

Since the decode clause acts as an alternative route by which instances of a class can be initialized, they also allow the constructor-chaining and initialization rules specified for Java. The component developer can specify the particular constructor and arguments to be used at the beginning of the decode clause. The decode clause shown in Figure 3-4 specifies `this()`, the no-argument constructor, as the constructor initializer that should be used when this instance is created. The decode clause then decodes the `that.currentConnections` vector into the implementation representation based on `Strings`, storing the resulting host/buffer pairs into the hashtable `this.privateOpenSends`.

3.2.4 Reconfigurables

Now we introduce our support for the reconfiguration point abstraction. Although reconfiguration points are declared as part of the method header (see Figure 3-5), they are part of the internal interface, and not in the method signature exposed to clients.

At every reconfiguration point, the programmer is asserting that the instance state is consistent and ready for abstract state encoding. The control flow at every such point can be imagined as a predicated exit point into a different method implementation, and all necessary clean-up and instance state repair must be complete. Correspondingly, every reconfiguration point must also be considered as a potential method entry point. The method must be prepared after a reconfigurable statement for the possibility that all local variables are again uninitialized, and must regain their values from a combination of the method parameters and instance state. More complete rules for determining definite assignment are given in the appendix as well.

```
component SimpleNetworkStack
{
  // PartialSentBuffer corresponds to partial transmission.
   void sendBuffer(byte[] buffer, int host)
                            throws java.io.IOException
                            reconfigurables PartialSentBuffer;
   ...
}
```

Figure 3-5: Component method declaration, w/ reconfiguration point.

In Figure 3-5, the method `sendBuffer` is declared in the component mold with a reconfigurable label `PartialSentBuffer`. Semantically, this label corresponds to an abstract point in the execution of the method where some part of the buffer has been sent. The abstract rep declared in the component mold would be used to hold the remaining buffer to be sent upon evolution. An implementation could choose to generate the abstract state in `SimpleNetworkStack` by changing the buffer to only hold remaining bytes, or by using the `bufferOffset` field.

3.2.5 Evolving from this primary class

The implementation of the method `sendBuffer` is shown in Figure 3-6. The implementation is straightforward; when a thread invokes this method, this implementation converts the `byte[]` argument `buffer` into a `String` class format and stores it in the local hashtable under the key of the host-name to be transmitted to. If the call to `internalSendStuff` only completes partially, then the `String` to be transmitted in the hashtable is adjusted to reflect the remaining buffer. This, along with the `encode` representation shown before, preserves the local rep invariant that `this.privateOpenSends` holds all pending buffers to be transmitted.

The reconfigurable point `PartialSentBuffer` indicates to the system that the instance state is consistent from the perspective of this thread context, and the `encode` clause would behave correctly if executed at this point. Note that at this point local variables like `sentCount` and `bufferSent` represent transient state that is now stored within the concrete rep. Upon evolution, the value of these variables could be lost, but the generated abstract rep would have equivalent state in the `bufferOffset` field that captured these values.

When the executing thread reached this point in the method, a pending reconfiguration event could cause the thread to block until all other conditions for instance reconfiguration have been met: namely, the instance state becoming entirely mappable because all on-going computations have reached reconfiguration points. However, the thread would not block when it reaches a reconfiguration point if it were executing a nested call within the current instance. The conditions for determining whether the state in the thread is ready to be evolved are discussed in great detail in the next chapter.

Once the conditions offered above are fully satisfied, all of the current instance's state is mapped over into an instance of the new primary class using the encode/decode clauses. Any threads blocked at this reconfiguration point will leave this method and transparently continue execution in the new instance. The new instance can then transparently replace the original instance in the application.

```
class NetworkStackImplA fulfills SimpleNetworkStack}
{
  ...
    void sendBuffer(byte[] _buffer, String host)
                              throws java.io.IOException
                              reconfigurables PartialSentBuffer
    {
      int sentCount = 0;
      int bufferLength = buffer.length;
      String buffer = new String(_buffer);

      if (!this.privateOpenSends.containsKey(host))
        return;

      while (sentCount < bufferLength) {
          int bufferSent = internalSendStuff(host, sentCount, host, false);
          sentCount += bufferSent;

          this.privateOpenSends.put(host, new String(buffer, sentCount, bufferLength - sentCount));
          reconfigurable PartialSentBuffer {
                ...
          };
      }
     }
  ...
}
```

Figure 3-6: `sendBuffer()` in `NetworkStackImplA`.

3.2.6 Evolving to this primary class

Now we consider how reconfiguration points are used as abstractions for entry points *into* this mapping frame for threads that were executing on a previous implementation of `SimpleNetworkStack`.

As described previously, all of the per-instance state was encoded into an abstract representation in `that`. This representation is then made available and decoded by the current instance to generate its own object representation. The abstract representation is not modified during decode.

The per-invocation state is mapped over into the new instance by placing a new method frame on the top of the thread stack for each thread blocked at a reconfiguration point in the previous instance, removing the old frame in the process. This new method frame is generated by mapping all of the relevant data from the previous one: method arguments and the return address are copied directly, while the program counter is replaced with the appropriate instruction in the new method implementation that matches the reconfiguration point in the replaced method implementation. Note that this does mean any modifications the previous method implementation made to the method arguments are also transferred over. All of the threads blocked for evolution can now continue consistent execution using the new method frame.

Instance invariants

The primary class might be implemented with specific instance invariants. By invariants we do not necessarily mean formally declared state relationships, but rather that the method and class

implementations assume certain properties of the computation and state. These properties can make implementing methods easier for the common case. Here, the common case is that the current object is created from scratch, and all invariants would be maintained by the method implementations. Programmability and performance of the component implementation as a whole should not be affected by potential reconfiguration.

Before evolution is declared complete and application threads begin executing again, the primary class should be given an opportunity to restore all of these necessary invariants. This is a key premise behind the design of the mechanisms used to decode the abstract rep upon evolution. The `decode` clause, for example, is used to restore any invariants associated with the object state.

Every method frame could also have its own local invariants. For example, method implementations are often most easily implemented using local stack-based variables. These local variables would have correct values if method invocations were made normally. In the method implementation in Figure 3-6, the variable `sentCount` is initialized upon invocation of the method, and then consistently updated as bytes are transmitted.

However, if a thread enters this method at a reconfiguration point in the middle of the implementation due to evolution, these local variables are no longer correct. We suggested in section 2.4.2 that providing a state abstraction for every method's local variables would be an unacceptable burden; instead, local variables were placed into the concrete rep and encoded along with the rest of the per-instance state. Therefore, as an optimization for easier implementation, we would like to restore the appropriate values for local state-based variables in the new method frame upon evolution.

To solve this problem, we introduce the *reconfigurable clause*. The reconfigurable clause is a generalized mechanism where a section of code provided in the primary class implementation is defined at a reconfiguration point, to be executed only when that point acts as an entry point to the method. Reconfigurable clauses can be used to adjust the state in the local method frame. After the decode transform function has first been executed to restore the per-instance state, and before the threads are re-activated in the current instance, the reconfigurable clause will be run for every thread mapped into the new instance at a reconfiguration point. The reconfigurable clauses are executed in non-deterministic order by the single evolution thread.

For our specific purpose of restoring the invariant that local variables in a method implementation have 'correct' values, the reconfigurable clause can determine these 'correct' values from the local object state. Figure 3-7 shows the reconfigurable clause implementation associated with the reconfiguration point `PartialSentBuffer` in the primary class `NetworkStackImplA`. In the reconfigurable clause, arguments to the component method are mapped directly from the previous method frame. Thus, at the beginning of this clause, the only variables that have initialized values are the method arguments and local rep already created in the decode clause. The `host` is used as a key to retrieve other parts of the local state (the `buffer` in this case) from the object state in the decode clause. The rest of the local variables are also in scope here and have allocated space within the method frame; the reconfigurable clause assigns values to those local variables.

In addition, there might also be object-wide invariants related to thread state that reconfigurable clauses alone can not restore. For example, component implementations might specify that a certain amount of local storage proportional to the number of threads currently active should be allocated; the reconfigurable and decode clauses are not capable of restoring this invariant.

```
reconfigurable PartialSentBuffer
  {
    if (!this.privateOpenSends.containsKey(host))
      return;

    buffer = this.privateOpenSends.get(host);
    sentCount = 0;
    bufferLength = buffer.Length();
  };
```

Figure 3-7: Reconfigurable clause for `NetworkStackImplA`.

For this purpose, we introduce the *finally clause*. The finally clause can be used in combination with the reconfigurable clauses as a mechanism for a primary class to track the threads mapped over from the previous instance, and then adjust state and behavior as needed to restore instance invariants.

The finally clause is executed as the last step in the evolution process after all reconfigurable clauses have completed execution. As each reconfigurable clause is executed, it can record in the local concrete rep the nature of the mapped method and thread. When the finally clause executes, it will be able to examine this concrete representation, including the thread state, and adjust implementation behavior appropriately. For example, the primary class could use the reconfigurable clause to increment a counter used to track the number of threads mapped over from the previous instance. The finally clause can then use this number to allocate appropriate storage.

The finally clause is used in our implementation of `NetworkStackImplA`. There is an implicit object-wide invariant in this class that a separate thread exists for processing each unsent buffer. This invariant follows naturally from the assumption made by the creator of `NetworkStackImplA` that the method `sendBuffer()` does not return until the entire buffer passed in as arguments in the call is completely transmitted. Based on this invariant, `NetworkStackImplA` does not normally need to provide any other mechanisms to transmit buffers. For all new method invocations to `sendBuffer()` on this class, the implicit invariant will hold.

This invariant may be violated if the current instance is instead created through evolution. If the prior instance did not hold to the same invariant (and it does not need to), then there might not be a thread for each pending buffer in the object. To restore this invariant, the object needs to track each of the incoming threads mapped from the previous instance, and then create additional threads as needed to invoke `sendBuffer()` for the unprocessed buffers.

First, we add a new field to the local representation that allows us to track mapped threads. This field is initialized in the decode clause:

```
class NetworkStackImplA fulfills SimpleNetworkStack {
    private java.util.Vector currentlySending;

    decode {
       this.currentlySending = new Vector();
    ...
    }
}
```

The reconfigurable clause is modified slightly to help us record the buffers that mapped threads

```
decode {
 ...

 finally {
  Enumeration keys = this.privateOpenSends.keys();
  if (keys.size() == 0)
   return;

  while(e.hasMoreElements()) {
   String currentHost = (String)keys.nextElement();

   if (!this.currentlySending.contains(currentHost)) {
      Component.createThread(that, sendBufferMethod,
         new Object[] {this.privateOpenSends.get(currentHost).getBytes(),
                                          currentHost} );
   }
  }
}
```

Figure 3-8: Finally clause for `NetworkStackImplA`.

will process once evolution complets:

```
reconfigurable PartialSentBuffer {
     currentlySending.addElement(host);
     ...
};
```

When evolution occurs *into* this class, the decode clause is first processed, creating the vector `currentlySending`. After the concrete rep has been initialized by the decode clause, the reconfigurable clause for `PartialSentBuffer` is executed for every thread mapped onto this new instance. As a result, the vector `currentlySending` will hold the names of all of the hosts that current threads will be processing once all of the reconfiguration clauses have been processed; this vector can be empty if no threads were mapped over from the previous instance.

We present a finally clause in Figure 3-8 that uses this information to create additional threads to handle unsent buffers that would otherwise be neglected. This will restore the expected invariant that all threads are handled by a thread invoking the method `sendBuffer()`. The finally clause is placed at the end of the decode clause, but it is executed only after all of the reconfiguration clauses have executed.

The finally clause checks all of the host-names held within the local instance state against the host-names that are already being processed by a thread, which are stored in `this.currentlySending`. For each host that is discovered to be unhandled, the finally clause makes an invocation to a special library call that will create a thread to invoke the method `sendBuffer()`. We only informally introduce the static library call `Component.createThread()` here.

```
void createThread(java.lang.Object, java.lang.reflect.Method, Object[]);
```

This call creates a new user-thread that then invokes the passed-in method (in the form of a `java.lang.reflect.Method` object) with arguments on the given object instance. Since the created thread invokes a component method through a component object, the first method frame pushed on to the stack has only abstract references that allows it to be mapped onto future underlying objects

```
class NetworkStackImplB extends java.lang.Thread fulfills SimpleNetworkStack}
{
  // Implementation state.
  private class PrivatePendingSendJob
  {
    byte[] buffer;
    String host;
    Integer priority;
  }

  util.Queue privatePendingQueue;

   NetworkStackImplA() {
     ... basic initialization ...
     privatePendingQueue = new util.Queue();
     this.start(); // start a thread running in this class.
   }

   public void run() {
      while(true) {
          if (!privatePendingQueue.empty())
                ... process the job in the queue, pop off queue ...
          wait(500);
      }
   }
  ...
}
```

Figure 3-9: Declaration for `NetworkStackImplB`.

in case evolution occurs before the method invocation completes. By relying on the abstraction layer, these created threads can survive future evolutions. We call such threads *external threads*, since they are effectively created outside of the current primary instance.

3.2.7 Alternative implementation

In Figure 3-9, we introduce another implementation of the network stack, `NetworkStackImplB`. This primary class differs from NetworkStackImplA in representation and behavior, and yet still satisfies the component mold. This implementation uses a simple (synchronized) queue to order pending transmissions. The queue is processed by a single thread which processes buffers pushed on-to the queue one by one. This implementation might be used to replace the previous network stack if multiplexing multiple out-going transmissions through the physical layer connection is no longer desirable (or possible).

As shown in Figure 3-9, this class relies upon a class `PrivatePendingSendJob` as part of its internal implementation. Although this class does not directly refer to the primary class, its behavior is tightly integrated with the implementation in this class, and it is being used as part of the implementation rep of this object. Therefore, `PrivatePendingSendJob` is a friendly class, and correspondingly defined as an internal class.

`NetworkStackImplB` also extends the Java library class `java.lang.Thread`, which means this class can start a new thread through the `run()` method. This thread is used within the primary

```
public void run() {
   while(true) {
       if (!privatePendingQueue.empty())
             ... process the job in the queue, pop off queue ...

       reconfigurable; // internal thread reconfigurable point.

       wait(500);
   }
}
```

Figure 3-10: Modified `run()` method in `NetworkStackImplB`.

class to process jobs placed into the queue. The constructor implementation shown in Figure 3-9 creates an instance of the queue, and then causes a new thread to be forked by invoking the method `start()`, which is inherited from `java.lang.Thread`. The new thread begins executing in the method `run()`.

However, the component mold declaration does not specify the existence of a thread that behaves this way. Since the thread only executes a primary-specific method `run()`, there is no consistent way to map this thread to another underlying object. This thread is not invoked through the component object on a component method like the use of external threads described before, and the lack of this shared abstraction makes it impossible for this thread to behave consistently if the underlying instance was evolved away.

We call these threads that can not be, and are not intended to be, preserved across the evolution transformation *internal threads*. Internal threads can be created in a number of ways. In the case of Java, internal threads can be created using internal instances that extend the `java.lang.Thread` class, as in this implementation. Alternatively, an internal thread can be created using the same `createThread()` method used previously to create external threads. Except now the call specifies that the method is to be invoked directly on the underlying primary class instance, rather than the component object:

```
Component.createThread(this, runMethod, new Object[] { });
```

The resulting thread will now have a method frame on its stack that is primary-specific; the frame directly refers to an object and method that are neither abstract nor mappable to future underlying objects. One way of contrasting the semantics of internal versus external threads is to consider external threads as being created for an application-wide computation. In contrast, an internal thread is only used for some computation that originated, and is only of interest to, this particular primary class implementation.

Internal threads still play a role in determining when evolution can occur. Even if we can not preserve on-going computation in this thread, we still can't evolve the instance if the internal thread is currently in the process of modifying local heap state. As with external threads, we can simply wait until this thread completes its invocation and removes the method frame from its stack before evolving the instance. For more timely evolution, we can instead provide unlabeled reconfiguration points exclusively for the purpose of indicating heap-state consistency. The run method must be modified accordingly; the modified version is shown in Figure 3-10. More details about internal threads are described in section 4.5.

The `sendBuffer()` method show in Figure 3-11 in this second implementation offers an inter-

```
class NetworkStackImplB fulfills SimpleNetworkStack
{
  ...
  void sendBuffer(byte[] _buffer, String host)
                                        throws java.io.IOException
                                        reconfigurables PartialSentBuffer
    {
      PrivatePendingSendJob sendJob = new PrivatePendingSendJob();
      sendJob.buffer = new byte[_buffer.length];
      java.lang.System.arraycopy(sendJob.buffer, 0, _buffer, 0, _buffer.length);
      sendJob.host = host;
      this.privatePendingQueue.push(sendJob);

      reconfigurable PartialSentBuffer {
         return;
      };
      ...
    }
  ...
}
```

Figure 3-11: `sendBuffer()` in queue-based `NetworkStackImplB`.

esting contrast to the other method implementation defined in `NetworkStackImplA`, as shown in Figure 3-6.

In this implementation, the method pushes an instance of `PrivatePendingSendJob` representing the buffer to be transmitted on to the queue rather than processing it directly. The method invocation is short-lived, and the actual buffer transmission occurs asynchronously from the perspective of the client. The bulk of the work is carried out by the internal thread started earlier that moves through the queue sequentially. Evolving *from* an instance of this primary class, even without the use of reconfiguration points, would be timely since the method `sendBuffer` returns so quickly.

The reconfiguration point `PartialSentBuffer` must still be defined to allow mapping of the program counter from and to previous implementations. If the reconfiguration point is used as an exit point, the instance state already meets the rep invariant, and evolution is straightforward.

The implementation in Figure 3-11 also shows that the reconfigurable clause can be used for more purposes than just restoring local stack variables. Here, the reconfigurable clause contains just a return statement in case this reconfiguration point is used as an entry point from previous instances. The `decode` clause for this class has already restored all pending transmissions and stored them in the queue for processing; the external thread brought into this context has no other work and should return immediately. This matches the implicit invariant of the current implementation, namely that buffer transmissions occur asynchronously through the internal thread.

This puts into sharp relief the interplay of thread semantics between instances of these two primary class implementations. When we evolve a component object to use an instance of the hashtable based `NetworkStackImplA`, we create external threads as needed to transmit buffers in the object state. When the instance is then evolved into the queue based `NetworkStackImplB`, all of the created external threads are now destroyed (by returning from the only method they have invoked), and instead a single internal thread created in their place to manage transmission. If another evolution back to the `NetworkStackImplA` implementation occurs, the internal thread is

now discarded and external threads created anew.

3.3 Summary

In this chapter, we presented the core of our Java-based language support for declaring and implementing evolvable components. Using a motivating example, we introduced the detailed mechanisms required for implementing a reconfigurable component.

Chapter 4

Application Reconfiguration

An application consists of multiple interacting objects, including component objects capable of evolution. The challenge is in transparently evolving the underlying implementation of existing component objects to use the new primary class implementation.

When an application is first executed, a single system thread is created and begins to run at some well-defined static point within core application classes. This thread can invoke methods through component objects.

Transparent evolution means that if the primary class is correctly implemented, a thread invoking a method on a component object will complete the method invocation correctly regardless of whether evolution occurs. The thread will return to the client outside of the component object that made the method invocation after the specified abstract behavior of the method has completed. Any state-changing side effects of the method invocation should be registered in the current concrete representation. In section 4.1, we discuss further the principles behind the preservation of transparency during evolution.

By expanding the scope of our analysis to the application level and beyond simple state mappability, we can extend our rules for quiescence to realistic implementations. In addition, this approach allows us to explore optimized conditions for quiescence and thread passivity without preserving all instance state.

In section 4.2, we introduce our application model. Section 4.3 discusses how reconfiguration occurs within our application model by first focusing on evolving an instance in a single-threaded environment. Section 4.4 looks at optimizing our conditions for quiescence. Finally, we consider the additional complexity added when concurrency is allowed in our system. In addition to holding on-going computation state, multiple threads introduce additional complexity because they can deadlock the application with held synchronization locks or parallel instance evolution. Section 4.5 describes how we can achieve quiescence without deadlock.

Finally, section 4.6 explains how our component model can leverage the typing rules in Java in a real implementation of this design.

4.1 Component Quiescence Principles

The term *quiescence* is used by Kramer and Magee in [24] to indicate the state of an instance when evolution is possible; we define the term similarly. In this section, we summarize the principles that guide when quiescence is actually possible.

An object becomes quiescent when we can satisfy our goal of evolving that instance to a different implementation while preserving transparency from the perspective of *all* clients. Transparency requires that the two instances are equivalent with respect to the invariants specified within the component mold; only when this occurs should the instance be considered quiescent. There are two general sets of clients where transparency is important: client objects and application threads.

- For client objects that hold references to the component object, behavioral transparency is preserved if the state and methods of the new underlying object have equivalent meaning and behavior with the previous instance's state and methods. Since methods are specified in the component interface, we assume both method implementations meeting this specification would behave similarly if the object state remains equivalent and consistent.

  Therefore, abstraction functions applied to the concrete state in both the new primary instance and the evolved primary instance must map to equivalent abstract component object state to achieve transparency. This constraint is satisfied (assuming the correctness of both primary classes) by creating the concrete state of the new underlying object directly from the abstract representation generated from the previous underlying object. The condition for quiescence here is that the state of the primary instance is consistent and mappable to the abstract representation.

- Threads represent another set of clients for the primary class. Threads execute by accessing the underlying object with a series of operations: the thread requests the instruction at a specific program counter value, and then modifies its local thread state (by changing the program counter, or by pushing on a new method frame), as well as the instance state accordingly. The thread's response to the instructions and state returned by the object instance are well-specified by the system's execution model, and this behavior should be independent of evolutionary changes in the underlying object.

  The challenge with preserving transparency for threads is that the threads are dealing intimately with implementation-specific state. The program counter, instruction sequences, further invocations of local (non-component) methods, and direct accesses to fields are all specific to the current implementation. For transparency to be possible, we need to locate points in the thread's sequence of operations where it is exclusively accessing the underlying object through shared abstract state, which can then be mapped onto a different underlying object.

  We use the term *passive* to indicate the state of a thread when it is prepared for the evolution of a particular instance. A thread is passive when its instance-specific state is in an abstract and mappable form. As the previous chapters suggested, reconfiguration points play a major role in this process. The next sections explore the precise conditions and mechanisms that allow a thread to become passive. Once the thread becomes passive, it will block and wait for the instance to evolve before becoming reactivated and continuing execution in the new underlying object.

4.2 Application Model

In this section, we define a simple model for the state of the application related to reconfiguration. The application state has two primary elements: a heap (set) where objects are held, and a set of current threads. This can be represented by a pair of sets containing these elements:

$$\langle \{O_1 ... O_n\}, \{T_1 ... T_n\} \rangle$$

O_1 is the root object where the primary thread began execution of the application.

Each object consists of a set of fields with references to other application objects, and a reference to a class containing methods with operations on those fields. Objects on the application heap can also be component objects. The component object on the heap can be seen an abstraction function (which functionally might be implemented by an indirection table or surrogate) that operates on an underlying object (another application object) to implement an application object ($O_i = AF_i(O_j)$). Each application object O_i that is a component object is equivalent (although not identical) to $AF_i(O_j)$ for some j.

Both $AF_i(O_j)$ and O_j can be referred to by other objects, but they have different identity on the object heap. As discussed previously in Section 2.3.2, in a correct program only other internal instance can hold direct references to the underlying object O_j. Although method invocations on either instance would appear to be executed identically from the perspective of the client, there is a distinction since references to one instance become invalid upon evolution.

A thread is specified as being a stack of method frames: $T_i = [M_1, M_2, ..., M_n]$. The method frame M_1 is at the bottom of the stack, while frame M_n is at the top of the stack, and represents the method most recently invoked by this thread. A method invocation within the right-most frame causes a new frame M_{n+1} to be added to the stack for the thread, and when the thread completes a method invocation the frame is popped off.

Each method frame consists of: a reference to the particular application object O_i the method is being invoked on, a reference to a set of application objects that are method parameters P_1 ... P_k, and a program counter value C reflecting the current instruction being evaluated in the method.

$$M_i = \langle O_i, \{P_1 ... P_k\}, C \} \rangle$$

The value C can be a concrete value referring to a particular instruction in the current method implementation, or a reconfiguration label that represents an abstract point in the method computation.

If the method being invoked is a component method, then the method frame references O_i (or equivalently $AF_i(O_j)$), and not the underlying object O_j; we call this type of frame an *abstract method frame.* If a component method is invoked through a reference to an underlying object, it is adjusted to refer to the component object O_i if possible. A method that is not part of the component type can also be invoked directly on the underlying object O_j; we call this a *concrete method frame.*

Objects on the application heap can be removed if not directly or indirectly referred to by the root object of the application. A thread is considered complete when its stack of method

frames becomes empty, and removed from the application state. The application is complete and terminates when the initial application thread completes.

4.3 Reconfigurations

Instance evolution is a transformation on the state of the application that does not change the meaning of the application. The evolution process, operationally, transforms a component object on the object heap. As part of the evolution process, application object $AF_i(O_j)$ is atomically replaced with $AF_i(O_k)$ where object O_k is an instance of the primary class from the new component implementation (again, functionally this might be implemented as changing the value in the indirection table). For correct application semantics, the component object should remain consistent during this change, meaning that $AF_i(O_k)$ and $AF_i(O_j)$ must be equivalent. O_j, and other internal instances that directly or indirectly referred to O_j are (conceptually) removed. The rest of the object heap (which only held references to the component object and not O_j or other internal instances directly) remains unmodified.

The evolution process also transforms all method frames in application threads that references the evolving object. Each of these frames is mapped into a similar frame that can accommodate the use of the new underlying object. Most of the state in the method frame remains unchanged initially: the instance referred to by the invocation remains the component object, and all method parameters remain the same. The program counter, however, is transformed to a matching value in the new method implementation. Of course, the only matching values within different method implementations are the reconfiguration points. Combining this with our early definition of a passive thread as any thread where all of its state can be transformed while preserving equivalent value, we see that this is another way of stating condition $1B$ for state mappability given in Section 2.4.2. The rest of this section focuses on this view of passivity and how it is achieved.

Although the application model supports multiple threads, for now we assume there is only a single external user thread within the application. Reconfiguration in this case only requires the single thread to be passive, allowing us to focus on the conditions needed to make this possible. Multiple threads in the application would introduce additional complexity in managing quiescence since they must all be simultaneously passive. For now, we also assume a single candidate component object $AF_i(O_j)$ is selected by the run-time system to be evolved. Section 4.5 will extend our results to an application with multiple concurrent threads.

4.3.1 Thread outside instance

If a reconfiguration event arrives while the thread is not executing within this object, the object heap state can be directly converted using the encode clause into the abstract rep. The only clients of the component object are other application objects that hold references to $AF_i(O_j)$, and this state transformation will provide transparency to those clients. The application thread does not contain any method frames that hold references to either O_j or the component object; thus, the thread is trivially passive toward the object since it contains no state that needs to be transformed upon evolution.

The run-time determines the thread is passive by scanning its method frame for references to either the component object or the underlying object. Once the thread is determined to be passive, the run-time system creates a separate system thread to handle the evolution process. We call this the *evolution thread.* The evolution thread first executes the encode clause. The evolution thread then continues on to decode the representation by executing the decode clause, including the finally clause.

The reconfigurable clauses are not executed in this case since there are no threads mapped into this object. The code that is executed in the decode, encode, and reconfigurable clauses are not allowed to create additional threads of any kind. Doing so would allow concurrent modifications of internal state that the evolution mechanism is unprepared for, requiring more concurrency mechanisms to determine when evolution can continue. Thus, for simplicity the only thread that can be active in an instance during evolution is the system thread.

4.3.2 Thread inside instance

Now we consider the case where the thread is active within $AF_i(O_j)$, meaning that it contains method frames that referred to the component object. This application thread only becomes passive when these method frames become mappable. When the reconfiguration event occurs, the run-time will again try to determine immediately if the thread is passive by scanning the thread stack and searching for method frames that refer to the component object. If the program counter in any method frame is not currently at a reconfiguration point, then the frame can not be mapped into the new method implementation, and the thread is not passive.

As part of the scan, the run-time system also marks the lowest method frame on the stack (the one earliest in the chain of method invocations) invoked on the component object; this is the frame that is ultimately blocking evolution of the object. Even if other frames higher on the stack became mappable, this frame would still have a program counter that did not point at a reconfiguration point. Only the reconfiguration points within this method frame become *active*, indicating that the application thread could block here.

The application thread continues to execute. Every time the application thread's program counter reaches a reconfiguration point in the current method frame, a predicated check (generated automatically from the reconfiguration point statement in the source code) is run to determine if the current frame is the one that was marked as blocking evolution, and this reconfiguration point is active. The same predicated check is run whenever the thread is about to return from a method frame, where the implicit reconfiguration point `END_METHOD` is placed.

Since the thread has now reached a reconfiguration point, *this* frame would no longer be blocking evolution. If the check fails, another method frame is responsible for preventing this thread from becoming passive, and execution can continue. The thread finally becomes passive when the check succeeds as all frames in the stack have become mappable.

As before, the evolution system thread is created to encode the current instance into the abstract representation. In addition to encoding object state as before, the evolution thread must now handle the mapping of thread state as well. As before, the evolution thread executes the decode clause, initializing object state for the new primary instance.

The thread then modifies the program counter in the top-level method frame to point at the instruction immediately following the matching reconfiguration point in the new implementation. The rest of the frame remains unchanged (although system invariants might require local variables on the method frame to be cleared). The system thread then executes the reconfigurable clause attached to the reconfiguration point. Any references to local stack variables while executing this clause will actually reference (and modify) the variable stored in the application thread method frame. The thread goes on to execute the finally clause defined as part of the decode clause when the reconfigurable clause completes.

A third possible scenario is that the application thread was outside the instance when the reconfiguration event arrived, causing the run-time to begin evolving the object. While the evolution thread was executing, the application thread then invokes a method on the component object. Clearly, this invocation can not be allowed to start since it would violate the condition of quiescence the evolving thread depends upon. Since the beginning of the method has an implicit `BEGIN_METHOD` reconfiguration point, the application thread could remain passive simply by blocking at it. Since this reconfiguration point does not have a reconfigurable clause associated with it, the evolution thread do not need to do any additional work to modify this method frame beyond the simple mapping. The order the evolution thread executes clauses remain unchanged; it only needs to confirm at the end of the process for recently blocked threads, which can then be mapped over in a straightforward manner.

After evolution is finally complete, the thread begins execution using its new method frame.

4.3.3 Summarized conditions for passive thread

A thread T is passive toward a component mold object $O_i = AF_i(O_k)$ if two conditions are met. First, for all of its method frames invoked on O_i, the program counter is currently at a reconfiguration point.

The second condition is new: the thread can not have *any* method frames that refer to the underlying component instance (this includes all internal instances). That is, for all method frames on the stack, none of them can be a concrete method frame invoked directly on O_k, the underlying primary instance; these method frames can not be mapped over into new instances of the primary class. This second condition has yet to arise in our discussion since a method on the enclosing component object should have been invoked before any methods on the internal instances. The enclosing method frame would fail the first condition, making the second condition unnecessary. However, for completeness we include this condition here; it will become relevant for our quiescence optimizations.

For thread $T = [M_1, M_2, ..., M_n]$ and component object $O_i = AF_i(O_k)$, the rule for determining whether T is passive is :

$$\begin{aligned} &\forall M = \langle O, \{P_1 ... P_k\}, C\rangle \in T_i \\ &\quad : (O = O_i \wedge C \text{ is a reconfiguration point}) \vee (O \notin \{O_k, O_i\}) \\ &\quad \Longrightarrow \text{Thread } T_i = [M_{i,1}, M_{i,2}, ..., M_{i,n}] \text{ is } \textit{passive} \text{ toward object } O_i \end{aligned} \tag{4.1}$$

4.4 Optimized quiescence

The scenarios given in the previous section demonstrates that these requirements for a passive thread can be restrictive. In a real application with multiple threads and long method invocation chains, it seems likely these requirements would delay timely reconfiguration.

The major weakness in our system is that only top-level abstract method frames can be mapped between different component instances. The program counter for lower frames refer to a pending method invocation, instead of a reconfiguration point shared between multiple components. For method frames such as these with pending method invocations to be mappable, the method invocation must have also been declared abstractly as part of the component mold.

For example, if the program counter for a method frame lower on the stack is currently pointing at an invocation of the method `foo(integer, string)`, this method frame can not be mapped over to an instance of the new primary class unless that method implementation also similarly invoked `foo(integer, string)`. Only if these two primary classes (and all other implementations of the same mold) could agree on invoking the same methods at this particular point with similar arguments could the frame be preserved. However, declaring such an "outerface" is impractical.

This weakness can be a significant burden on implementation and run-time performance. Essentially, reconfiguration points are only active for a thread (meaning that thread can become passive by blocking at them) if they are placed in the first method frame invoked through the component object. Objects can *only* become quiescent once a thread returns to the top-level method implementation for a mold method.

4.4.1 Tail-calls

For example, imagine an instance of class FooBar that invoked other helper methods on itself (including recursive invocations to the current method) to process a mold method.

```
class FooBar fulfills FooMold {
  void foo(int count) reconfigurables Bar
  {
    if (count < 0)
      return;

    ... some computation ...
    reconfigurable Bar { ... }
    foo(count-1);
   }
  ...
}
```

Consider a thread T that invokes `foo(10)`. The thread T does some computation, and then recursively invokes `foo` again with the counter decremented by 1; the method runs 10 times before the thread returns to the caller. Methods invoked within a primary instance on `this` is automatically converted to a method-call through the component object when possible; therefore, a new component implementation should be just as capable of completing the behavior of `foo` after evo-

lution. The reconfiguration point `Bar` is *intended* to allow the thread T to become passive toward the instance, and allow evolution to a new component without having to complete the entire series of computation.

However, if the reconfiguration event arrives after the thread T has already made the first recursive method invocation `foo(9)`, the component object can not evolve until the entire method has run to completion (with all ten recursive invocations). The first method frame holding a reference to the component object in thread T has a program counter that points to the method invocation of `foo`, and not a reconfiguration point; therefore, the thread is not passive and can not be mapped to new instances.

In this case, the invocation at the end of the method `foo` is actually a *tail-call.* A tail-call is a method invocation that will be the last instruction executed in the current method, and the return value of the invocation (if there is one) is also the return value of the current method. As soon as a tail-call invocation returns, the current method frame does not have any further computation and would return to its caller immediately. Therefore, there is really no need for the thread to retain this method frame within its application state. This is the same compiler optimization used to improve performance on tail-recursive invocations by removing the stack allocation normally necessary for a new method frame. Similar optimizations are used to improve performance in monitors to minimize lock re-acquisition.

When the run-time system is checking for thread passivity by scanning the stack as before, it can safely ignore these tail-call frames where the next instruction to be executed after the current invocation returns is just another immediate return. These frames are effectively "folded" away during checks for thread passivity. In the example above, the thread T becomes passive the next time it reaches the reconfiguration point `Bar` since we can fold away all prior method frames for the method `foo` on the stack.

Note, however, that synchronized methods can *not* be "folded" if the lock it holds on the current object are only released after the method invocation has completed. An exception to this rule exists if the lock it held is shared by the method invoked, in which case it could be released within that method as before.

Depending on the system-specific semantics of the thread stack, this "folded" method frame might be removed upon evolution. Clearly, this frame is no longer valid and its references can not be considered legitimate. For example, if stack introspection is used to determine system security, as in Java, then these frames do indeed need to be removed, or the security system must be modified to ignore tail-call frames as well.

As an optimization, the compiler can determine that a particular invocation in a component method is in fact a tail-call. It can then generate code such that before this invocation occurs, the method frame is marked (using a specialized local variable) to indicate this frame do not need to be preserved in case of evolution. The run-time system can safely ignore all method frames that are marked this way when scanning to determine thread passivity.

```
class FooBar fulfills FooMold {

  Date difference;

  void foo() reconfigurables Bar
  {
    Date currentDate = new Date();

    ... some computation ...
    foo2();
    reconfigurable Bar { ... }

    // Statistics/Clean-up
    Date lastDate = new Date();
    difference = lastDate - currentDate;
    ...
  }

  void foo2() reconfigurables Bar2
  {
    ...
    reconfigurable Bar2 { ... }
    ...
  }
```

Figure 4-1: Sample program with pseudo-tail-call.

4.4.2 Beyond tail-calls

We can do better than just improving pure tail-call invocations. *We are only interested in preserving consistency for the component object, and not the underlying object.* Up until now, we have assumed these two goals were one and the same. However, the primary class could include behavior (and corresponding state) that is primary-specific, and do not affect client-observable behavior of the component object. Thus, we provide mechanisms that allow developers to declare certain parts of the primary class behavior as not needing to be preserved.

Consider another version of the class `FooBar` as shown in Figure 4-1. As before, the method `foo` invokes another method that includes a reconfiguration point. As a result, the presence of the `foo` method frame makes it impossible for the thread to become passive within the `foo2` method frame. Unlike before, the method `foo` has additional computation after the invocation used to track local statistics (in this case, the amount of time the computation consumed) that prevents this call from being recognized by the compiler as a pure tail-call. For total consistency for an instance of class `FooBar`, the state contained within the `Date` variables must also be consistent and complete, meaning that evolution within `foo2` is not possible.

If, however, the computation that follows the invocation can be discarded and still maintain consistency for the abstract instance, this method frame *can* be folded safely. For example, implementation-specific clean-up code can usually be discarded since the current instance would be discarded regardless after evolution. The developer can indicate this status in code:

```
void foo() reconfigurables Bar
{
  ...
  reconfigurable foo2();
  ...
}
```

The `reconfigurable` keyword here is used to indicate that the computations within this method frame have left the local instance state consistent at the point of this *reconfigurable method invocation*, which is essentially a pseudo-tail-call. The developer is stating that for this method frame, the `encode` transform function generating the abstract instance would return identical values regardless of whether the rest of the method is evaluated or not. Now, a thread with method frame `foo` on its stack can also become passive if the pointer counter in that frame is pointing at a reconfigurable method invocation.

These reconfigurable method invocations can be compiled into the same run-time representation as was used for the previous tail-call case. The method invocation is surrounded by instructions that mark a specialized local variable in the method frame, indicating this frame can also be "folded" during run-time checks. We can no longer depend on the frame to return to its caller automatically when the pseudo-tail-call returns. Thus, this frame would have to be removed when the thread was mapped onto a new primary instance, or a series of predicated instructions must be generated to follow the method invocation to return immediately in case of evolution. Alternatively, the return value for the next method frame could be adjusted to point directly to the return value for the previous method frame, allowing the method return instruction to completely skip the current frame at run-time.

Revised conditions for passive thread The relaxed conditions for determining whether a thread is passive allow method frames referring to the component object in the current thread to have program counters at either reconfiguration points, tail-calls, or reconfigurable method invocation. In addition, a thread can now be passive even while containing method frames that reference an *internal* instance if and only if the program counter in that frame points at either a tail-call or reconfigurable method invocation.

4.4.3 Concrete Method Quiescence

The techniques discussed so far for folding method frames allow incomplete method invocations lower on the stack to be removed from consideration, allowing a thread to become passive more quickly. When these frames are folded away, the top-level frame remains, and can be mapped into the new instance as long as it is referring to an abstract method shared in both implementations. However, if the top-level frame is a concrete primary-specific method instead, a mapping during evolution is still not possible.

From a practical point of view, however, quiescence in concrete methods can be desirable. For example, consider the primary class shown in Figure 4-2.

The method `foo` must take some (long-lived) action depending on a passed-in condition. The best design for this component would be to separate the different actions into different helper

```
class FooBar fulfills FooMold {

  void Helper1()
  { ... }

  void Helper2()
  { ... }

  void foo(Bool bCondition) reconfigurables Bar
  {
     if (bCondition)
       Helper1();
     else
       Helper2();
    ... reconfigurable bar ? ...
  }
}
```

Figure 4-2: Primary class implementation, showing difficulty in placing reconfiguration point in top-level method.

methods (or even helper classes), which can then be invoked as needed. If a thread is not allowed to be passive in a concrete method, then `Bar` would have to be placed within the abstract mold method `foo(Bool)`.

```
void foo(Bool bCondition) reconfigurables Bar
{
    if (bCondition)
    {
       Helper1Part1();
       reconfigurable Bar { ... }
       Helper1Part2();
    }
    else
    {
       Helper2Part1();
       reconfigurable Bar { ... }
       Helper2Part2();
    }
}
```

This is undesirable since the best place for the reconfiguration point `Bar` is actually within the concrete helper methods, where it could be integrated with the rest of the behavior of the method. A reconfiguration point that logically belongs inside a loop, for example, would be difficult to capture if we had to move it up to the closest abstract method implementation.

The rules for thread passivity can be extended to support concrete method frames. We allow threads to block at reconfiguration points within a method frame as a proxy for points declared for previous calling method frames. This technique can apply to both concrete and abstract method frames. A method implementation can now include labeled reconfiguration points that are not actually declared within the method header; these are *proxy reconfiguration points*. Proxy points act as extensions of reconfiguration points defined in earlier method frames.

A set of contiguous fold-able method frames can now extend their declared reconfiguration points

to be satisfied by the proxy reconfiguration point, which is defined in a method higher up on the stack invoked by these frames. In a sense, the method frame with the proxy reconfiguration point is being "collapsed" down the method stack to meet the method frame where the reconfiguration point is actually declared. Threads becoming passive at a proxy reconfiguration point is mapped into an equivalent state where the thread is becoming passive at the underlying true reconfiguration point. The method frame generated for the new primary instance will be based on a frame that was not actually at the top of the thread stack when the thread became passive.

This is different from the behavior of the tail-call optimization, which only allowed method frames in the middle of the stack to be folded *up* toward the top-level method frame. Proxy reconfiguration points, in contrast, are used to collapse the top-level method frame (which can not be evolved) down toward a method frame that can.

In this case, a thread T can also be considered passive if:

- T reaches a reconfiguration point L inside a method (either concrete or abstract),
- some set S of previous method frames for this thread can all be folded (based on any of the mechanisms described previously),
- *and* the label L is declared in one of the abstract method frames in the set S.

Proxy reconfiguration points are not defined or compiled any differently. However, at run-time, the check for thread passivity at a reconfiguration point is now modified slightly. The system will check if the current reconfiguration point is declared as part of the current method frame. If so, this is a true reconfiguration point, and evolution occurs as before.

If not, then the system begins to scan backwards on the thread stack as it collapses this top method frame away. First, it checks if the next frame on the stack is marked fold-able. If not, then the computation within that frame is not complete and it can not be collapsed away. Thus, the thread is not passive. If this frame is fold-able, then the system checks if the proxy reconfiguration point might be an extension of a declared reconfiguration point in the current frame. The system checks if this frame refers to a component method invoked on the current object, and if so, whether the reconfiguration point is declared in the method header. If a matching reconfiguration point is not found in this frame, the system continues to scan down the stack until a method frame that is not fold-able is found, in which case the thread is not passive.

If these conditions are all satisfied, the thread is declared passive. Instead of blocking at the top level reconfiguration point, the thread is semantically blocked at the lower abstract method frame where the matching reconfiguration point was actually found. Note that this means method frames that referred to component methods in other objects, if declared foldable, can also be removed from the system. Upon evolution, the new method frame where the thread continues execution is generated to match this lower method frame, with the program counter set to the matching reconfiguration point.

For example, consider the same method `foo(Bool)` as before; a more efficient definition is shown in Figure 4-3.

`foo2()` is another abstract method also declared on the component mold interface. When a thread T invokes `foo(Bool)` on the component object $O_i = AF_i(j)$ as before, there are now a

```
void foo(Bool bCondition) reconfigurables Bar
{
  ...
   if (bCondition)
     Helper1();
   else
     foo2();
}

void foo2() reconfigurables Bar2
{
   ...
   reconfigurable Bar2 { ... };
   ...
   reconfigurable Bar { ... };
   ...
   Helper1();
}

void Helper1()
{
   if (some condition)
      return;

   ... computation ...;
   reconfigurable Bar { ... };
   Helper1();
}
```

Figure 4-3: Revised primary class `FooBar` showing placement of proxy reconfiguration points.

number of different ways T could be made passive and its state transformed through evolution, depending on the timing of the reconfiguration event. For each of these different ways, the thread will restart at a different point on the new evolved underlying object based on the newly generated method frame.

- First, the call to `Helper1()` is a tail-call. If a reconfiguration event arrives when T enters concrete method `Helper1()`, the method frame for `foo(Bool)` can be folded away, and T can block at the proxy reconfiguration point `Bar`. During evolution, these two frames in the thread would be mapped to a single method frame for the method `foo(Bool)` (as the closest abstract method on the stack), a reference to O_i, and a program counter referring to `Bar` in the next instance.
- Next, since the method `Helper1()` is also tail-recursive, a number of concrete method frames for `Helper1()` could be pushed on to the stack for T. If the reconfiguration event arrives after these tail-recursive calls, all of the concrete method frames and the abstract method frame for `foo(Bool)` can be collapsed and then merged. As in the last case, the closest abstract method frame on the stack is still the `foo(Bool)` frame, and the thread blocks at the reconfiguration point `Bar`. Therefore, after evolution, all of the concrete method frames are removed, and the top-level abstract method frame's program counter is adjusted to reference `Bar`.
- When T executes into the abstract method `foo2()`, the timing of the reconfiguration event determines the reconfiguration point the thread becomes passive at. The thread could become passive at either point `Bar2` or `Bar`.
  - If there is a pending reconfiguration event when the thread reaches `Bar2`, then we can fold away the method frame for `foo(Bool)` since the invocation of `foo2()` is a tail-call. The top-level method frame is updated with a reference to the method `foo2()` instead of `foo(Bool)` with a program counter referring to the reconfiguration point `Bar2`.
  - If the thread reaches `Bar` before a reconfiguration event is received, the rules used to make the thread passive (and correspondingly to evolve the instance) are changed. Since `foo2()` was invoked from the previous method as a tail-call, we can use the rules introduced in this section to use the current method frame as a proxy for the reconfiguration point. In contrast to the last scenario, we fold away the method frame of `foo2()` and instead retain a top-level method frame of `foo(Bool)` with a program counter referring to `Bar`. If instead the thread had been used to directly invoke `foo2()`, instead of passing through `foo(Bool)` first, then it would have ignored `Bar` since that reconfiguration point is not declared as part of this method's abstract signature.
- If T continues on to `Helper1()` through the tail-call in `foo2()`, the same rules for method folding apply. Depending on what reconfiguration points `Helper1()` had, this thread might end up becoming passive on either a method frame with reconfiguration points of `Bar2` in `foo2()`, or `Bar1` in `foo(Bool)`.

4.4.4 Entry Points

These method frame folding techniques have made it possible to evolve an instance in a more timely fashion since threads can become passive more quickly. There are now multiple reconfiguration

points and method frames where the thread can block waiting for quiescence. If the instance is evolved away, the thread could leave the computation in embodied in the current method invocation through any of these reconfiguration points.

However, reconfiguration points also act as entry points for the current method frame. If a thread is being mapped from a previous instance into the current class, then every reconfiguration point declared with the method header in the component mold must be defined in the current method implementation. There is no notion of a proxy reconfiguration point that could act as a method entry point. Once a thread has entered a method through a reconfiguration point, the method implementation could always easily dispatch the thread to a helper method to complete the computation.

Each component is checked statically to confirm that it has a clear entry point within every implementation method for declared reconfiguration points, but it is safe for components to leave out sufficient exit points if so desired. This way, every evolution process that starts would complete.

4.5 Quiescence with Threading

Now we extend our discussion to address multi-threaded applications. Component instances can create new threads that get added to the set of active threads. Rules for quiescence of an instance must now be extended to apply to multiple application threads. Now, a component object is only quiescent *if every thread in the application state is passive* towards the object. As each thread becomes passive, it will block at its reconfiguration point until the entire instance is quiescent, meaning that all threads have become passive. At that point, the encode and decode clauses are executed by a single system thread as before. The reconfigurable clauses for each thread blocked at a reconfiguration point is run sequentially to adjust the new top-level method frame for each thread. After the finally clause is executed, all threads are activated and can run from their current program points.

We require that new threads initiate computation from *outside* the component by invoking a method on the abstract component instance. This allows the previous conditions stated for thread passivity to hold, allowing the instance to evolve. If instead new threads initiated computation from *inside* the component with a method frame that directly referred to an internal instance, then the thread would never be considered passive toward the abstract instance, preventing it from reaching quiescence.

For example, consider an object of primary class A fulfilling component mold B that created a thread T executing the run-loop `run()`. We would need shared reconfiguration points for all of the frames in thread T for the thread to be mappable to an instance of primary class B (which also meets the component mold). By forcing new threads to invoke methods through the component mold interface, we assure there will be shared reconfiguration points, and the thread can be mapped to future instances.

We call these threads that are intended to survive evolution *external threads.* Any application-wide thread that spans the lifetime of any particular primary object is external, including the user thread first used to start the application. The sample implementation of `NetworkStackImplA` introduced in section 3.2 relies on external threads exclusively to handle each pending buffer trans-

mission.

4.5.1 Internal Threads

Component instances might also need to create threads for certain computations that should *not* be mapped over into the new implementation. For example, a network stack might use an internal thread to maintain system statistics particular to the current implementation; it is uninteresting to allow this thread to exist in the evolved instance. For this purpose we introduce the notion of an *internal thread.* The internal thread was first introduced and used in the sample implementation of `NetworkStackImplB`, also introduced in section 3.2.

The internal thread is also tracked as part of the application state. Along with other threads, internal threads must be passive before an instance is quiescent. Once the internal thread T has been created by a particular object, it can invoke methods on other component objects. For those other invoked objects to become quiescent, thread T must *also* become passive toward that object in the same way other external threads must become passive. From the perspective of those other objects, T (and any other internal thread created by another object) are external as well.

The internal thread can become passive differently within the instance where it was actually created. Since the internal thread is not mapped over during evolution, it does not need to block at a reconfiguration point that denotes the end of an abstract core computation. Instead, the only requirement is that the internal thread leaves the instance heap-state in a consistent state that the encode clause can operate upon.

Thus, we introduced another use of reconfiguration points strictly for the purpose of marking heap-state consistency. These private reconfiguration points are unlabeled, and are not declared as part of the method header since they are not part of the shared abstraction. Normal threads ignore these unlabeled reconfiguration points entirely, but internal threads can use both labeled and unlabeled reconfiguration points. Unlabeled reconfiguration points only act as exit points for threads, and never as entry points from previous instances.

Upon evolution, internal threads 'disappear' since they do not have a corresponding context within the new instance. Before this can occur, *all* of the method frames on the thread's stack (and not just those involving this instance) must have completed all changes and left their heap state consistent as well. This is our condition for determining whether an internal thread is passive within the scope of its own instance: *all* method frames in an internal thread must be at a reconfiguration point, either labeled or unlabeled, indicating heap-state consistency.

These conditions for determining passive internal threads differs from the conditions given for general external threads in two ways.

1. These conditions allows a thread directly invoking methods on the underlying instance to be considered passive if the program counter is pointing at a reconfiguration point (either labeled or unlabeled).

2. The internal thread is *not* passive towards its creating instance if it has method frames that referred to *other* component objects that are not fold-able. This condition makes it unlikely internal threads would be used significantly outside of its defining instance.

An internal thread does not have to be specified explicitly. The run-time system can differentiate between internal and external threads by checking the method frame at the bottom of the thread stack. If the frame refers to a concrete internal instance or concrete method, then this thread is considered an internal thread. Otherwise, the thread is considered external.

4.5.2 Synchronization and Deadlocks

Multi-threaded applications use synchronization locks to prevent race conditions where multiple threads access the same data simultaneously. A common issue in designing multi-threaded applications using synchronization is in dealing with deadlocks [37], which occurs when a thread T_1 is blocked waiting for a shared resource A held by thread T_2 while holding a shared resource B that thread T_2 requires.

A number of standard techniques exist that allow user applications to avoid deadlocks. Deadlocks occur because of cycles in the directed-graph of held-locks and lock-dependencies. A simple approach is to order all locks and assure that threads acquire them in the same increasing order; this will also prevents cycles in the directed-graph. As long as the application developer is able to control when locks are acquired by each thread, deadlock can be prevented.

Fundamentally, synchronization locks held by a thread should also be considered part of the instance state where the lock was acquired, since that instance should be held responsible for releasing the lock when necessary. If our internal interface supported declarations of held locks, locks claimed within the current underlying object could also be mapped over to the new underlying object after evolution. However, it is not immediately clear how held-lock abstractions can be reasonably declared in the internal interface, especially since the abstraction would also need to specify some place in the abstract control flow where these locks could be safely released.

For the purposes of this work, we assume synchronization locks of interest are acquired by a thread *outside* of the instance to be evolved (i.e., around a method invocation of this instance), and do not need to be mapped across evolution. Invocations of synchronized component methods in Java can be de-sugared into a pair of lock-acquire and lock-release invocations surrounding the actual method invocation, which would cause the lock to be acquired outside of the component object.

However, our rules for quiescence can make it impossible for the application developer to determine when threads could block. Once a thread is determined to be passive, it blocks while waiting for other application threads to also become passive. If a thread is made passive while holding synchronization locks that another thread requires to become passive, the application can deadlock.

For example, consider thread T_1 as the first to become passive while waiting for instance I of class `FooBarPrimary` to become quiescent, as shown in Figure 4-4. It might hold a lock associated with object A. Before I can become quiescent, imagine another thread T_2 is also invoking a method `foo2` on the instance I.

Thread T_1 will be blocked waiting for T_2 to reach the reconfiguration point Bar2. However, T_2 will never reach that point since it blocks waiting to acquire the lock associated with object A.

```
... some client of FooBarMold ...
  synchronized(A) {
     ...
     FooBarMold I = new FooBarMold();
     I.foo();
     ...
  }

class FooBarPrimary fulfills FooBarMold {
  void foo() reconfigurables Bar
  {
    ...
    reconfigurable Bar { ... }
    ...
  }

  void foo2() reconfigurables Bar2
  {
     ...
     synchronized(A)
     {
        ...
     }
    reconfigurable Bar2 { ... }
  }
}
```

Figure 4-4: Potential deadlock causing invocation and class implementation.

Deadlocks occur despite best efforts of the application developer because our system is inserting additional lock dependencies. Every thread, as it becomes passive and blocks at a reconfiguration point, is essentially waiting for a lock held by every other thread not yet passive. By adding a number of edges into the directed-graph of lock dependencies, we are creating new cycles that lead to deadlock.

DBMS face a similar problem in that the order in which synchronization locks are acquired often can not be limited by developers to prevent deadlock. Instead, DBMS often directly monitor this directed-graph to recognize deadlocks when they occur, aborting transactions as needed to release locks. We use a similar approach.

Our mechanism for achieving quiescence is adapted to take into account held locks. The solution is to create and monitor a directed-graph of locks (including objects) and threads as part of the run-time system.

- When a thread acquires a new lock, we insert an edge into the directed graph from the lock to the thread.
- When a thread blocks waiting for a lock, we insert an edge from the thread to the lock.
- Upon receiving a reconfiguration request, we insert edges from instances to be evolved to every application thread.
- When a thread becomes passive for an instance, we reverse the edge such that it points from the thread to the instance, indicating that the thread is blocked on the instance waiting for quiescence.

Every time the directed graph is modified, the run-time system checks it for a cycle of directed edges. If such a cycle exists, the system searches for an edge that marks a passive relationship between a thread and an instance. If more then one such edge exists, the system can choose one arbitrarily (or more likely using heuristics to select the 'best' choice). That thread is then made *active* again and continues execution from its previous point. This way, progress will continue to be made until a lock is released and the cycle is broken permanently, allowing all threads to become passive and the instance to become quiescent. However, this could lead to starvation for some application threads that remain passive while other system threads run indefinitely.

A simple improvement should lead to better performance. The system can keep a running counter of the number of times a thread has been allowed to continue execution instead of remaining passive. When a new cycle is created, a thread with the lowest counter in that cycle is allowed to go free. This should prevent starvation since all threads will be allowed to execute over time.

Other schemes might analyze the directed graph to locate closely connected thread sets. These are threads that have invoked methods on overlapping sets of instances. When one of these threads go passive on an instance, other threads that are closely connected to this one are selected more often to break cycles of passive threads. This is because these other threads are unlikely to actually become passive on an instance that is quiescent if a thread they are closely connected with is blocked elsewhere. A number of other related heuristics might be used to analyze the directed-graph structure, but the round-robin approach given above remains the simplest, and yet also avoids starvation effectively.

There is little our system can do to break lock-cycles that do not involve a passive thread; the application would have deadlocked regardless.

Wait and Notify

Wait and *notify* are often used to synchronize interaction between multiple threads as well. Two threads can be implicitly connected through this wait and notify relationship, where one thread blocks on a *wait* call while the other thread finishes with some shared resource, at which point it would *notify* the other thread. Unfortunately, while the application developer might be aware of this link, there is no way for the run-time system to detect the existence of this implicit relationship between threads. Thus, there is no way to actively detect dependency cycles and prevent deadlock. The thread that should be using the resource and then notifying the *waiting thread* could become passive and blocked at a reconfiguration point, while quiescence is blocked waiting for the *waiting thread* to become passive.

Alternatively, the developer could use a timed variant of wait where the thread would automatically abort the wait after some time-out period. This would guarantee that some progress is made over time. Or, wait could be annotated (either statically or at run-time) with the set of instances that are possible sources of the notify signal. This makes explicit the synchronization relationship, and again makes it possible for the run-time system to monitor dependencies and prevent passive threads from being a source of deadlock.

4.5.3 Evolution with Multiple Instances

Our application model assumes that evolution occurs on a single instance at a time, although a reconfiguration event should cause all instances of the same component mold to evolve. A thread can only be passive to one instance at a time since it must block at a reconfiguration point in that instance, unless the thread does not refer to the instance at all. This would make it possible for the reconfiguration process to deadlock while waiting for a thread to become passive. Fortunately, the directed graph of thread dependencies generated to track synchronization deadlocks will work just as effectively to counter this problem.

Imagine two instances of the same component mold, A_1 and A_2, and two threads T_1 and T_2. When reconfiguration is requested for these two objects, an edge is created from both instances to each thread. As soon as one of the two threads become passive, for example T_1 on instance A_1, the edge is reversed. If T_2 now also becomes passive on A_1, there is no cycle and A_1 can evolve since it has become quiescent.

If instead T_2 becomes passive on A_2, there is potential for deadlock since the two threads are now passive on different instances, and yet neither instance is quiescent and capable of evolving. Deadlock is averted since a cycle is created on the directed graph when the edge is created from T_2 to A_2: T_1 to A_1 to T_2 to A_2 to T_1; therefore, we would not allow T_2 to become passive on A_2. Instead, it would continue execution until it completed any pending computations on A_1, at which point that instance could evolve.

If we assume that the thread that would actually cause the cycle in the dependency graph is the thread that is not made passive, then the ordering in which instances are evolved depends on the order in which threads become passive. If a thread is *not* passive to a set of instances (which would require that thread actually contained method frames referring to those instances), the first instance of that set it becomes passive to will definitely become quiescent before the rest of the instances in the set. Threads would not be allowed to become passive if it would make contradictory orderings since a cycle would result in the directed graph. In this case, the run-time system has a difficult time predicting an optimal order for causing instances to become quiescent in since it can not predict when threads will become passive toward that instance. Within these constraints, it is possible to imagine that the application developer might actually specify desired orderings for instance evolution that affect which thread is allow to go passive first.

4.6 Java-based Application Reconfiguration

In this section we describe part of our strategy for building a run-time for adaptive applications on top of the Java platform. While the Java platform is sufficiently powerful to support any kind of a system, the challenge is in doing so with the least amount of overhead by leveraging existing services in the run-time, such as garbage collection. In addition, we strongly rely on the Java type system for type safety within our system, both statically during compilation and during runtime. We wish to provide typing mechanisms in Java to support the semantics of our model.

Java provides the class-loader mechanism [25] to make it possible to extend the type hierarchy dynamically at run-time by loading new classes (which define concrete classes as well as interfaces). However, when combined with Java's requirement for strongly-typed and totally type safe opera-

tion, component molds and instances in this system must be handled carefully to guarantee legal operation.

Run-time types are uniquely determined by a ⟨ *defining class-loader, type-name* ⟩ pair. Every class-loader can map a run-time type to only a single class or interface for the lifetime of the class-loader, and although the class-loader is allowed to unload a class (removing this mapping between type-name and class) if there are no further references to the class, its support is strictly implementation-specific and undependable.

In our model of adaptive applications based on components, multiple classes of the same type name could be loaded during the lifetime of the application. If we are upgrading a component implementation to a more recent version of the implementation, for example, class names would very likely conflict. One approach is to use binary rewriting to modify component class names as they are loaded into the system; their names could be munged to create an unique identifier for every implementation of the class. All references to that particular class implementation would also be modified to reflect the new munged type-name. The binary rewriting of class-names requires the implementation of a new class-loader, makes debugging difficult, and would break Java reflection entirely.

Instead, we use a design that leverages the advantages of Java class-loader semantics. In Java, every class-loader effectively defines a single type-space. If a class with a single type-name is loaded in different class-loaders (and type-spaces), it would have different run-time types. Attempts to assign an instance of one run-time type to a variable typed with another run-time type would fail, even if they have the same compile-time type-name. Similarly, attempts to invoke a method on an instance of one run-time type from an instance within a different type-space is also not allowed (without using reflection).

This does not mean that type spaces defined by different class-loaders are distinct, and that instances can not interact in any way. When a new class-loader is created, it can specify a class-loader parent with which it will share a type-space. The new class-loader could then choose to defer certain type-names to the parent class-loader and take on the run-time type the parent class-loader has provided for that type-name. On every instance of the Java virtual machine, there lies a primordial class-loader (we refer to it as L_0). Most instantiated class-loaders choose to use the primordial class-loader as an ancestor such that system classes loaded by the class-loader (for example, `java.lang.Integer`) has a single run-time type of $\langle L_0, java.lang.Integer \rangle$ across all type-spaces.

Therefore, standard types that are intended to be shared across different components, including the component mold declarations (translated into a native Java interface), are deferred to the primordial class-loader for loading and resolution. This is an elegant solution that provides the properties we already declared for a component mold interface; namely, that they are static and unique for the life-time of the application. When a new component is brought into the system, we create a new class-loader as a child of L_0 to hold the component-specific type-space. By instantiating a new class-loader, we avoid class-name collisions; shared type-names from different components (including different versions of a 'component' from the same source) are now contained within different type-spaces and have different run-time types entirely.

Another benefit of using different class-loaders for components is that it hardens the abstractions between them by making it *impossible* for classes to exploit implementation-specific knowledge of

other components. Clients that might know of an implementation-specific type-name can not use the type-name to access that type within the component since they lead to different run-time types. Otherwise, the client would be able to use a simple type-cast to gain direct access to implementation-specific methods and fields.

For example, imagine a component with and a standard-class `FooBarHelper` that implements `java.util.Collection`. This class could include both implementation-specific methods (used to generate the collection) and the abstract `Collection` methods used by clients to view the contents of the collection. Another component that is familiar with this implementation might be tempted (either due to laziness or maliciousness) to call `FooBarHelper` methods directly. The use of separate classloaders and type-definitions help encapsulate components more fully.

4.7 Summary

In this chapter, we extended the rules given for evolving component instances to issues related with real-world applications. We provided an application model for determining instance quiescence. We then introduced various rules for quiescence that allowed us to optimize timely evolution by preserving only instance state that is relevant to the specification given in the abstract component mold. We also considered a number of other real-world application issues, such as deadlock due to concurrent operations and evolutions. Finally, we discussed how this system could be implemented on top of the Java platform using Java classloaders.

Chapter 5

Related Work

This chapter discusses research that is closely related to the content of this thesis. In addition to what is discussed here, there is a huge body of work that has some bearing on the thesis research, including work on specifications of abstract datatypes ([29], [20]), and on the construction of component-based applications either using an object-oriented approach ([5], [32]) or an architectural analogy ([6]).

There are two different aspects to application adaptations. The first aspect concerns structural modifications (such as component replacements) to the application that do not retain state, while the second focuses on the actual evolution process for instances in the application.

5.1 Structural reconfiguration

The system presented in [33] is a framework demonstrating the case of managing run-time evolution for architecture-based applications. The architectural model supported by this system allows only two-sided connectors (called **C2**) representing asynchronous message queues to connect all components. Application reconfiguration, then, consists of the re-binding of connectors to new component instances. Since messages are delivered asynchronously, they are held in the connector until the re-binding is completely. Any state from the previous component instance is not preserved. This system includes a family of tools used to manage the evolution and restructuring process. The authors of [33] hypothesize constraints must be described to limit when and how evolution can occur, but do not explore techniques for doing so.

At the individual component (or type) level, there are two different aspects of stateless reconfiguration. The first is type adaptation, which allows layers incremental changes on top of existing types or implementations. The second approach is straightforward code replacement, which replaces existing code with new code.

5.1.1 Type adaptation

The work in [41] argues that without a shared central authority that specifies component abstractions, different component implementations may only be subtly different (for example, slightly different method names), but still unqualified for straightforward integration. Instead, components must first be statically adapted for a specific use. For example, adaptation could be used to rename implementation methods, or synthesize an abstraction method out of existing implementation methods. This leads to work in [35] that uses user-defined "delta-files" to make dynamic binary changes to deployed Java class files. The mechanisms included in this work supports interface evolution, type-hierarchy evolution, and the definition of adapters that support changed specifications. However, this work only allows off-line component adaptations to the component.

Component adaptation could be used to support evolutionary changes in the component mold. Our system assumes a consistent component mold to allow forward and backward reconfigurations, but type adaptation would allow component implementations to emulate a different (and possibly non-backward-compatible) interface. This allows new implementations to provide consistency to clients still relying on the previous interface, as well as allowing older components to temporarily take the place of a newer interface.

Meta-object systems represent another approach toward adapting component types for this purpose. The work in Encore [39] introduces error handlers to catch attempted accesses to methods or properties that are no longer available in a newer component. These handlers could be used to dynamically map state from the original domain to the current implementation domain. The work in [10] allows for evolution on a larger granularity called *zones*, which is expected to span multiple machines. All types and objects are uniquely identified within each zone. Interaction between different zones are specified via contracts, but change absorbers can then be layered on top of these contracts to shelter evolving types. These changes can introduce online adaptations to types in the application.

5.1.2 Code replacement

The systems in this section only provide mechanisms for replacing part of the implementation in the application. These systems do not have the notion of objects that must be preserved over time.

The most basic example of this is support for dynamically linked libraries provided in systems as early as Multics [34]. Dynamically linked libraries are only loaded at application load-time (or the first time they are used), allowing these libraries to change between application executes. Most systems do not allow dynamic re-binding of linked library references at run-time.

Jini [30] is a family of Java-based technologies and runtimes intended to allow dynamic composition of various network-based resources. In Jini, *services* are the equivalent of traditional components. A service is actually a Java class file dynamically loaded in to the client application to perform some operation, possibly in coordination with a resource available over the network. Jini's run-time support for reconfiguration is based on the use of leases for services. At expiration of the lease for a service, the previous class used to implement the lease is discarded. The application must then re-acquire and re-integrate the service; the new service could potentially have a new class implementation. While the application does not need to terminate during reconfiguration, all

existing instances of the service are lost and must be re-instantiated and initialized with necessary state.

The work presented in [19] take advantage of the abstraction provided by the virtual dispatch table in C++ to allow dynamic replacement of class implementations. After reconfiguration occurs, all new objects of the class created by the class-factory have their virtual dispatch table adjusted to refer to the most recent class implementation. Existing objects of the class are left unmodified.

PODUS (Procedure-Oriented Dynamic Updating System) [28] provides a framework for replacing code at the procedure level. Procedures are updated by changing the binding between procedure name and implementation. A procedure can only be updated when it is considered "inactive", which requires that it is not currently on the invocation stack. PODUS identifies syntactic and semantic dependencies between specific procedure implementations, and requires all of the dependent procedures be inactive such that they may be updated simultaneously to avoid version mismatch. PODUS also supports changes to the interface with "interprocedures" and "mapper procedures" that convert method invocations and static data between different implementation versions. The PODUS approach of linking procedures through semantic dependencies is similar to the abstract component type approach, but the definition of these dependencies is significantly less intuitive than the use of components in an application. The condition of waiting for "inactive" procedures also restricts timely reconfiguration significantly.

5.2 Instance Evolution

Fabry described one of the first systems supporting dynamic implementation changes for abstract datatypes while preserving object state [11]. In his system, abstract datatypes are defined via modules (which are equivalent to components in our system). Change of the datatype implementation is handled through the use of indirection tables for method jumps, which map invocations to the current implementation. If multiple (single-threaded) processes are accessing the module concurrently, a mechanism called *capability revocation* is used to assure that the indirection tables are adjusted only after each process has completed its current invocation. Thus, the application is only ready for reconfiguration once all existing computation in the abstract datatype has stopped. The reconfiguration process atomically changes the indirection table such that all new incoming invocations are sent to the new datatype. Fabry's system also does not support type evolution.

5.2.1 Related Patterns

The state design pattern [14] shares our objective of dynamically adapting object implementations while preserving instance identity. It provides for the separation of changing state implementations from clients. Thus, there are some strong similarities between the state design pattern and the component mold specification.

The component type is analogous to the notion of a *context type* in the state design pattern. In the state pattern, the context type is a concrete class that separates the current state implementation from clients by acting as a proxy that redirects instantiations and method calls to the current implementation. The context type meets the same interface as the underlying implementations in

order to expose appropriate methods. An instance of the context type is semantically very similar to our component object, although the component object does not need to be an actual concrete class.

As part of the state design pattern, the context type implementation is aware of all possible underlying implementations, and will actively switch between candidate implementations as needed. Our design assumes all implementations are independently developed, and not necessarily known to any central repository during development of using code. Thus, the context type class would never actually be implemented. The role of the context type is instead played by the run-time system, which dynamically adjusts bindings to any component implementation that meets the necessary abstractions.

5.2.2 Transform Functions

This section discusses different approaches for the transformation of object state between different representations and implementations.

One approach for implementing this change is exemplified by Fabry's system, which allowed different concrete datatype implementations to have different data-representation formats. Fabry's approach was to transform obsolete instances using programmer-defined functions. Each transform function is specific for the two concrete datatypes it is converting between. This means each transform function is only applicable to a particular implementation change. Fabry's system also allows instances to be evolved lazily, which means that objects are evolved the next time they are accessed, rather than immediately. Multiple representation updates might have arrived by then, and the Fabry system resolves this by queuing all transform functions and applying them in order as evolution occurs.

The transform function approach is also shared by persistent object databases, which use *schema evolution* schemes for adapting object representation overtime. The O_2 [9] system automatically generates transform functions based on changes in the declared type consisting of basic schema evolution primitives. The developer can choose to further augment these transform functions. The work in [38] is built on top of Thor, a distributed object-oriented database management system, and introduces the Lazy Transformation Model (LTM) for "upgrading" typed objects.

Schema evolution primitives are by definition implementation-specific; they add or remove a particular field to a type. Thus, an object in these systems must also have transform functions applied in the same linear order they were defined in to evolve correctly. Under this implementation-specific transform function approach, updates are applied in a specific order.

An alternative approach allows objects to be transformed in any order by using abstraction representations. The DAS (Dynamically Alterable System) [16] is an operating system that supported the replacement of a module with another module meeting the same interface. DAS differs from the previous approach in that each module was expected to specify its own *in* and *out* operations in order to support data-restructuring. Using a common data specification for all versions, the composition of the *in* and *out* operation created a transform function that allowed transformation between any pair of implementations.

Systems in [21] [40] [36] use an approach similar to DAS to support process migration be-

tween heterogeneous environments. Although not strictly object state, these systems use abstract datatype transmission (as discussed in [18] to encode and decode the meta-state related to computation.

The O_2 system further defined two classifications of transform functions: *simple transform functions* that only access the transforming object (as in the Fabry system), and *complex transform functions* allowed to access other objects. If non-backward-compatible changes to the objects are allowed, and transform functions can be applied lazily, a complex transform function may attempt to access objects that have already been evolved in to an incompatible representation. The LTM does allow the use of complex transform functions, but requires that some set of *base methods* used by the transform function must remain consistent between all updates.

While our system does not allow non-backward-compatible changes to the component type, we also have restrictions on the process of generating the component object abstract rep that are similar to those imposed in simple transform functions. We restrict the set of objects that can be accessed during evolution to fully internal instances (as described in Section 2.4.1) to prevent deadlock.

Correctness

In [23] Wing and Ockerbloom presents rules for characterizing the information preserved by a type converter. They formalize the notion of a *respectful type converter*, which is a type transformation (including state mapping) that should assure the new object respects the same super-types (which certainly includes interface types) the previous object respected. Respect of another type, in this case, indicates the new object should have the same observable behavior as the type being respected. Their approach is based on the notion of behavioral sub-typing, and specifies that a respectful type converter should preserve all of the properties of the respected super-type.

Bloom [3] discusses correctness conditions in the context of work on module replacement in the Argus system. He concludes it is too constraining to require new modules (which are similar to objects in that it contains state) to completely satisfy all declared abstractions. Instead, an evolution mechanism would be correct as long as it preserves continuation abstractions that constrain *future* behavior of the module.

5.2.3 Quiescence

Past works in reconfiguration systems have recognized that transformations can only occur at certain times. In the case of systems introduced previously that support only code replacement, the implicit requirement is that the loss of any application state associated with the previous implementation is acceptable. Systems that support state mapping and schema evolution assume their transform function must be operating on a valid and consistent local representation. This implicitly assumes that any on-going computation, either in the form of method invocation or transactions, has completed entirely. Systems for schema evolution, such as O_2 and LTM, often use aborts to force timely completion of on-going transactions. This section discusses past work that gives more consideration to the issue of when evolution can actually occur.

Kramer and Magee's work [24] defined the term *quiescence* as part of their system for node reconfiguration. In their system, the particular reconfiguration abstractions provided could only map instance state when there were no pending transactions. This causes their particular conditions for quiescence to require all pending transactions within an instance to have completed. application capable of initiating transactions to be *passive*, which allows them to continue current on-going computation, but prevents them from initiating new transactions. This approach is used in [2] to support the dynamic reconfiguration of objects in CORBA.

More timely quiescence is possible if the abstraction used to map instance state is more comprehensive. For example, the work by Hofmeister [21] has implicit rules for quiescence that are less constrained than conditions in most systems, including those we introduced previously. This system achieves quiescence if the application thread reaches *any* reconfiguration point, regardless of the state currently present on the thread stack. For all non-top-level stack frames in an instance's state, the program counter points at a method invocation. Hofmeister uses an abstract encoding to capture the underlying meta-state from all of the method frames on the stack, which can then be used to recreate the computation entirely.

This increased power for state mappability comes with the cost of additional abstraction constraints. In fact, the form of quiescence used within this system is only possible if the new implementation is constrained to have *exactly* the same sequence of method invocations and very similar method implementations. It is this constraint that allows the mapping of meta-state to be sufficiently powerful to capture an entire computation. Hofmeister's work is intended strictly for the purpose of process migration, where the same implementation uses this meta-state to continue execution.

Gupta et. al. [7] introduces a formal framework for modeling limited on-line software change. They consider the issue of *valid* evolution, which requires the object state after state mapping be provably *reachable* under the new component implementation. In other words, they propose that the state mapping associated with evolution is acceptable if the new state would have been reached under the new implementation without evolution. They prove that the problem of determining whether a general state mapping is valid is undecidable. However, they show that validity is provable under a simple programming model if evolution only occurs at select points in the execution. Their discussion of a more complicated programming model which includes functions leads them to declare that validity requires all functions to be off the invocation stack, since they do not consider mechanisms for mapping the program counter inside functions.

5.2.4 Reconfiguration Points

Hofmeister [21] first proposed the notion of *reconfiguration points* for state-mapping. The use of reconfiguration points indicates consistency in the underlying implementation, where the meta-state state is consistent and mappable. Reconfiguration points allow consistency even if there are on-going method invocations. The MMLite system [17] uses *clean points* similarly, although their approach does not allow changes to the method implementation.

Work in HERAKLIT [1] extends this notion further by abstracting out reconfiguration points in order to extend their use to differing implementations. In the HERAKLIT system, instances of components can be replaced by sub-classes which inherit definitions of reconfiguration points

(called *consistency points* in this context). Currently executing method calls blocked at consistency points will be able to continue execution in the new sub-class, with consistency points automatically inherited in sub-classes. Actual evolution occurs when every method access in the object is blocked at a consistency point which can be transferred over to the new object. This notion of abstracting out consistency points was an important contribution to this work.

Chapter 6

Conclusions

In this thesis, we have presented a number of techniques used to support dynamic evolution of component instances within an application.

In section 6.1, we summarize our work and primary contributions. Section 6.2 proposes research directions for future work.

6.1 Summary

Applications become more powerful and useful if portions of their implementations can be adapted dynamically at run-time, especially in application domains where the execution environment can change substantially and frequently.

This thesis contributed a programming model for the on-line adaptation of component-based applications. Components are specified via the *component mold*, which defines two separate interfaces: the external and internal interfaces. The external interface provides the abstract datatype of the object by extending a standard type-based interface with constructor declarations. The internal interface provides the abstractions necessary to support the mapping of instance state between different implementations, including meta-state associated with on-going computations. The combination of these two interfaces allows dynamic replacement of component implementations to be entirely transparent to clients of the component, without requiring developers of the component implementation to understand other implementations.

We recognized that the notion of *reconfiguration points*, declared as part of the internal interface, could be used to map the program counter in on-going method invocations across different component implementations. Reconfiguration points indicate places in the methods of the component's objects at which evolution is possible. By specifying reconfiguration point abstractions, different component implementations are constrained sufficiently that an on-going computation in one method implementation, in the form of a method invocation, would have an equivalent computation in another method implementation. This allows the method invocation to continue in either method.

The internal interface also declares an abstract representation shared by all implementations of

the component. Component implementations must define *encode* and *decode* functions that allow instances of the implementation to transform its local concrete representation into, and from, this declared abstract representation.

We described the implementation of components, including the various kinds of classes available to implement the behavior of the component specification. We define the difference between *internal* and *standard* instances to restrict the flow of references within the application, since some objects are no longer a valid portion of the application state upon reconfiguration.

Based on this model, we offered Java language extensions used to declare the necessary component abstractions. We also offered Java constructs that are used to implement the various abstractions, including constructs that enable the encoding and decoding of meta-state information contained within on-going computations. Component implementations written based on these Java language extensions could be pre-processed and translated into normal Java, which would allow the system to run on a standard Java virtual machine.

We contributed an execution model that describe the evolution process, based on notions of *quiescence* and *passive threads*. Component implementations place reconfiguration points declared on the internal interface in its method implementations to indicate points where the meta-state associated with computation contained in an instance of this implementation becomes mappable. By capturing this meta-state associating with method invocations when the instance is quiescent, we can evolve existing component instances in a much more timely fashion than other systems. Most important of all, our execution model has overhead that is substantially reduced from other related works.

By considering all of the state present in a component instance, we determined precisely what must be captured consistently across evolution transformations to satisfy the invariants specified in the component abstraction. This analysis allows us to optimize the conditions for determining whether a thread is passive, making it possible to evolve component objects at points of the execution even when all existing instance state can not be captured consistently.

6.2 Future Work

This section lists some future research directions that extend the results of this work.

Behavioral specifications The component mold specification is limited in the scope of behavior it describes. In this work, the component mold only specifies interface declarations that express type constraints on argument parameters, and syntactic constraints in the form of reconfiguration point labels. It would be desirable to extend the component mold with additional semantic specifications.

For example, the component mold could specify more formally the meaning of the abstract datatype defined by the external interface; this can be used to assure usage and implementation of the component type is correct. Similarly, the abstract rep and reconfiguration points declared in the internal interface might be specified more fully to describe the way concrete object representation and on-going computation should be mapped to the abstract datatype; this could be used to prove that evolution maintains object consistency. The behavioral specifications appropriate for proving

consistency upon evolution for our model is an interesting direction of future research.

Structural reconfigurations This thesis focused on reconfigurations in which the binding between a component and a component implementation can be changed dynamically. However, it does not address the issue of more flexible structural changes in the application. For example, we do not allow the reconfiguration of multiple components with a single implementation that meets their combined abstractions. Similarly, we might wish to compose multiple component implementations, dynamically, to implement the behavior of a single component mold. Achieving all of this with an automated system, augmented with a declarative language, would add substantial power and flexibility to the system.

Component type changes In our system, the component type must remain static for the lifetime of the application and the component implementations. The system should be extended to support backward-compatible evolutions of the component type (via the sub-type), and possibly even non-backward-compatible evolution. This increases component re-use, and opens up a wide range of desirable application reconfigurations (such as simple version upgrades) that would otherwise be impossible. However, this leads to substantially more complexity in the evolution process. One can imagine the creation of version-dependent encode/decode clauses used to transform the current state into an abstract representation that is also version dependent.

Component implementation selection Our work ignored the very important issue of determining optimum reconfigurations. As prior work in software architecture has suggested, the developer could benefit from playing a more active role in choosing component implementations to be used in certain situations. The run-time system could provide automated mechanisms that can adapt the application in response to specific property changes in the execution environment, perhaps by matching predicate attributes defined on each particular component implementation. This could lead to the automatic composition of applications from independent component implementations.

Communication channels The application model we considered used only method invocations as the communication channel between different components. However, applications can also benefit from other paradigms, such as event- or message-based communications. It would be desirable to extend our system to allow the declaration of abstract message queues and event handlers that supported reconfiguration appropriately.

Implementation Our work presented a Java-specific language extension, but a real-world implementation was lacking. It would be desirable to construct a compiler/virtual-machine system that allowed the development of adaptive applications based on the models described in this thesis. In addition, it would be desirable to provide a language-independent compilation back-end to support the integration of components with different programming languages.

Distributed systems Our execution model takes the form of a single monolithic run-time system that can monitor the references and method invocations between all objects in the application.

Distributed systems represent a natural extension of this work, however, since applications in that domain would also benefit greatly from transparent reconfigurations. This would require the creation of a distributed run-time system capable of monitoring cross-host interaction, scanning method stacks across multiple hosts, and managing distributed object lifetimes.

Appendix A

JLS extensions

This chapter introduces additions and changes to material presented in the JLS [15]. First, we present grammars for the Java language extensions provided. Next, we also introduce new rules for determining variable definite assignment (necessary within Java for type-safety).

A.1 Grammar Rules

We have introduced several extensions to the Java language used to declare and implement abstractions necessary for component evolution. The productions shown below are either additions to or modifications of those productions provided as part of the Java Language Specification [15]. This grammar preserves the original JLS's LALR(1) status, and has been implemented for `jikespg`, a parser-generator released as part of the open-source Jikes [22] Java-to-bytecode compiler.

Several new keywords are also added to the lexer:

```
component decode encode fulfills reconfigurable reconfigurables.
```

A.1.1 Reference types

ReferenceType :
    ClassOrComponentOrInterfaceType
    ArrayType

ClassOrComponentOrInterfaceType :
    Name

ClassType :
    ClassOrComponentOrInterfaceType

InterfaceType :
    ClassOrComponentOrInterfaceType

ComponentType :
ClassOrComponentOrInterfaceType

A.1.2 Component Declarations

TypeDeclaration :
ComponentDeclaration

ComponentDeclaration :
Modifiers$_{opt}$ component Identifier
{ *ComponentMemberDeclarations*$_{opt}$ }

ComponentMemberDeclarations :
ComponentMemberDeclaration
ComponentMemberDeclarations ComponentMemberDeclaration

ComponentMemberDeclaration :
ComponentConstructorDeclaration
AbstractMethodDeclaration
AbstractFieldDeclaration
ClassDeclaration
InterfaceDeclaration

The component declaration allows declaration of abstract constructor signatures.

ComponentConstructorDeclaration :
ConstructorDeclarator Throws$_{opt}$ *Reconfigures*$_{opt}$;

Both the new *ComponentConstructorDeclarator* production and the unchanged *AbstractMethodDeclaration* also use changed productions provided for *MethodHeader.*

MethodHeader :
Modifiers$_{opt}$ *Type MethodDeclarator Throws*$_{opt}$ *Reconfigures*$_{opt}$
Modifiers$_{opt}$ void *MethodDeclarator Throws*$_{opt}$ *Reconfigures*$_{opt}$

Reconfigures :
reconfigurables *ReconfiguresList*

ReconfiguresList :
SimpleName
ReconfiguresList , SimpleName

Abstract field declarations do not have initializers, which in other Java types are compiled into a static method in the class file. Without initializers, the bytecode class-file for the component type does not need to include any method implementations, and we do not need to provide complicated rules specifying the order for invoking these initialization methods.

AbstractFieldDeclaration :
Modifiers$_{opt}$ *Type AbstractVariableDeclarators* ;

AbstractVariableDeclarators :
AbstractVariableDeclarator
AbstractVariableDeclarators , *AbstractVariableDeclarator*

AbstractVariableDeclarator :
VariableDeclaratorId

A.1.3 Class Declarations

Primary classes are declared using the optional *fulfills* production.

ClassDeclaration :
Modifiers$_{opt}$ class Identifier *Super*$_{opt}$
Interfaces$_{opt}$ *Fulfills*$_{opt}$ *ClassBody*

Fulfills :
fulfills *ComponentTypeList*

ComponentTypeList :
ComponentType
ComponentTypeList

ClassBodyDeclaration :
...
EncodeClause
DecodeInitializer

EncodeClause :
encode *MethodHeaderMarker Block*

DecodeInitializer :
decode *MethodHeaderMarker DecodeBody*

DecodeBody :
{ *ExplicitConstructorInvocation*
BlockStatements$_{opt}$ *Finally*$_{opt}$ }

ConstructorDeclaration :
Modifiers$_{opt}$ *ConstructorDeclarator Throws*$_{opt}$ *Reconfigures*$_{opt}$
MethodHeaderMarker ConstructorBody

A.1.4 New Statements

ReconfigurableStatement :
reconfigurable Identifier ;
reconfigurable Identifier *Block* ;

ClassInstanceCreationExpression :
new *ClassOrComponentOrInterfaceType* ($ArgumentList_{opt}$)
$ClassBody_{opt}$

ArrayCreationExpression :
new *ClassOrComponentOrInterfaceType DimExprs* $Dims_{opt}$

A.2 Definite Assignment

The rules for determining variable definite assignment (as specified in chapter 16 of the JLS [15]) are adapted to take into account the new method entry/exit and instance initialization points present in this design. Definite assignment rules are necessary to assure type safety such that uninitialized variables can not be accessed within Java.

A.2.1 Reconfigurable Clause

- V is definitely assigned before the first statement of the reconfigurable clause iff V is definitely assigned after the statement immediately preceding the finally clause in the decode clause, or V is a method parameter.
- V is definitely assigned after the reconfigurable clause if: V is definitely assigned after the last statement of the clause or V is definitely assigned in the finally clause in `decode`, **AND** V is definitely assigned before the reconfigurable statement.

A.2.2 Finally Clause (in Decode)

- V is definitely assigned before the first statement in the finally clause iff it is definitely assigned after the statement before the finally clause. (The finally clause can not assume there are *any* active thread contexts that execute a reconfigurable clause.)

Bibliography

[1] P. Arius and W. Betz. Dynamic object replacement in the HERAKLIT system. Technical report, University of Erlangen, Computer Science, 1993.

[2] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, Annapolis, Maryland, May 1998.

[3] T. Bloom and M. Day. Reconfiguration and module replacement in Argus: Theory and practice. *Software Engineering Journal*, 8(2), March 1993.

[4] B. Bokowski and J. Vitek. Confined types. In *Proceedings of the Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, 1999.

[5] Microsoft Corporation. Component Object Model white papers. Available at http://www.microsoft.com/com/wpaper.

[6] D. E. Perry, and A. L. Wolf. Foundations for the Study of Software Architcture. *ACM SIGSOFT, Software Engineering Notes*, 17(4), October 1992.

[7] D. Gupta, P. Jalote, and G. Barua. A formal framework for online software version change. In *IEEE Transactions on Software Engineering*, February 1996.

[8] James Noble David G. Clarke, John M. Potter. Ownership types for flexible alias protection. In *Proc. OOPSLA '98*, Vancouver BC, Canada, October 1998.

[9] O. Deux and et al. The O_2 system. *Comm. of the ACM*, 34(10):34–48, October 1991.

[10] H. Evans and P. Dickman. Zones, contracts and absorbing change: An approach to software evolution. In *Proceedings of the Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, 1999.

[11] R. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 470–476, 1976.

[12] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, pages 236–248, 1998.

[13] J. Flinn and M. Satyanarayanan. Energy-aware adapation for mobile applications. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, Kiawah Island, SC, 1999.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse in object-oriented designs. In O. Nierstrasz, editor, *Proceedings of ECOOP'93*, Berlin, 1993. Springer-Verlag.

[15] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, August 1996. ISBN 0-201-63451-1.

[16] H. Goullon, R. Isle, and K. Lohr. Dynamic restructuring in an experimental operating system. In *IEEE Transactions on Software Engineering*, pages 298–307, July 1978.

[17] J. Helander and A. Forin. MMLite: A highly componentized system architecture. In *Proceedings of the ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[18] M. P. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[19] G. Hjalmtysson and R. Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proc. of the USENIX Annual Technical Conference (NO 98)*, New Orleans, LA, June 1998.

[20] C. A. R. Hoare. Communicating sequential processes. *Comm. of the ACM*, 21(8):666–677, August 1978.

[21] C. Hofmeister. *Dynamic Reconfiguration of Distributed Applications.* PhD thesis, University of Maryland, College Park, MD, 1993.

[22] IBM Research. IBM Research Jikes Compiler Project, 2000.

[23] J. Wing, and J. Ockerbloom. Respectful Type Converters. In *IEEE Transactions on Software Engineering*, July 2000.

[24] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *Transactions on Software Engineering (TSE)*, 16(11):1293–1306, November 1990.

[25] S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. In *Proc. OOPSLA '98*, Vancouver BC, Canada, October 1998.

[26] T. Lindholm and F. Yellin. *The Java Virtual Machine.* Addison-Wesley, Englewood Cliffs, NJ, May 1996.

[27] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison Wesley Publishing, 2001.

[28] M. E. Segal, and O. Frieder. On-the-fly program modification: Systems for Dynamic Updating. *IEEE Software*, pages 53–65, March 1993.

[29] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, 1988.

[30] Sun Microsystems. Jini connection technology: Overview. Available at http://www.sun.com/jini/overview/overview.ps.

[31] Oscar Nierstrasz and Laurent Dami. Component-oriented software technology. *Object-Oriented Software Composition*, 1995.

[32] OMG. *The Common Object Request Broker: Architecture and Specification*, December 1991. OMG TC Document Number 91.12.1, Revision 1.1.

[33] P. Oreizy, N. Medvidovic, and R. Taylor. Architecture-based runtime software evolution. In *Proceedings of the International Conference on Software Engineering (ICSE 98)*, Kyoto, Japan, April 1998.

[34] E. I. Organick. *The Multics System.* M.I.T. Press, Cambridge, MA, 1972.

[35] R. Keller, and U. Holzle. Binary Component Adaptation. In *ECOOP '98 Proceedings*, 1998.

[36] P. Roe and C. Szyperski. Transplanting in Gardens: Efficient heterogeneous task migration for fully inverted software architectures. In *Proceedings of the Fourth Australasian Computer Architecture Conference*, Auckland, New Zealand, January 1999.

[37] M. Roesler and W. Burkhard. Deadlock resolution and semantic lock models in object-oriented distributed systems. In *Proceedings of the 1988 SIGMOD Conference*, pages 361–370, Chicago, IL, 1988. ACM.

[38] Shan Ming Woo. Lazy type changes in object-oriented databases. Master's thesis, Massachusetts Institute of Technology, January 2000.

[39] A. Skarra and S. Zdonik. Type evolution in an Object-Oriented Database. In *Proceedings of the Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, 1987.

[40] M. Theimer and B. Hayes. Heterogeneous process migration by recompilation. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, 1991.

[41] U. Holzle. Integrating Independently-Developed Components in Object-Oriented Languages. In *ECOOP '93 Proceedings*, 1993.