E.P. Kasten and P.K. McKinley, "Adaptive Java: Refractive and Transmutative Support for Adaptive Software," Tech. Rep MSU-CSE-01-30, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, USA, December 2001.

Adaptive Java: Refractive and Transmutative Support for Adaptive Software*

Technical Report MSU-CSE-01-03

E. P. Kasten and P. K. McKinley

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824
{kasten,mckinley}@cse.msu.edu

Abstract

Many middleware platforms use computational reflection to support adaptive functionality. Most approaches intertwine the activity of observing behavior (introspection) with the activity of changing behavior (intercession). This paper explores the use of language constructs to separate these parts of reflective functionality. Specifically, reflective component operations are defined in terms of two types of primitives: refractions and transmutations. These relatively low-level operations can be grouped and combined in different ways to construct different meta-object protocols for various cross-cutting concerns. This separation and "packaging" of reflective primitives is intended to facilitate the design of correct and consistent adaptive middleware. This prototype language was constructed using a source-to-source compiler such that code written using these new language constructs are expanded into standard Java code. Limitations of this approach and insights revealed by our study are presented.

Keywords: adaptive middleware, reflection, component design, mobile computing, wireless networks, forward error correction.

* This work was supported in part by the Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744. This work was also supported in part by National Science Foundation grants CDA-9617310, NCR-9706285, CCR-9912407, and EIA-0000433.

Further Information. A number of related papers and technical reports of the Software Engineering and Network Systems Laboratory can be found at the following URL: `http://www.cse.msu.edu/sens`.

1 Introduction

Increasingly, distributed applications are required to adapt to their environment during execution. This need arises partly from the emergence of a dynamic and heterogeneous mobile computing infrastructure, and partly from users who expect the Internet to provide the same (or higher) quality of service as found in telephone networks and cable television networks. To meet (or even approach) these expectations, distributed software must adapt to its environment in several dimensions. For example, communication software must accommodate wireless networks that are far less reliable and stable than their wired counterparts. In addition, users will expect seamless handoff among different types of networks (cellular PCS, wireless LAN, wired LAN) and computing devices (wearable computer, palmtop, traditional workstation). Hence, user interfaces must conform to devices with widely varying display characteristics and capabilities, from conventional workstations to palmtop devices. Moreover, the reliance on relatively inexpensive commodity hardware resources, both inside the network and at its edge, places much of the burden of fault tolerance on host-level software. Finally, applications must confront the vulnerability of a connectionless packet infrastructure by protecting themselves against intrusions and other security threats.

Adaptability can be implemented in different parts of the system. Integrating adaptive behavior in the application itself is possible, but this approach leads to redundant effort and complex code, and does not address interoperability among different applications. However, even the most advanced network protocols cannot directly meet the changing needs of all applications, and new network-level services are relatively slow to be standardized and deployed. Another approach that is gaining in popularity is to in-

troduce a layer of adaptive *middleware* between applications and underlying transport services [1–10]. Middleware can be used to enhance application performance and functionality regardless of the degree of underlying network support. The appropriate middleware platform can help to insulate application components from platform variations and changes in network conditions, and simplify the maintenance of fault-tolerance and security invariants. Many approaches to the design of adaptive middleware involve computational reflection [11, 12], which refers to the ability of a computational process to reason about (and possibly alter) its own behavior. Typically, the *base-level* functionality of the program is augmented with one or more *meta* levels, each of which observes and manipulates the base level. In object-oriented environments, the entities at a meta level are called meta-objects, and the collection of interfaces provided by a set of meta-objects is called a meta-object protocol, or MOP [13, 14]. Different MOPs may be constructed to handle different dimensions of adaptability [14].

Our interest in reflection arises from our work on the RAPIDware project, which addresses the design and use of adaptive middleware for protection of critical infrastructures, such as power grids, financial systems, and command and control networks. In many such environments, an event as simple as increased packet loss on a wireless channel can trigger different (and possibly multiple) responses from the middleware, depending on the event characteristics and the current system state. Possible responding subsystems include communication quality of service (increasing redundancy on a communication channel), fault tolerance (establishing communication on an alternative network interface) mobile computing (invoking services for disconnected operation), and security (responding to possible malicious channel jamming). The RAPIDware project focuses on developing unified software technologies, based on rigorous software engineering principles, to support different dimensions of adaptability.

We begin our study by examining the foundations of computational reflection. Most adaptive mid-

dleware approaches adopt an interpretation of reflection in which the process of observing behavior (introspection) and changing behavior (intercession) are intermingled. In this paper, explore the idea of using language constructs to separate the two major aspects of reflection. Specifically, we develop a framework for defining metamodels in terms of two types of primitive operations: *refractions*, which provide a (limited) view of the underlying base-level component, and *transmutations*, which modify the functionality of the base-level component. We argue that such separation can simplify the development of adaptive functionality by constraining the possible responses of the system to events while helping to ensure correctness and consistency among different adaptive subsystems. In this manner, the proposed techniques are intended to complement existing approaches to adaptive middleware design by facilitating the development of higher-level adaptive services such as MOPs. Moreover, adaptive systems need mechanisms for defusing concerns about subjectivity [15–17]. Supporting reflective mechanisms in an adaptive system suggests that modification of the reflective interfaces is necessary as the system changes. Without interface adaptation, the meta level won't provide an accurate reflection of the base level. Equally, system adaptors may need augmented, or subjective, views of the base level to make correct decisions. In a sister paper [18] these concepts and methods are used to build *metamorphic sockets* that can adapt to changes in the network environment.

The remainder of the paper is organized as follows. In Section 2, we discuss the origins of reflection and its application to middleware. Section 3 describes the basic concept of separating reflection in order to support the development of metamorphic software. In Section 4, we describe a prototype implementation whereby we added several new constructs to the Java programming language; we refer to the prototype as *Adaptive Java*, or simply *AJ*. Section 5 discusses how *AJ* was implemented. Section 6 discusses related work in the adaptive middleware community. Section 7 discusses what extensions and

enhancements are proposed and discusses future directions. and Section 8 presents our conclusions.

2 Background

Computational reflection, as proposed by Smith [11] and Maes [12], refers to the ability of a computational process to reason about itself. Recently, reflection has gained considerable attention in the middleware community because it offers a principled (as opposed to ad hoc) means to observe and modify base-level behavior [13]. While middleware researchers have long recognized the need for adaptive functionality, in middleware, many systems lacked this degree of formalism. The problem with ad hoc approaches to the design of adaptive middleware is that they do not allow multiple dimensions of adaptability to be addressed in a unified way. Whereas a given external event can potentially affect several different parts of the middleware (and indeed some higher-level functions of an application), such cross-cutting aspects are difficult to express in an ad hoc framework.

Ideally, a middleware designer could systematically specify the relationships between events and several possible middleware responses, such as code migration, tuning of communication protocols, and fault tolerance actions. However, the actual run-time responses will also depend on QoS preferences and security requirements as specified by the application designer. Moreover, a unified framework should enable these preferences to be specified declaratively, thereby insulating the application designer from middleware details. Conversely, ad hoc solutions cannot support such separation of concern because there is no underlying framework for integrating adaptability constraints from multiple sources.

2.1 Properties of Reflection

Maes [12] presents five properties that are considered important in the design and implementation of an object-oriented reflective architecture. These five properties are briefly included here.

Property 1:	Disciplined separation between the object-level and the meta-level.
Property 2:	Uniform self-representation.
Property 3:	Complete self-representation.
Property 4:	Consistent self-representation.
Property 5:	Modifications to the meta-level result in changes to the run-time computation.

These properties were defined with respect to an object oriented language (OOL) designed to support reflection. However, these definitions can be recast to apply to reflective middleware as discussed below.

Property 1 indicates that the meta-level and the level that includes the imperative function, or base-level, are cleanly separated. That is, design and implementation of base-level aspects should not be entangled with the design and implementation of the meta-level aspects. Clearly, this separates the concerns of implementing the imperative functionality from those of reflection, promoting a cleaner design.

Property 2 means that not only are fields, methods, meta objects, etc. modeled as objects, but that these objects also have a refractive and transmutative interface. Thus, observation and modification can take place in much the same way at different depths of encapsulation.

Property 3, from the point of view of the design of a reflective object-oriented language implies each object in the system has a meta-representation. This is impossible, in a practical sense, since each meta object used to compose the meta-level is itself an object. This results in an infinite sequence of meta objects. Practically, this means a way to ”top out” is implemented into the language such that an infinite number of objects won’t be constructed.

Another interpretation of property 3 might be that all useful permutations of the imperative function are represented by the refractive and transmutative aspects. That is, the reflection of the imperative function can be considered complete if it includes an interface to observe and modify all usefully modified characteristics of the imperative function. One way to accomplish might be to reflect every object in the system. However, as discussed above, we know that this results in an infinity of objects and thus is impossible, from a practical point of view, without artificially terminating the sequence of meta object creation. To ensure that all useful permutations are included requires careful selection on where reflection begins and ends. A more practical approach might be to explicitly include the necessary refractive and transmutative interfaces such that useful permutations can be implemented when needed.

Another way to state properties 4 and 5 is that system is causally connected. Property 4 essentially indicates that any modification to the base-level is reflected in the meta-level. Conversely, Property 5 ensures that any modification made to the meta-level is carried back to the base-level. These properties are necessary for reflection to provide a useful interface for observing and modifying a system.

These five properties, even though some of them may or may not be fully attainable, represent the basic goals of a reflective, metamorphic middleware framework. In a nutshell, it is desirable to implement these five properties such that the tendency of the system to preempt how it is later used is *reasonably* minimized. This applies both during the implementation of an application that uses this middleware and during run-time reconfiguration.

2.2 Preemption vs. Open Implementation.

A key issue that arises in the application of reflection to middleware platforms, and the subject of our study, is the degree to which the system should be able to change its own behavior. As discussed by

Kiczales for meta-level interfaces [19] (and earlier by Shaw and Wulf for programming languages [20]) *preemption* occurs when the designer of a programming language or framework makes a decision in the implementation that prevents a programmer from using a feature of the language or framework in a way that would otherwise seem natural. That is to say, decisions made when the framework is implemented preemptively restrict how a programmer can effectively use the framework.

On the other hand, a completely open implementation implies that an application can be recomposed entirely at run-time. Specifically, it is possible for all the default components of the system to be destroyed and and new ones instantiated such that the goal of the imperative (base-level) computation is changed. For example, this extreme allows a calculator to be recomposed as a video player. Thus, a run-time recomposition can produce a system that is inconsistent with the programmer's intended goal. Moreover, a higher degree of openness in a system entails greater resource consumption to construct and maintaining the necessary data structures that enable run-time reflection. In particular, storage and processing power are needed to support meta-models [21–23], which must be reified as objects and maintain a causal connectivity to the base-level.

Typically, the analysis and modification of a run-time system requires that metamodels be reified. Additionally, these metamodels need to be causally connected to the base-level application [12]. These two requirements result in the metamodel being instantiated as additional objects in the running system, requiring storage resources above those used by the base-level application. Causal connectivity requires that additional computational resources are expended to maintain this causal relationship.

Thus, complete openness, particularly at run-time, is not entirely desirable. In fact, it seems that greater openness is more desirable in languages than in run-time reflective systems.

We begin our investigation by focusing on the reflective interfaces exhibited by components. Rather

than considering MOPs as orthogonal portals into base-level functionality [14], we consider an alternative architecture in which MOPs are constructed from a set of primitive operations. While different MOPs address different aspects of behavior, they may well overlap in their use of these primitives.

Figure 1 illustrates this view of MOPs and their composition. Different MOPs are defined for different dimensions of adaptability (e.g., fault tolerance, security, quality-of-service, power consumption) Each MOP accesses the base layer through a subset of the primitive operations, and these subsets may intersect. This design appears to exhibit several desirable features. First, explicitly defining intersections in MOP functionality may facilitate coordinated adaptation to events. Second, additional MOPs can be constructed to address issues that did not arise in the original design. Third, limiting interaction with the base level may improve the ability of the system to check, at run-time, the consistency of modifications with the specified behavior of the component. Finally, since MOPs are composed of mutable primitives, they can be adapted to meet the subjective concerns of adaptive agents. MOPs can be augmented or new ones built such that these agents can construct views of the system to suit their needs.

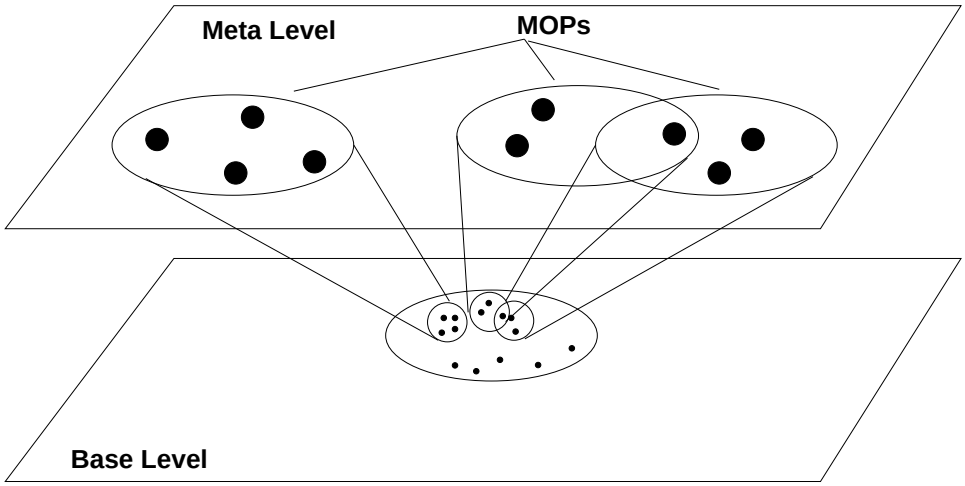

Figure 1. Relationship between MOPs and primitive operations.

In this paper, we propose an approach to defining and constructing such primitive metaoperations that

is based on whether the operation involves introspection or intercession of the base-level. We point out that in her landmark paper on computational reflection, Maes [12] provides the following definition: *Computational reflection is the activity performed by a computational system when doing computation about (and by that possibly affecting) its own computation.* [12] Interestingly, many works on reflection reinforce the parenthetical implication in Maes' definition and combine introspection and intercession. As shown in Figure 2, we can instead view these as functionally orthogonal to each other and to the imperative computation of the application. The computation dimension of the application has the goal of fulfilling the principle goal imbued by the designer. The goal of the introspection dimension is to allow the application to observe itself, while that of the intercession dimension is to allow the application to modify its own behavior and structure.

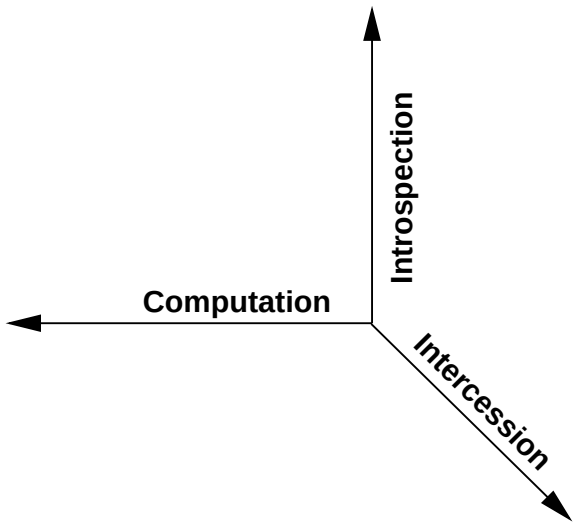

Figure 2. Dimensions of component behavior.

We hypothesize that designing a reflective middleware system such that these aspects of reflection are not entangled will help clarify the meta-interface and meta-object protocol definitions. Additionally, a functional division between the imperative goal and the goals of the observation and modification

aspects may simplify the implementation of system entities that are allowed to observe but not modify components; an example is the data gathering agents of an intrusion detection system or a fault tolerance module.

3 Model of Adaptive Components

The basic building blocks used in our adaptive system are *components*. A component can be accessed through three interfaces corresponding to the three dimensions discussed above. Operations in the computation dimension are known as *invocations*; and we will see shortly why we distinguish invocations from simple methods. Operations in the introspection dimension are called *refractions*, since they offer only a partial view of internal structure and behavior. Operations in the intercession dimension are called *transmutations*; they are used to transform the imperative behavior of the component. Following are formal definitions of these terms.

Computation:	The interpretation of the imperative function of a computer application
Refraction:	The function of observing an application's composition, resources or other internal properties in a principled fashion.
Transmutation:	The function of transforming an application's computational interpretation in a principled fashion.

Refractive and transmutative interfaces are reified by meta components to support introspection and intercession. Figure 3 illustrates the structure implied by our understanding of these component interfaces. In this figure, the meta-level reflects the base-level computation. A causal connection between the meta-level and the base-level is maintained such that any changes resulting from the use of refractive and transmutative interfaces are carried to the base-level.

With the above definitions in hand, a formal definition of computational metamorphosis can be presented.

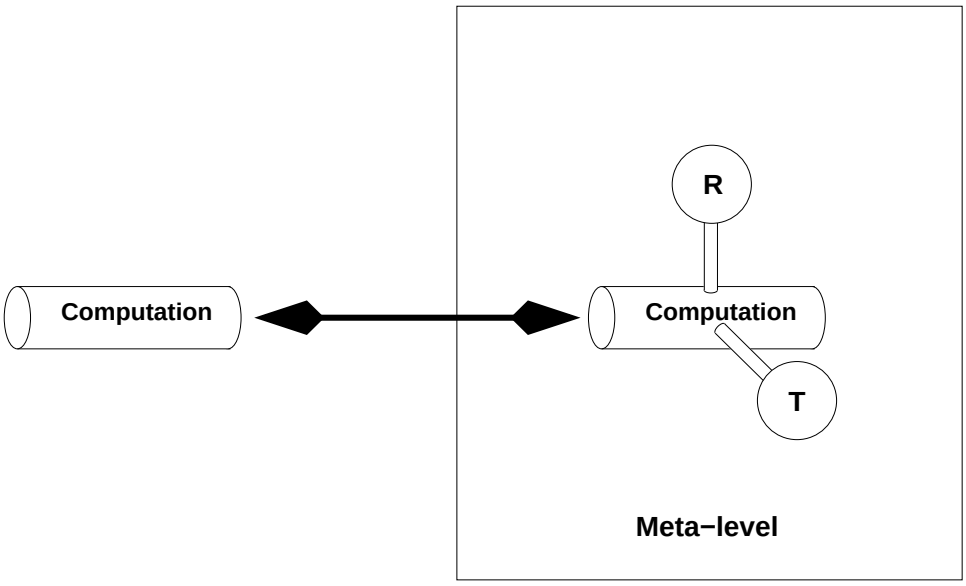

Figure 3. Basic Metamodel

> *Computational metamorphosis* is the principled transmutation of of a running system from one composition to another such that the imperative goal remains unchanged. The set of compositions that have the same imperative goal composes the *morphology* of the imperative function. Each member of this set is called a *polymorph*.

In short, this definition recasts the property of completness such that a run-time adaptive system can be considered complete as long as the imperative goal of the system is maintained. Implicitly, it can be assumed that such a system remains functional such that it does not crash or exhibit anomalous behavior as a result of being recomposed.

3.1 The Role of Encapsulation

Most object-oriented languages are based on a static binding of inheritance between subclasses and superclasses. This structure prohibits dynamic restructuring of a program at run-time. For this reason, we adopt encapsulation as the principle mechanism for the composition in our system. Encapsulation, as shown in Figure 5, provides a means by which the functionality of a component can be extended or

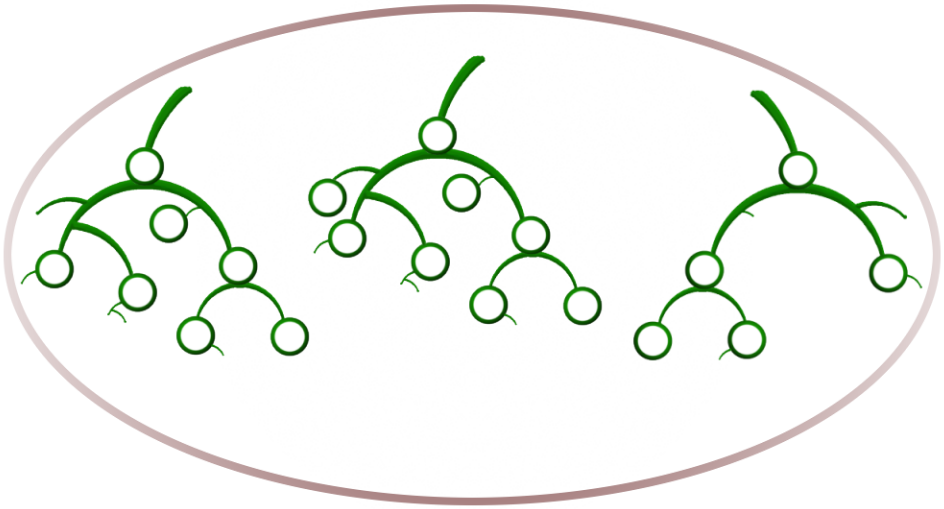

Figure 4. Morphology of an Application

limited by dynamically encapsulating it within another. Moreover, the addition, deletion and exchange of encapsulated components can be carried out dynamically at run-time.

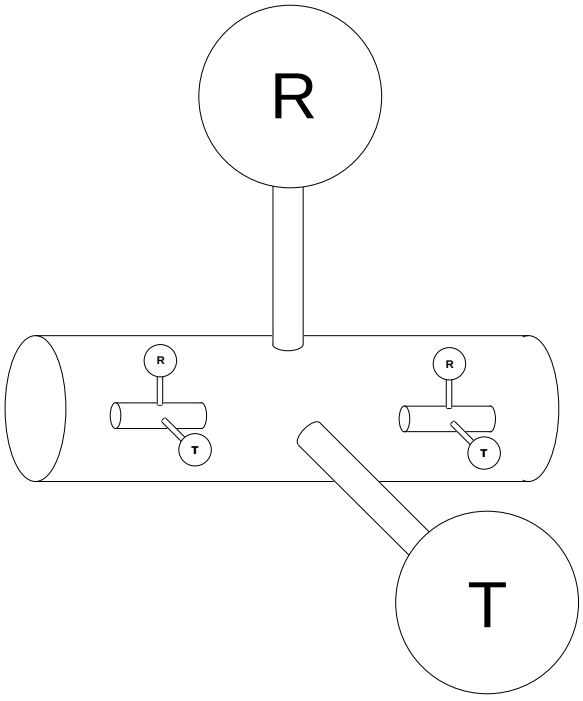

Figure 5. Abstract view of encapsulation.

The composition of a system can be viewed as the parameterization of one component with another. We represent these relationships using notation from GenVoca [24]. Specifically, S = G[F] states that system S is composed of component G parameterized by F, or that F is encapsulated by G. Multiple components can be encapsulated within another component and is represented as S = G[E,F]. Figure 6 shows a structural visualization of the system S = G[E,F[A,B]]. Notice that the morphology of a system built using encapsulation can be described by a set of system definitions:

$$S = \begin{cases} G[E, F] \\ G[E, F[A, B]] \\ H[G[E, F]] \end{cases}$$

Moreover, a system polymorph may result from encapsulting (parameterizing) an existing system within a new component such that S = H[G[E,F]].

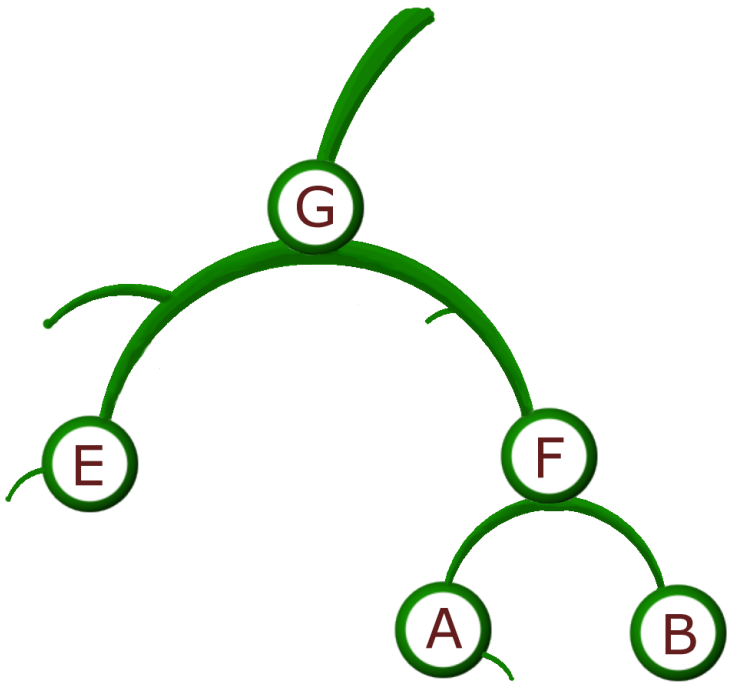

Figure 6. Tree View of a Composition

3.2 Absorption

Components are constructed from objects defined in an object-oriented language (OOL). We used Java in our study. The process of constructing a component from an existing class is referred to as *absorption*. In effect, an object, as provided for by the OOL, can be considered a component without refractive and transmutative capacity. That is, objects are essentially black boxes that do not facilitate reflection.

Absorbing components provides a way to recognize when actions that bore down through the encapsulation layers, such as inheritance, should terminate. Figure 7 illustrates the absorption of a class and the metafication of the resulting "base-level" component to support refractions and transmutations. As part of the absorption procedure, mutable methods called *invocations* are created on the base-level component to expose the functionality of the absorbed class. Invocations are mutable in the sense that they can be added and removed from existing components at run-time using meta-level transmutations. However, the relationship between invocations on the base-level component and methods on the base-level class need not be one-to-one. Indeed, When a component is added to an adaptive system it may be necessary to modify the component's interface such that it fits properly into the system structure. Since component interfaces are mutable and composed of primitive operations, augmentation of an existing interface is possible. Thus, a component can be adapted to achieve a subjective fit. However, some of the base-level methods may be occluded or even combined under a single invocation as the system's form is modified.

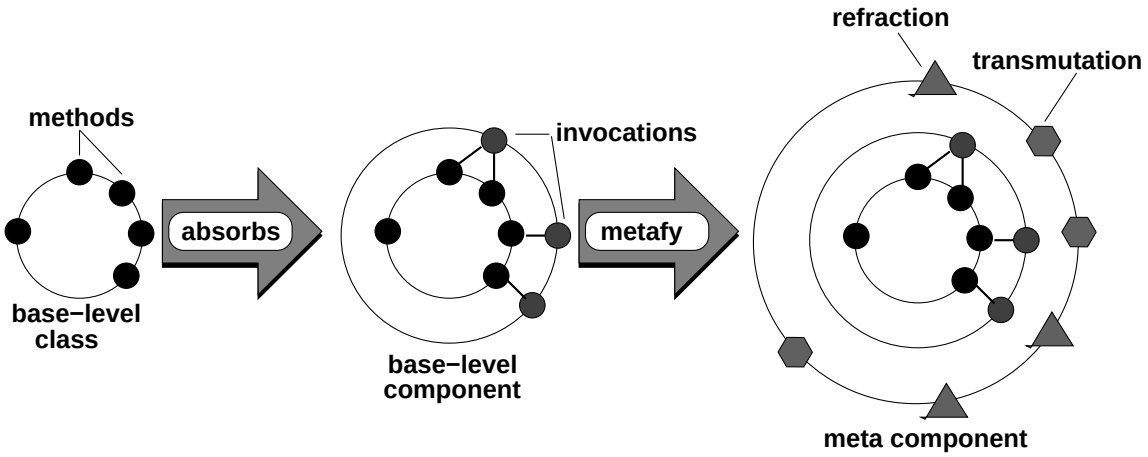

Figure 7. Component absorption and metafication

3.3 Metafication

Metafication is used to create refractions and transmutations that operate on the base component, as shown in Figure 7 Refractions and transmutations embody limited adaptive logic and are intended for defining how the base level can be inspected and changed. The logic for how and why these tool-like operations should be used is provided at other component levels or by other components entirely. That is, a component may refract and transmute itself or a component can be refracted and transmuted by another. For instance, an execution thread providing the imperative functionality of the system might be defined as S = A[B] where A is a meta component reflecting B. Another execution thread, R, could adapt S using the refractions and transmutations defined by A to recompose S = A[C].

Implementing a metamorphic system using components creates a uniform self-representation in two different ways. The first is that, except for the need of boot strapping from the underlying OOL objects, the system can be built with the uniform use of components. The second is that both the base and meta-levels are constructed using components. Thus, the meta-level can also be given a meta-level. This meta-meta-level can then be used to refract and transmute the meta-level. In theory, this reification of

meta-levels for meta-levels could continue infinitely.

4 A Prototype Language: Adaptive Java

In order to gain a better understanding of how the addition of refractive and transmutive elements to a language would affect its use and structure, we defined a prototype language, AJ, as an extension to Java.

In this initial study, we simply used CUP [25], a parser generator for Java, to implement AJ. CUP takes our grammar productions for the AJ extensions and generates an LALR parser, called ajc, which converts AJ code into Java. Semantic routines were added to this parser such that the generated Java code could then be compiled using a standard Java compiler.

We emphasize that AJ is a prototype whose purpose is to improve our understanding of which language constructs and mechanisms are desirable in dynamic and adaptive languages. Eventually, we intend to reimplement the AJ grammar as a compiled language and extend the Java JVM to support needed dynamic constructs such as dynamic casting and immutable invocations and variables. In this section we provide some examples of the language constructs used to code refractive and transmutive software.

4.1 Basic Component Structure

Figure 8 shows the language structure of a typical AJ component. Most Java statements are supported within invocation and constructor blocks. Constructors in AJ are essentially identical to Java constructors and are immutable, only being used to provide flexibility in the initial instantiation of a component. Standard Java methods are replaced by invocations and standard immutable variable declarations are

supplemented with mutable variable declarations. Mutable variables can be added and removed from components, using transmutations at the meta-level, in much the same way as can invocations.

```
/* A simple component */
component BasicComponent {
  /* Constructor */
  public BasicComponent() { ... }
  /* Invocation  */
  public invocation void method1(String arg) { ... }
          :
          :
          :
}
```

Figure 8. AJ component structure.

Optionally, a component can be declared to extend another component. Extending a component encapsulates the extended component within the newly declared component. The extended component is called the *inner component* whereas the extending component is called the *outer component*. Inheritance is simulated by examining the outer component's invocations for the desired invocation. If the invocation isn't found then a recursive search is performed of encapsulated components. For instance, let S = A[B[C]] be a system composed by extending component C with B and then extending B with A. If the execution of invocation `A.exec()` is requested, first component A then B and finally C will be searched for `exec()`. The first instance of `exec()` found will be executed, thus allowing inner invocations to be overridden by those found at more outer encapsulation levels. It is worth noting, that simulating inheritance in this way allows the inheritance chain to be decomposed and recomposed with different components, possibly modifying the internal processing of component composition.

4.2 Absorbing Existing Classes

The **absorbs** keyword is used to construct a component from a regular Java class. Figure 9 shows the AJ code for absorbing a Java socket class into a socket component that is used only for receiving packets. Invocations are created to expose selected functionality of the absorbed class, in this case only the receive and close methods. The absorbed class is accessed by the absorbing component through the **base** keyword. The other methods of the base class are hidden at the this level.

```
/* receive-only socket component */
public component RecvSocket absorbs Socket
{
  /* constructor */
  public RecvSocket(int port, String group,
                    byte ttl)
    throws UnknownHostException, IOException
  {
    setBase(new Recv(port, group, ttl));
  }
  public invocation void receive(DatagramPacket p)
    throws IOException
  {
    base.receive(p);
  }
  public invocation void close()
    throws IOException
  {
    base.close();
  }
}
```

Figure 9. Absorbing a class into a Component

4.3 Reifying a Meta-level

Meta-components encapsulate other components and support only reflective functionality. The encapsulated component is the meta-component's base level. Meta components are declared using the **metafy**

keyword. Figure 10 shows an example in which we metafy the RecvComponent component defined in Figure 9. Both a refraction and a transmutation are defined.

`Refractions` are used to examine the inner, or base, component. The base component is immutable from within a refraction. That is, the base component can only be inspected. Assignment of variables and calling base-level invocations is not supported.

`Transmutations` support intercession of the base level. Invocations of this type can execute only set or invoke operations on the base level. Thus, values of variables can be changed and components recomposed.

```
/* Meta receive-only socket component */
public component MetaRecvSocket metafy RecvSocket
{
  /* Constructor */
  public MetaRecvComponent(int port, String group, byte ttl)
    throws UnknownHostException, IOException
  {
    setBase(new RecvComponent(port, group, ttl));
  }
  /* Transmutation that sets the data stream
     compression level. */
  public transmutation void SetCompression(int level)
  {
            .
            .
            .
  }
  /* Refraction that returns the observed
     bytes transfered by the RecvSocket
     component.  */
  public refraction long GetBytesXmit()
  {
            .
            .
            .
     return bytes_transfered;
  }
}
```

Figure 10. Metafying a component

4.4 Indirection

An invocation is called using the `invoke` keyword, as shown below. This causes the retrieval of a matching invocation object from a HashMap that is then cast to the appropriate invocation type. This effects a prototypical indirect binding for invocations. Indirection enables support for adapting component interfaces and supporting subjective MOPs.

```
invoke rSock.recive(pckt);
```

4.5 Immutability

Immutability is an important concept in the construction of refractions. Refractions, by definition, provide only viewing portals into the processing of the base level program. From the perspective of a refraction, the base level is immutable. Immutability comes in two basic flavors. Shallow immutability is likely the most well known. An object that is shallow immutable disallows any *set* operations (e.g. assigning to a variable) or calls to base level methods. Thus, only *get* operations, e.g. reading a variable, are available to refractions. Deep immutability is a super set of shallow immutability that allows calls to base level methods that don't issue *set* operations or call methods that cause modification of base-level behavior or structure. As of this writing, AJ supports shallow immutability in refractions. However, it is recognized that deriving algorithms and tools that can verify base level methods as usable by refractions would help extend meta level functionality. These types of algorithms could also be used for the automatic categorization of meta level invocations as either refractions or transmutations, obviating the necessity of their explicit declaration.

4.6 Limitations

There are a number of limitations present in our prototype. Many of these limitations can be traced to the observation that the AJ semantic routines output a language that doesn't support many desireable dynamic constructs, such as dynamic casting, or immutability. Uncovering some of these deficiencies (with respect to dynamic software) was instructive and helped further our understanding of adaptive systems. Other limitations, such as much of the generated code being declared public and thus exposed to possible misuse, are not considered serious problems for a prototype implementation. In a production compiler these types of problems will be removed and their existence doesn't inhibit our use of the prototype as a research tool.

5 Implementation Details

Java CUP [25] was used to implement the AdaptJ prototype language. For this initial study, AdaptJ was implemented as a parser, called ajc, that converts AdaptJ code into Java. This Java code can then be compiled just like other Java code. This prototype helped us understand how AdaptJ could be used to implement reflective programs and what language constructs are desirable in languages that support adaptation.

5.1 Language Productions and Parsing

Figure 11 shows the Java CUP [25] grammar productions for the addition of components to the Java [26] language Other productions were also added to support invocations, refractions and other AdaptJ language constructs (See Appendices A, B, C and D). From this extended grammar Java CUP produces a LALR parser. Semantic routines were added to this parser such that Java code was produced

and could then be compiled. Notably, we consider AdaptJ to be a prototype built to help improve our understanding of what language constructs and mechanisms are desirable in dynamic and adaptive languages. Eventually, we intend to reimplement the AdaptJ grammar as a compiled language and extend the Java JVM to support needed dynamic constructs such as dynamic casting and immutable invocations and variables.

```
component_declaration ::=
        modifiers_opt COMPONENT IDENTIFIER component_opts
      ;
component_opts ::=
          interfaces_opt component_body
        | METAFY class_type component_args interfaces_opt
            component_body
        | ABSORBS class_type component_args interfaces_opt
            component_body
        | EXTENDS class_type component_args interfaces_opt
            component_body
      ;
component_args ::=
        | LPAREN argument_list_opt RPAREN
      ;
component_body ::=
        LBRACE component_body_declarations_opt RBRACE
      ;
component_body_declarations_opt ::=
        | component_body_declarations
      ;
component_body_declarations ::=
          component_body_declaration
        | component_body_declarations component_body_declaration
      ;
component_body_declaration ::=
          component_member_declaration
        | static_initializer
        | constructor_declaration
        | block
      ;
component_member_declaration ::=
          field_declaration
        | invocation_declaration
        | mutable_field_declaration
        | modifiers_opt CLASS IDENTIFIER super_opt interfaces_opt
            class_body
        | modifiers_opt COMPONENT IDENTIFIER component_opts
        | interface_declaration
      ;
```

Figure 11. Parse productions for the AdaptJ component (See Appendices A, B, C and D for further details)

AdaptJ code for the extension of an InputStreamComponent is shown in Figure 12. The extended

component is a component constructed by absorbing a standard Java InputStream. It is notable that at this time proper name mangling support for invocations is not provided by the AdaptJ prototype parser. As such, overloaded invocations that only differ in number and type of arguments are not supported. Thus, it was necessary to provide unique names for invocation overloading when extending components. This limitation will be corrected when AdaptJ is reimplmented as a compiled language.

Figures 13, 14 and 15 show an example of the Java code generated from the AdaptJ code shown in Figure 12. This code contains the structures and methods need to store and retrieve invocation objects from a HashMap. As shown in Figure 12, an invocation is called using the `invoke` keyword. An invocation call is then translated into Java as the retrieval of an invocation object from a HashMap that is then cast to the appropriate invocation type. The AdaptJ parser also constructs special abstract classes to support the casting of invocations as shown in Figure 16.

```
import java.lang.*;
import java.net.*;
import java.io.*;
public component MyInputStreamComponent extends
    InputStreamComponent {
  public MyInputStreamComponent(InputStreamComponent aInputStream) {
    setBase(aInputStream);
  }
  protected MyInputStreamComponent() { }
  public invocation void closeISC1() throws IOException {
    invoke closeIS1();
    bytesxfer = 0;
  }
  public invocation int readISC1() throws IOException {
    int n = invoke readIS1();
    bytesxfer += n;
    return n;
  }
  public invocation int readISC2(byte[] b) throws IOException {
    int n = invoke readIS2(b);
    bytesxfer += n;
    return n;
  }
  public invocation int readISC3(byte[] b,int off,int len)
     throws IOException {
    int n = invoke readIS3(b,off,len);
    bytesxfer += n;
    return n;
  }
  public long bytesxfer = 0;
}
```

Figure 12. AdaptJ code for an extended InputStream

```
/*
   Created with the AdaptJ ajc compiler.
   @ADAPTJ version 0.5.0a
*/
import java.util.*;
import adaptj.lang.*;
import  java.lang.*;
import  java.net.*;
import  java.io.*;
public class MyInputStreamComponent extends InputStreamComponent {
  private InputStreamComponent base = null;
  protected void __adaptj_invoke_init__(InputStreamComponent aBase) {
    Iterator iter = this.__adaptj_invoke_table__.values().iterator();
    while (iter.hasNext()) {
      ((AdaptJInvocation)iter.next()).__adaptj_setbase__(
                                                  (Object)aBase);
    }
  }
  public Object __adaptj_invoke_invocation__(String aInvoc) {
    Object invoc = this.__adaptj_invoke_table__.get(aInvoc);
    if (invoc == null) {
      if (this.base == null) throw new
            InvocationNotFoundException("cannot locate invocation "+
              aInvoc);
      invoc = this.base.__adaptj_invoke_invocation__(aInvoc);
    }
    if (invoc == null) throw new
            InvocationNotFoundException("cannot locate invocation "+
              aInvoc);
    return invoc;
  }
  public Object __adaptj_refract_invocation__(String aInvoc) {
    throw new InvocationNotFoundException(
                         "cannot locate refraction "+aInvoc);
  }
  public Object __adaptj_transmute_invocation__(String aInvoc) {
    throw new InvocationNotFoundException(
                         "cannot locate transmutation "+aInvoc);
  }
  public Object __adaptj_getbase__() {
    return (Object)(this.base);
  }
  public void __adaptj_resetbase__(Object aBase) {
    InputStreamComponent typbase = (InputStreamComponent)aBase;
    this.base = typbase;
    this.__adaptj_invoke_init__(this.base);
  }
```

Figure 13. Java code produced by ajc

```
protected void setBase(InputStreamComponent aBase) {
  if (this.base != null) throw new
       IllegalBaseAssignmentException("cannot setBase() a non
          null base");
  this.__adaptj_resetbase__(aBase);
}

public MyInputStreamComponent(InputStreamComponent aInputStream) {
  setBase(aInputStream);
}
protected MyInputStreamComponent() { }
class closeISC1_class extends closeISC1_role {
  public closeISC1_class(__AdaptJ_Basic_Component_Class__ aJMClass) {
    super(aJMClass);
  }
  public void invoke() throws IOException {
   ((closeIS1_role)(this.__adaptj_invoke_invocation__
                                    ("closeIS1"))).invoke();
    bytesxfer = 0;
  }
}
class readISC1_class extends readISC1_role {
  public readISC1_class(__AdaptJ_Basic_Component_Class__ aJMClass) {
    super(aJMClass);
  }
  public int invoke() throws IOException {
    int n =  ((readIS1_role)(this.__adaptj_invoke_invocation__
                                    ("readIS1"))).invoke();
    bytesxfer += n;
    return n;
  }
}
class readISC2_class extends readISC2_role {
  public readISC2_class(__AdaptJ_Basic_Component_Class__ aJMClass) {
    super(aJMClass);
  }
  public int invoke(byte [] b) throws IOException {
    int n =  ((readIS2_role)(this.__adaptj_invoke_invocation__
                                    ("readIS2"))).invoke(b);
    bytesxfer += n;
    return n;
  }
}
```

Figure 14. Java code produced by ajc (cont.)

```
  class readISC3_class extends readISC3_role {
    public readISC3_class(__AdaptJ_Basic_Component_Class__ aJMClass) {
      super(aJMClass);
    }
    public int invoke(byte [ ] b,int off,int len) throws IOException {
      int n =  ((readIS3_role)(this.__adaptj_invoke_invocation__
                                  ("readIS3"))).invoke(b,off,len);
      bytesxfer += n;
      return n;
    }
  }
  public long bytesxfer = 0;
  protected void __adaptj_construct__() {
    this.__adaptj_invoke_put__("readISC3",new readISC3_class(
                          (__AdaptJ_Basic_Component_Class__)this));
    this.__adaptj_invoke_put__("readISC2",new readISC2_class(
                          (__AdaptJ_Basic_Component_Class__)this));
    this.__adaptj_invoke_put__("readISC1",new readISC1_class(
                          (__AdaptJ_Basic_Component_Class__)this));
    this.__adaptj_invoke_put__("closeISC1",new closeISC1_class(
                          (__AdaptJ_Basic_Component_Class__)this));
    if (this.base != null) {
       this.__adaptj_invoke_init__(this.base);
    }
  }
}
```

Figure 15. Java code produced by ajc (cont.)

```
/*
   Created with the AdaptJ ajc compiler.
   @ADAPTJ version 0.5.0a
*/
import java.util.*;
import  java.lang.*;
import  java.net.*;
import  java.io.*;
import adaptj.lang.*;
public abstract class readISC2_role extends AdaptJInvocation {
  protected InputStreamComponent base = null;
  public void __adaptj_setbase__(Object aBase) {
    this.base = (InputStreamComponent)aBase;
  }
  public readISC2_role(__AdaptJ_Basic_Component_Class__ aJMClass) {
    super(aJMClass);
  }
  public abstract int invoke(byte [ ] b) throws IOException;
}
```

Figure 16. Java code produced by ajc for invocation type casting

5.2 Immutability

Immutability is an important concept in the construction of refractions. Refractions, by definition, provide only viewing portals into the processing of the base level program. As such, they are not allowed to recompose or issue modification type operations that affect the base level. From the perspective of a refraction, the base level is immutable.

Immutability comes in two basic flavors with respect to object oriented programs. Shallow immutability is likely the most well known. An object that is shallow immutable disallows any *set* operations (e.g. assigning to a variable) or invocation of base level methods. Thus, only *get* operations, e.g. reading a variable, are available to refractions.

Deep immutability is a super set of shallow immutability that also allows the invocation of base level methods that don't issue *set* operations or invoke methods that cause modification of base level behavior or structure. Essentially, deep immutability allows refractions to invoke base level methods that don't result in base level transmutations.

As of this writing, AdaptJ supports shallow immutability in refractions. However, it is recognized that deriving algorithms and tools that can verify base level methods as usable by refractions would help extend meta level functionality. These types of algorithms could also be used for the automatic categorization of meta level invocations as either refractions or transmutations, obviating the necessity of their explicit declaration.

5.3 Dynamism and Prototype Limitations

There are a number of limitations present in our prototype. Many of these limitations can be traced to the observation that the output of the AdaptJ semantic routines is not a language that supports dynamic recomposition. As such, certain mechanisms, such as dynamic casting and support for run-time immutability, would have obviated the necessity of having to generate special classes for casting invocations and would have eased the implementation of refractions and transmutations.

Uncovering some of the constructs that would be desirable in languages that support dynamic, run-time adaptability was instructive. Insight into what an AdaptJ compiler will need to support and why this support is needed helps further extend our understanding of adaptive systems.

Many other limitations, such as much of the generated code being declared public and thus exposed to possible misuse, are not considered serious problems for a prototype implementation. In a production compiler these types of problems will be removed and their existence doesn't inhibit our use of the prototype as a research tool.

6 Related Work

In recent years, numerous research groups have addressed the issue of adaptive middleware frameworks that can accommodate dynamic, heterogeneous infrastructures. These projects have greatly improved the understanding of how middleware can accommodate device heterogeneity and dynamic network conditions, particularly in the area of adaptive communication protocols and services. Here we focus on those contributions that are most related to the work presented here.

Several project involve adaptive extensions to CORBA [5, 6, 8, 27]. For example, in the Adapt project at Lancaster [5], CORBA is extended to support open bindings, which enable manipulation and reconfiguration of communication paths through the use of object graphs. This mechanism could be used directly to implement dynamically composable services for FEC and other QoS-related functions. In contrast to a CORBA-based design, however, our focus in this study is on programming language constructs to support adaptive interfaces to arbitrary components.

The PCL project being conducted by Adve [28] also focuses on language support for adaptability. PCL is intended for use directly by applications. Our concept of "wrapping" classes with base components is similar to the use of *Adaptors* used in PCL. However, modification of the base class in PCL appears to be limited to changing variable values, whereas AJ transmutations can modify arbitrary structures or subcomponents. Moreover, by combining encapsulation with metafication, AJ can be used to realize adaptations in multiple meta-levels.

Also related is the concept of composition filters [29], which provide a mechanism for disentangling the cross-cutting concerns of a software system. This system declares filters that intercept messages received and sent by objects. As such, messages can be massaged and checked before they are delivered to an object, separating aspects, such as security authentication or bounds checking, from the objects that

are the recipients of these messages. AJ's approach to composition using encapsulation could be used to instantiate a similar message filtering design where components are extended and invocations added such that a call to an invocation would be filtered through subsequent encapsulation layers. Moreover, AJ allows these filters to be adapted during execution to address environmental changes.

7 Future Directions

The absence of good support for immutable objects in Java limits the safety of the prototype's refractive interfaces. Immutable, or read-only, objects require support from the run-time environment in addition to language constructs to properly detect indirect reference and modification of immutable objects. The immplementation of this support in the Java virtual machine requires further study.

Runtime recomposition is a difficult problem. Arbitrary recomposition allows a program to be recomposed such that imperative goals of the developer are preempted. This type of recomposition isn't desirable and needs to be prevented through the use of techniques that can verify a recomposition as being valid. A valid recomposition is characterized the program continuing behave reliably. If a program is not functionally valid, it will crash or function incorrectly. Moreover, recompositions need to be constrained by the imperative goal imbued by the designer. A calculator program that is converted into a video player is not likely desireable. Thus, a valid composition maintains the imperative goal of the design. A recomposed calculator will continue to function as a calculator with enhanced function or improved performance. Language support for the definition of program composition and techniques to properly validate a program's recomposition are needed.

Recomposition of a running system requires mechanisms supporting state management. For instance, Adaptive proxy systems often provide adaptive mechanisms that allow the insertion and removal of filters

into the data stream. Such filter systems require that state information, such as the number of bytes read and written on the stream, be transfered to the components of the new composition. Management of state is necessary for most any compositional change. The addition of checkpointing or state tracing that supports adaptive change would be a valuable addition to the AJ system.

Finally, composition and state management require more than language and basic virtual machine support. A middleware layer needs to be built supporting easy access and use of adaptive mechanisms. The middleware will provide a run-time environment for a program written using the AJ language.

8 Conclusions

In this work, we studied a possible approach, based on separation of introspection and intercession, for designing reflective primitives. We developed a prototype language, AJ, and showed how it can be used to construct adaptive components from existing classes. We intend these low-level mechanisms to provide a foundation for the construction and maintenance of meta-object protocols for cross-cutting concerns (communication quality, fault tolerance, security, power consumption). Our ongoing work addresses two key issues: the use of refractive and transmutative operations to support adaptability through run-time tailoring of component interfaces, and the application of rigorous software engineering techniques to ensure consistency of adapted software.

Acknowledgements. The authors would like to thank S. M. Sadjadi, and R. E. K. Stirewalt for their contributions to this work. This work was supported in part by the U.S. Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744. This work was also supported in part by U.S. National Science Foundation grants CDA-9617310, NCR-9706285, CCR-9912407, and EIA-0000433.

A Java CUP Parser AJ Terminals

```
terminal COMPONENT; // component_declaration
terminal METAFY; // metafy other components
terminal ABSORBS; // absorbs other components
terminal INVOCATION; // invocation method
terminal REFRACTION; // refraction method
terminal TRANSMUTATION; // transmutation method
terminal INVOKE; // component invoke operator
terminal BASE; // base special identifier
terminal REIFY; // reify an existing component
terminal DEIFY; // deify an existing component
terminal COMPOSE; // compose an existing component
terminal REDUCE; // reduce an existing component
terminal USING; // using with reify
terminal REFRACT; // refract invocation
terminal TRANSMUTE; // transmute invocation
terminal COLONEQ; // recompose an existing component :=
terminal DOLLAR; // mutable reference $
terminal MUTABLE; // mutable variable modifier
```

B Java CUP Parser AJ Nonterminals

```
non terminal component_declaration, component_opts;
non terminal component_body, component_args;
non terminal component_body_declarations;
non terminal component_body_declarations_opt;
non terminal component_body_declaration;
non terminal component_member_declaration;
non terminal invocation_declaration, invocation_declarator;
non terminal invocation_assignment;
non terminal invocation_header, invocation_body;
non terminal invoke_invocation;
non terminal refract_invocation;
non terminal transmute_invocation;
non terminal invoke_invocation_body;
non terminal base_expression, base_body;
non terminal reify_expression, deify_expression;
non terminal reduce_expression, compose_expression;
non terminal mutable_field_name, mutable_field_declaration;
non terminal recompose_expression, recompose_assignment;
```

C Java CUP Parser AJ Affected Java Productions

```
name ::= simple_name
| qualified_name
| mutable_field_name
;
block_statement ::=
                local_variable_declaration_statement
        |       statement
| class_declaration
| component_declaration
| interface_declaration
;
statement_without_trailing_substatement ::=
block
| empty_statement
| expression_statement
| switch_statement
| do_statement
| break_statement
| continue_statement
| return_statement
| synchronized_statement
| throw_statement
| try_statement
| reify_expression
| deify_expression
| compose_expression
| recompose_expression
| reduce_expression
;
expression_statement ::=
statement_expression SEMICOLON
| left_hand_side EQ reify_expression
| left_hand_side EQ deify_expression
| left_hand_side EQ compose_expression
| left_hand_side EQ reduce_expression
| left_hand_side recompose_expression
;
statement_expression ::=
                assignment
        |       preincrement_expression
        |       predecrement_expression
        |       postincrement_expression
        |       postdecrement_expression
        |       method_invocation
| class_instance_creation_expression
| invoke_invocation
| refract_invocation
| transmute_invocation
| base_expression
;
```

```
primary_no_new_array ::=
literal
| THIS
| LPAREN expression RPAREN
| class_instance_creation_expression
| field_access
| method_invocation
| array_access
| primitive_type DOT CLASS
| VOID DOT CLASS
| array_type DOT CLASS
| name DOT CLASS
| name DOT THIS
| invoke_invocation
| refract_invocation
| transmute_invocation
| base_expression
;
```

D Java CUP Parser AJ Added Productions

```
component_declaration ::=
modifiers_opt COMPONENT IDENTIFIER component_opts
;
component_opts ::=
interfaces_opt component_body
| METAFY class_type component_args interfaces_opt
                component_body
| ABSORBS class_type component_args interfaces_opt
                component_body
| EXTENDS class_type component_args interfaces_opt
                component_body
;
component_args ::=
| LPAREN argument_list_opt RPAREN
;
component_body ::= LBRACE component_body_declarations_opt RBRACE
;
component_body_declarations_opt ::=
| component_body_declarations
;
component_body_declarations ::=
component_body_declaration
| component_body_declarations component_body_declaration
;
```

```
component_body_declaration ::=
component_member_declaration
| static_initializer
| constructor_declaration
| block
;
component_member_declaration ::=
field_declaration
| invocation_declaration
| mutable_field_declaration
| modifiers_opt CLASS IDENTIFIER super_opt interfaces_opt
                 class_body
| modifiers_opt COMPONENT IDENTIFIER component_opts
| interface_declaration
;
invocation_declaration ::=
  modifiers_opt INVOCATION invocation_header
                 invocation_body
| modifiers_opt INVOCATION IDENTIFIER
                 invocation_assignment
  | modifiers_opt REFRACTION invocation_header
                 invocation_body
| modifiers_opt REFRACTION IDENTIFIER
                 invocation_assignment
  | modifiers_opt TRANSMUTATION invocation_header
                 invocation_body
| modifiers_opt TRANSMUTATION IDENTIFIER
                 invocation_assignment
;
invocation_header ::=
| type invocation_declarator throws_opt
| VOID invocation_declarator throws_opt
;
invocation_declarator ::=
IDENTIFIER LPAREN formal_parameter_list_opt RPAREN
;
invocation_body ::= block
;
invocation_assignment ::=
EQ name SEMICOLON
| EQ NEW class_type LPAREN argument_list_opt RPAREN
                 SEMICOLON
| SEMICOLON
;
invoke_invocation ::=
INVOKE invoke_invocation_body
;
refract_invocation ::=
REFRACT invoke_invocation_body
;
transmute_invocation ::=
TRANSMUTE invoke_invocation_body
;
```

```
invoke_invocation_body ::=
IDENTIFIER LPAREN argument_list_opt RPAREN
| BASE DOT IDENTIFIER LPAREN argument_list_opt RPAREN
| name DOT IDENTIFIER LPAREN argument_list_opt RPAREN
| THIS DOT IDENTIFIER LPAREN argument_list_opt RPAREN
| SUPER DOT IDENTIFIER LPAREN argument_list_opt RPAREN
| name DOT SUPER DOT IDENTIFIER LPAREN
                argument_list_opt RPAREN
;
base_expression ::=
BASE DOT IDENTIFIER base_body
;
base_body ::=
| LPAREN argument_list_opt RPAREN
;
reify_expression ::= REIFY IDENTIFIER USING name SEMICOLON
;
deify_expression ::= DEIFY name SEMICOLON
;
reduce_expression ::= REDUCE name SEMICOLON
;
compose_expression ::= COMPOSE IDENTIFIER USING name SEMICOLON
;
recompose_expression ::= COLONEQ recompose_assignment SEMICOLON
;
recompose_assignment ::=
name
| method_invocation
| invoke_invocation
| class_instance_creation_expression
;
mutable_field_name ::= DOLLAR IDENTIFIER
;
mutable_field_declaration ::=
MUTABLE type IDENTIFIER SEMICOLON
;
```

References

[1] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri, "Generic support for distributed applications," *IEEE Computer*, vol. 33, no. 3, pp. 68–76, 2000.

[2] H. Miranda, M. Antunes, L. Rodrigues, and A. R. Silva, "Group communication support for dependable multi-user object-oriented environments," in *SRDS Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, (Nürnberg, Germany), October 2000.

[3] L. Burness, A. Kassler, P. Khengar, E. Kovacs, D. Mandato, J. Manner, G. Neureiter, T. Robles, and H. Velayos, "The BRAIN quality of service architecture for adaptable services," in *Proceedings of the PIMRC 2000*, (London), September 2000.

[4] T. Kramp and R. Koster, "A service-centered approach to QoS-supporting middleware (Work-in-Progress Paper)," in *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, (The Lake District, England), September 1998.

[5] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin, "A software architecture for adaptive distributed multimedia applications," *IEE Proceedings - Software*, vol. 145, no. 5, pp. 163–171, 1998.

[6] F. Kuhns, C. O'Ryan, D. C. Schmidt, O. Othman, and J. Parsons, "The design and performance of a pluggable protocols framework for object request broker middleware," in *Proceedings of the IFIP Sixth International Workshop on Protocols For High-Speed Networks (PfHSN '99)*, (Salem, Massachusetts), August 1998.

[7] B. Li and K. Nahrstedt, "A control-based middleware framework for quality of service adaptations," *IEEE Journal of Selected Areas in Communications*, vol. 17, September 1999.

[8] R. Vanegas, J. A. Zinky, J. P. Loyall, D. A. Karr, R. E. Schantz, and D. E. Bakken, "QuO's runtime support for quality of service in distributed objects," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, (The Lake District, England), September 1998.

[9] B. D. Noble and M. Satyanarayanan, "Experience with adaptive mobile applications in Odyssey," *Mobile Networks and Applications*, vol. 4, pp. 245–254, 1999.

[10] B. Stiller, C. Class, M. Waldvogel, G. Caronni, and D. Bauer, "A flexible middleware for multimedia communication: Design implementation, and experience," *IEEE Journal of Selected Areas in Communications*, vol. 17, pp. 1580–1598, September 1999.

[11] B. C. Smith, "Reflection and semantics in Lisp," in *Proceedings of 11th ACM Symposium on Principles of Programming Languages*, pp. 23–35, 1984.

[12] P. Maes, "Concepts and experiments in computational reflection," in *Proceedings of the ACM Conference on Object-Oriented Languages (OOPSLA)*, dec 1987.

[13] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas, "An architecture for next generation middleware," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, (The Lake District, England), September 1998.

[14] F. Costa, H. Duran, N. Parlavantzas, K. Saikoski, G. Blair, and G. Coulson, "The role of reflective middleware in supporting the engineering of dynamic applications," in *Reflection and Software Engineering, Lecture Notes in Computer Science 1826*, Springer-Verlag, 2000.

[15] D. Batory and B. J. Geraci, "Composition validation and subjectivity in genvoca generators," in *IEEE Transactions on Software Engineering*, pp. 67–82, feb 1997.

[16] W. Harrison, H. Ossher, and H. Mili, "Subjectivity in object-oriented systems workshop summary," in *Addendum to OOPSLA*, 1995.

[17] W. Harrison, H. Ossher, R. B. Smith, and D. Ungar, "Subjectivity in object-oriented systems workshop summary," in *Addendum to OOPSLA '94 Conference Proceedings*, (Portland, Oregon), pp. 131–136, October 1994.

[18] M. S. Sadjadi and P. K. McKinley, "Design, implementation, and evaluation of metamorphic sockets." `ftp://ftp.cs.arizona.edu/ftol/papers/hotos.pdf`, 2001. in preparation.

[19] G. Kiczales, "Towards a new model of abstraction in the engineering of software," in *International Workshop on Reflection and Meta-Level Architecture*, (Tama-City, Tokyo, Japan), nov 1992.

[20] M. Shaw and W. Wulf, "Towards relaxing assumptions in languages and their implementa tions," *ACM SIGPLAN Notices*, vol. 15, pp. 45–61, March 1980.

[21] L. Capra, W. Emmerich, and C. Mascolo, "Reflective middleware solutions for context-aware applications," in *Proceedings of Reflection 2001*, (Kyoto, Japan), 2001.

[22] H. Duran-Limon and G. Blair, "A resource management framework for adaptive middleware," in *3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'2K)*, (Newport Beach, California), mar 2000.

[23] J. Itoh, R. Lea, and Y. Yokote, "Adaptive operating system design using reflection," in *5th Workshop on Hot Topics in Operating Systems*, may 1995.

[24] D. Batory and S. O'Malley, "The design and implementation of hierarchical software systems with reusable components," *ACM Transactions on Software Engineering and Methodology*, vol. 1, pp. 355–398, October 1992.

[25] S. E. Hudson, ed., *CUP User's Manual*. Usability Center, Georgia Institute of Technology, july 1999.

[26] K. Arnold and J. Gosling, *The Java Programming Language*. Addison-Wesley, second ed., 1997.

[27] C. Becker and K. Geihs, "Quality of service and o.o. oriented middleware multiple concerns and their separation," in *Proceedings of the International Workshop on Distributed Dynamic Multiservice Architectures*, (Phoenix, Arizona), April 2001.

[28] V. Adve, V. V. Lam, and B. Ensink, "Language and compiler support for adaptive distributed applications," in *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001)*, (Snowbird, Utah), June 2001.

[29] L. Bergmans and M. Aksit, "Composing crosscutting concerns using composition filters," *Communications of the ACM*, vol. 44, pp. 51–57, October 2001.