Reflecting on Aspect-Oriented Programming, Metaprogramming, and Adaptive Distributed Monitoring[*]

Bill Donkervoet and Gul Agha

Open Systems Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign
{donkervo,agha}@uiuc.edu

Abstract. Metaprogramming and computational reflection are two related techniques that allow the programmer to change the semantics of a program in a modular fashion. Although the concepts have been explored by researchers for some time, a form of metaprogramming, namely aspect-oriented programming, is now being used by some practitioners. This paper is an attempt to understand the limitations of different forms of computational reflection in concurrent and distributed computing. It specifically studies the use of aspect-oriented programming and reflective actor libraries, and their relation to full reflection. We choose distributed monitoring as the primary example application because its requirements nicely fit the abilities of the two systems as well as illustrate their limitations.

1 Introduction

Addressing software complexity through modular decomposition is an old idea. Objects provide one mechanism for modularity–namely, support for abstract data types, thus separating the interface from the representation. The notion of objects generalizes to components and supports a functional decomposition of large software systems. However, the functional behavior of a system represents only one 'aspect' of this decomposition. Observe that concurrency is common in real-world systems and sequential computation is simply a degenerate case of concurrent computation; therefore, we focus only on concurrent and distributed systems. Concerns in concurrent systems include synchronization, fault-tolerance, scheduling, real-time, coordination, etc. Code to implement these requirements is a major cause of software complexity.

For almost two decades, programming language researchers have explored mechanisms to support a separation of concerns. The goal of separating code to implement functional (or transformational) requirements from code to satisfy other concerns is motivated by the usual advantages of components: namely to

[*] This research has been supported in part by NSF under grant CNS 05-09321 and by ONR under DoD MURI award N0014-02-1-0715.

F.S. de Boer et al. (Eds.): FMCO 2006, LNCS 4709, pp. 246–265, 2007.

simplify the process of building complex software systems and to facilitate reuse of software modules in different contexts. Researchers have proposed a number of techniques to facilitate a separation of design concerns; these include computational reflection [35,29], metaprogramming [10], generative programming [14], aspect-oriented programming [13], filters [2], coordination constraints [22] and circuits [3]. From a theoretical perspective, all these techniques can be understood as forms of metaprogramming although many of them are far more restrictive than (full) computational reflection.

The idea of *metaprogramming* is to allow the manipulation of computational structures containing a representation of a program and its data [35]. With *computational reflection* (or simply *reflection*, a program may inspect and modify itself while running.Reflection and metaprogramming, two related concepts, are tools that not only simplify certain problems but make it possible to address problems requiring dynamic program adaptation. The basis for computational reflection is provided by a foundational concept in the theory of computation as well as computer architecture: namely, that programs are data and may therefore be stored and manipulated as other data. The semantics and architecture of different programming languages and frameworks provide different kinds and degrees of reflection.

On the more static end of the metaprogramming spectrum lies generative programming. *Generative programming* is a form of metaprogramming where additional code is generated based on a high-level specification and the application source code. This generated code is then added to the original source code before interpretation or compilation. *Aspect-oriented programming* may be thought of as a kind of generative programming, enabling code insertion at certain, well-specified points in a program. AOP allows clean, modular specification of both the primary task and other orthogonal aspects, which are then woven together into a single program either at compile-time or runtime.

Our goal is to further the understanding of the semantics of languages and frameworks supporting a separation of design concerns. We focus on two programming concepts that support a separation of concerns: reflection and aspect-oriented programming. Reflection is a powerful programming mechanism with a formal semantics that has been studied in sequential programming languages (e.g., [20,8]) and in concurrent (actor-based) programming languages [5,16].

One way to understand the expressive power of two programming languages is to encode one in the other and show that equivalence relations hold in the translated system. An example of this kind of semantic analysis can be found in [30], which shows that an actor language with remote procedure calls and local synchronization constraints can be translated into a pure actor language (as is done in [27]) without modifying the actor semantics. From a translational perspective, it is straightforward to represent aspect-oriented programming using reflection [9]. Thus, the interesting problem is to understand the limitations of aspect-oriented programming. The approach we take is to pick an example and study its implementation in a reflective actor system and in a programming system supporting aspects in Java.

In order to best understand the strengths and weaknesses of various forms of reflection, we choose an example of a concurrent program that illustrates modularity, orthogonality, and dynamicity. Specifically, we will explore, compare, and contrast these attributes using distributed monitoring. Distributed monitoring of error conditions and constraint violations is a difficult and, as yet, only partially solved problem. Distributed monitoring requires each node (actor) in a distributed system to record and communicate its knowledge of the world, checking for specified conditions. The necessary knowledge may be communicated efficiently by piggybacking state information along with regular messages, thus propagating state knowledge to acquaintances [33].

Although monitoring of statically defined constraints is relatively straightforward, the ability to add and modify such constraints during runtime requires an added level of flexibility. Past solutions have utilized aspect oriented programming to provide insertion of error checks and modification of message format at compile time. We will describe this and other approaches and then discuss the problem of runtime insertion or modification of monitors. It is our conjecture that such flexibility can only be supported in a fully reflective architecture.

The outline of the paper is as follows. Section 2 introduces metaprogramming, leading into a discussion of reflection in Section 3. Aspect-oriented programming is discussed in Section 4. Section 5 addresses reflection's relation to concurrent programming with threads and with actors. Section 6 introduces the case study problem of distributed monitoring and Section 7 describes our implementation. Section 8 surveys related work and the final section concludes with a summary and discussion of open research problems.

2 Metaprogramming

A *metaprogram* is a program that creates or manipulates another program (possibly itself), where the latter program is represented as data. As mentioned earlier, examples of metaprogramming techniques are reflection [35] in which a program can inspect and modify its own behavior, and generative programming, in which the output of a program is the source code for another program [15].

2.1 Meta-architectures

A *meta-architecture* is a representation that not only captures knowledge about a task being performed but also captures knowledge about the performance of the task. Meta-architectures provide access to *metadata*, i.e., access to information about the data. A meta-architecture provides knowledge about a program beyond the information relevant to the application domain. An example of metadata in operating systems is information about a file–such as its creation date, owner, and access information–that is generally stored with the application data.

Access to metadata can enable certain tasks that are not otherwise possible. For example, metadata is useful for governing control-flow in programs as well as for debugging [24]. Every runtime system necessarily maintains some metadata and thus can be thought of as a meta-architecture. The meta-architecture interface greatly affects the flexibility of a system and thus the types of problems a program can address. The systems of interest in this paper are ones that explicitly provide access to not only information about the program's data but also information about the program and its execution.

2.2 Metaobjects

Object-oriented programming languages often maintain metadata in the form of metaobjects. *Metaobjects* contain metadata about their associated objects that specifies how the system is to interpret the data, typically information about the use or representation of associated objects. For example, in Java, the metaobject protocol encodes additional information about methods and variables to allow access by name, information that is typically removed by the compiler.

Because metaobjects are objects themselves, there can also be metametaobjects, metametametaobjects, and so on. Each additional meta-level adds more flexibility and possibilities, but at some point languages ground the hierarchy; for example, a metaobject's meta may itself also be a metaobject, thus eliminating the need for an infinite number of meta-levels. Accessing these metaobjects allows a program to inspect and even modify information about itself.

Smalltalk. Smalltalk is an object oriented language offering strong reflective capabilities using a well-defined metaobject protocol [32]. The metaobject of a Smalltalk object is its class. In Smalltalk, an object may gain an understanding of its structure and behavior by viewing its class. The class of an object provides access to the object's instance variables and methods; thus modification of the class results in immediate modifications to the object's state or behavior.

Because everything is an object in Smalltalk, even a class is an object and has an associated metaobject. The metaobject for a `Class` object is a `Metaclass` object. In order to prevent the need for an infinite number of meta-levels, the metaobject for `Metaclass` is `Metaclass Class`, whose metaobject is again `Metaclass`. Thus the Smalltalk class meta-hierarchy is limited to three levels.

For example, the 3.14 is a `Float` object with an associated `Float Class` metaobject. Modification of the `Float` results in the expected modification of the number. However, modification of the `Float Class` results in redefinition of the behavior of all `Float` objects. The metaobject of the `Float Class` is `Metaclass` and the metaobject of `Metaclass` is `Metaclass Class`.

The ability for a program to view or manipulate its metadata provides powerful yet complex programming techniques. This model of programming is able to address many tasks that are otherwise difficult or impossible.

3 Reflection

With sufficient access to metadata, reflective programming becomes possible. Different systems provide different levels of reflection because of how they limit access to metadata. In order for a program to be fully reflective, modifications to the program's metadata should be reflected in modifications to the data itself. Similarly, modifications to metadata about the program should be reflected in the execution of the program. This produces a causal relationship between the data and the metadata.

Full reflection is composed of two properties: introspection and intercession [32]. *Introspection* is the ability of a program entity to view its internal state or representation. *Intercession* allows program entities to modify their representation, thereby changing the program's behavior. Since introspection is easier to implement than intercession, some languages (including Java) offer only introspection. On the other hand, languages offering intercession also offer introspective capabilities. This is not surprising given that intercession requires a more complex reflective system.

The use of reflection can be illustrated by the following: Consider a robotic arm; a computer maintains data structures that represent the location and position of the arm; these structures are metadata, they simply exist as the program's internal representation of the arm, whereas the actual data is the arm's physical position. Modification of the internal data results in an external movement of the arm. Similarly, external manipulation of the arm results in modification of the internal data. Extending this causal relationship beyond program data to runtime data allows similar modification of the computation itself.

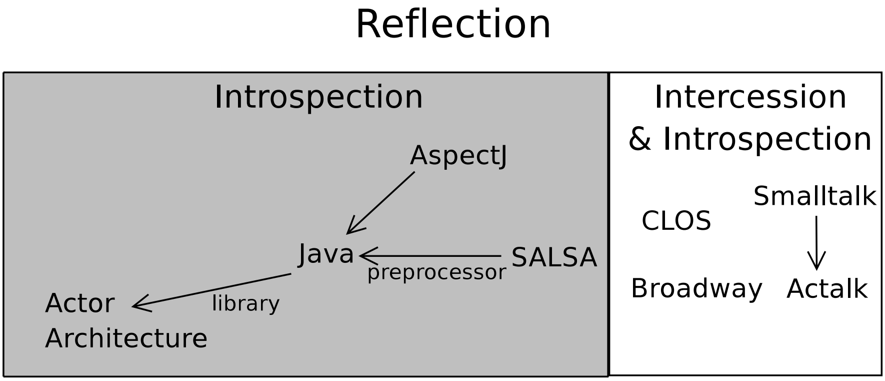


Fig. 1. An illustration of reflective capabilities of various systems. Java and its related systems offer only introspection whereas Smalltalk and CLOS-based systems offer much fuller reflective capabilities.

3.1 Reification

Reification is the process of creating a concrete representation of something abstract. Reification is a necessary step for reflection. In the context of reflection, reification is the act of creating a data representation of the program's internal data (including current state). Once a reification of the program is created, viewing the reification by the program is reflective introspection. Similarly, if the reification is modifiable by the program, reflective intercession is possible.

3.2 Reflection in the Real World

Since reflection allows modification of the program and how the program is interpreted, full reflection even allows modification of the interpreter and runtime system. Thus, most reflective systems offer only a subset of reflective capabilities. For example, Java reflection primarily allows observation of objects and very little modification [23]. Java objects can determine class information, list and access methods and fields, and create new objects. Fields can be accessed and modified but methods cannot be changed so program modification is not possible. Thus, Java offers introspection but not intercession.

On the other hand, Smalltalk has much more powerful reflective capabilities–allowing both introspection and intercession [19]. Objects can dynamically modify their behaviors, change their methods, fields, and even their class. Since the Smalltalk compiler is part of the Smalltalk library, the entire runtime system can be modified–giving the program almost total flexibility. Even so, there are still reflective facilities not offered by Smalltalk for the sake of efficiency and simplicity: in particular, the limited recursion of metaobjects as described in Section 2.2.

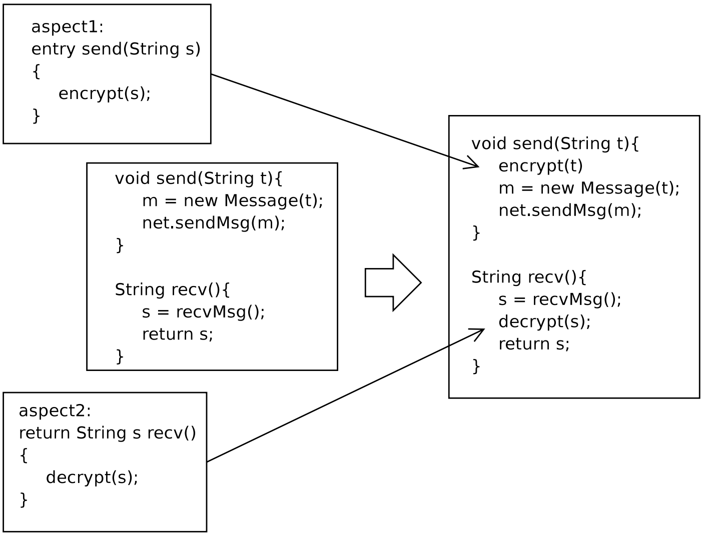
4 Aspect Oriented Programming

Aspect-oriented programming (AOP) is a metaprogramming paradigm that allows separation of concerns through code *cross-cutting*. Different modules of the program are specified, ideally each addressing a singular concern, and these aspects are woven into the main code to produce a single program addressing all concerns. For example, a program can use clean, simple communication method calls and security concerns can later be woven into the program using a security aspect. This not only greatly simplifies the original program but also eases modifications caused by changing specifications; in particular, the original program need not be changed if only some of its aspects change. Similarly, modification of the original program is simplified as it contains only the program logic and not the orthogonal aspect code.

AOP weaves aspects together with the main program at *joinpoints*, points in the program that meet a programmer's specification. Such points must be at designated "control points" such as entry to method calls, exit from method calls, or object creation or deletion [13]. In the above example of network security, encryption wrappers can be applied to a message at the call joinpoint of the network send method and decryption can take place at the return joinpoint of the network receive method. Although the idea of aspect weaving is very standard, the method can vary substantially and be either very dynamic, using reflection, or completely static, using code generation.

4.1 Code Weaving

One form of AOP weaves aspects into the program code statically in the compilation process as a form of generative programming, which requires no runtime meta-architecture. This form of aspect weaving can generate woven programs in the original language or in bytecode. This is then run with the aspects compiled directly and permanently into the program as if the aspects had been hardcoded in initially.



```
aspect1:
entry send(String s)
{
    encrypt(s);
}
```

```
void send(String t){
    m = new Message(t);
    net.sendMsg(m);
}

String recv(){
    s = recvMsg();
    return s;
}
```

```
aspect2:
return String s recv()
{
    decrypt(s);
}
```

```
void send(String t){
    encrypt(t)
    m = new Message(t);
    net.sendMsg(m);
}

String recv(){
    s = recvMsg();
    decrypt(s);
    return s;
}
```

Fig. 2. An example of static code weaving using the secure network example. On the left are two aspects, above and below the regular program code. To the right the code is woven together ready to be compiled and run.

4.2 Reflective AOP

However, if the aspect weaving is done during runtime, certain meta-architectural facilities are necessary. Runtime weaving requires modification of joinpoint behavior and may also require modification of variables.

This behavior modification is achieved through modification of the metadata controlling program execution, although only introspection and a very basic level of self-modification is required. AspectJ, an aspect-oriented language based on Java, utilizes Java's reflective facilities to enable AOP through creating hooks at all specified joinpoints. In the case of a method call joinpoint, the method name and arguments are analyzed to determine if a match occurs and, if so, the aspect code is called before, after, or around the regular method code.

5 Reflection in Concurrency

Because reflection is often a feature of a programming language, some aspects of reflection are unaffected by concurrency. Distributed systems that provide reflection not through a language but through a library often respect the logical bounds of concurrent structures. Reflection need not change the behavior of the entire system but of individual actors [6] or of a group of actors [42]. In case of actors, formal models of reflection have been developed.

5.1 Threads and Objects

A number of languages that have been designed primarily for sequential computing provide concurrency through threading. One example of such of a language is Java. A Java object can be assigned a thread by implementing the `Runnable` interface or subclassing the `Thread` class. However, within that thread and its 'member' objects, all communication is synchronous using regular method calls. Because of this, metaobject protocols in Java typically remain unchanged in the presence of concurrency. Because reflection respects the bounds of objects, procedures, and functions just as threads do, concurrency and reflection are non-conflicting. Any reflection is done entirely within the bounds of a single thread; as a consequence, reflection has no effect on the concurrency in the program.

However, in situations where reflection is able to modify the interpreter, compiler, or runtime system, the effect of reflection can drastically affect how the entire system (including threads) behave. For example, the ability to reflectively modify the runtime system means that the scheduler may be modified, influencing thread behavior. Modern operating systems often export some reflective facilities to programs–in particular, this includes access to the scheduler. As a consequence, processes and threads have the ability to modify their priorities and choose the scheduling policy. In some cases, even greater access is granted: processes may modify the scheduler algorithm [28]. Using these system-wide reflective facilities can greatly influence concurrency in multithreaded programs.

5.2 Actors

Actors are a model of concurrent computation that encapsulate not only data and behavior (as objects do) but also have their own locus of control [1]. Actors can send and receive messages, process messages, create new actors, and compute. Because of their simplicity and inherently concurrent nature, they are a natural model for distributed systems.

Just as object oriented languages introduce metaobjects, actor frameworks and languages often provide meta-actors [37]. These are simply the 'meta' equivalents of actors. Meta-actors allow access to base-actor internals and behavior. Typically meta-actors only intercept incoming and outgoing messages, delaying, modifying, or discarding them to alter the behavior of the actor system. Such modification of message delivery affects the scheduling of actors, and can be used to modularly provide atomicity or enforce a precedence order in the processing of messages. Another form of reflection in actors is providing the ability to copy state. With this additional reflective capability, arbitrary protocol stacks can be defined [5]. The techniques have been applied to provide separate specification and dynamic composition of dependability protocols (security, fault-tolerance, reliability) with distributed application code [38].

For message interception, the actor send and receive methods are modified so that both incoming and outgoing messages are routed through the meta-actor. The normal operational semantics for sending messages in actors are:

$$\texttt{send}\text{: } [R[\mathit{send}(t, m)], \emptyset]_a \rightarrow [R[\mathit{nil}], \emptyset]_a, < t \Leftarrow m >$$

where R is a reduction context, t is the target, m is the message, a is the local actor, and $\emptyset$ signifies that there is no associated meta-actor.

When a meta-actor, $\hat{a}$, is installed to intercept messages, the semantics of sending messages are modified so that, rather than sending directly to the message target, the message is passed to the meta-actor using a transmit message.

$$\texttt{send}\text{ (with meta): } [R[\mathit{send}(t, m)], \hat{a}]_a \rightarrow [R[\mathit{nil}], \hat{a}]_a, < \hat{a} \Leftarrow (\mathit{transmit}(t, m)) >$$
$$\texttt{transmit}\text{ (on meta): } [R[\mathit{transmit}(t, m)], b]_{\hat{a}} \rightarrow [R[\mathit{send}(t, m)], b]_{\hat{a}}$$

The default `transmit` semantics on a meta-actor are simply to propagate the message on to the target. b is either another meta-actor or $\emptyset$. since meta-actors can be layered, as in the *Russian dolls* model [31], these semantics apply regardless of whether there is another higher level of meta-actor. In the case that $b \neq \emptyset$, $\hat{a}$'s overloaded `send` method simply passes the message to the next higher meta-actor. The `transmit` semantics can be overridden to achieve other tasks such as modifying the outgoing message.

Similarly, receive semantics are modified. Normal operational semantics of receive allow an actor a in a wait state to receive and apply message m:

$$\texttt{rcv}\text{: } [R[\mathit{wait}()], \emptyset]_a, < a \Leftarrow m > \rightarrow [R[\mathit{app}(m)], \emptyset]_a$$

In a reflective system, the semantics of receiving messages is replaced by two methods that allow interaction with the meta-actor:

$$\begin{aligned} \texttt{rcv}\text{ (with meta): } & < a \Leftarrow m > \rightarrow < (\hat{a} \Leftarrow \mathit{dlv}(m)) > \\ & \text{where } \mathit{meta}(a) = \hat{a} \\ \texttt{proc}\text{ (with meta): } & [R[\mathit{wait}()], \hat{a}]_a, < a \Leftarrow m >_{\mathit{meta}} \rightarrow [R[\mathit{app}(m)], \hat{a}]_a \end{aligned}$$

`rcv` redirects the incoming message to the meta-actor in `dlv`, a deliver message. `proc` is the second half of the receive, when the message is actually delivered to the actor by it's meta-actor. $< a \Leftarrow m >_{\mathit{meta}}$ is a special message transmission from a meta-actor to its associated base. This is required because a message sent using the traditional means would again be redirected to the meta-actor.

The final piece of the actor/meta-actor communication is the meta-actor semantics for a message receive.

$$\texttt{dlv}\text{ (on meta): } [R[\mathit{dlv}(m)], b]_{\hat{a}} \rightarrow [R[\mathit{nil}], b]_{\hat{a}}, < a \Leftarrow m >_{\mathit{meta}}$$
$$\text{where } \mathit{meta}(a) = \hat{a}$$

Similarly to `transmit`, the `dlv` message can be overridden to achieve desired meta-actor functionality.

Two-level Actor Model. Although the above semantics allow for any number of meta-actors, as in object-oriented languages, actor systems often limit the number of meta-levels for practical reasons. In the Two-Level Actor Model (TLAM), there is a base-level actor and an optional meta-actor [39]. This two-level, reflective architecture provides a dynamic, flexible distributed system in which the meta-actor may alter or enhance the behavior of the base-actor [40].

For c, t actor ids, n a number
States: $T(n)$ Messages: tick, $\mathit{time}@c$, $\mathit{reply}(n)$
Reaction Rules:

$$
\begin{aligned}
&(t|T(n)) : < t \Leftarrow \mathit{tick} > \rightarrow\ (t|T(n+1)) : < t \Leftarrow \mathit{tick} > \\
&(t|T(n)) : < t \Leftarrow \mathit{time}@c > \rightarrow\ (t|T(n)) : < c \Leftarrow \mathit{reply}(n) >
\end{aligned}
$$

Fig. 3. Tick base-level actors in TLAM

Figures 3 and 4 show an example TLAM system borrowed from [39]. The base-level actors (Figure 3), respond to *tick* messages by incrementing their internal counter and sending another *tick* message to themselves; they respond to $\mathit{time}@c$ messages by maintaining their previous state and sending the current counter value to address c. The meta-level actors are used as a logging service and to manipulate a base-level actor by resetting its counter to zero (Figure 4). The first meta-actor rule dictates that on delivering a $\mathit{time}@c$ message to its base-actor, the meta-actor logs the event and participants by sending a *log* message to the observer o. On receipt of a *reset* message, the meta-actor resets the value of its base-level actor's counter to zero and sends a *resetAck* message to o.

For t, o, c actor ids, n, m numbers
States: $M(t, o, m)$ Messages: $\mathit{log}(t, n, m, c)$, reset, $\mathit{resetAck}$
Reaction Rules:

$$
\begin{aligned}
(tm|M(t,o,m)) :\ & \mathit{dlv}((t|T(n)) : < t \Leftarrow \mathit{time}@c >)\ \rightarrow \\
& (tm|M(t,o,m+1)) : < o \Leftarrow \mathit{log}(t,n,m+1,c) > \\
(tm|M(t,o,m)) :\ & < tm \Leftarrow \mathit{reset} > \rightarrow \\
& (tm|M(t,o,0)) : \{/t := T(0)\}, < o \Leftarrow \mathit{resetAck} >
\end{aligned}
$$

Fig. 4. Tick Monitor using meta-actors in TLAM

The logging example shows how meta-actors can be used to address secondary concerns in a modular manner. A more involved problem that has been addressed by TLAM is that of garbage collection [40]. Using meta-actors to create a reachability snapshot, unreachable base-level actors can be detected and garbage collected. Meta-actor functionality may be composed with the TLAM migration service to provide garbage collection that works in the presence of migration. Another application of the model has been to provide modular specification and implementation for Quality of Service requirements in multimedia [41].

6 Case Study: Distributed Monitoring

To understand the strengths and weaknesses of various reflective techniques, we choose to study a specific example. The task implemented by reflection should be orthogonal to the functional (transformational) behavior of an application. However, the reflective behavior needs to be related to the primary task in such a way that it must make use of the state or other internals of the primary task. If there were not such a relation, there would be no reason for the primary and reflective tasks to be joined; they would simply be separate programs.

Considering all of these requirements, we've chosen distributed monitoring as a case study. Distributed monitoring involves runtime observation of safety or error conditions of a distributed system. There are several monitoring approaches, each with advantages and disadvantages. Each approach also requires a different level of reflection, thus offering a different level of flexibility and dynamicity.

6.1 Monitoring Details

Distributed monitoring may be done either centrally or in a decentralized manner. Centralized approaches maintain a single monitor to which all distributed nodes report. This provides a global, sequential view of the entire system and makes monitoring extremely simple. However, this also violates the goals of a distributed system by providing a single point of failure and a system bottleneck hindering scalability.

Distributed monitoring of a distributed system requires monitors local to each distributed node. In order to perform the monitoring task, the monitors must each maintain a view of the entire monitored system. Thus, state data for monitoring is propagated with normal messages between nodes.

Because each node maintains recent state information gathered from messages from remote nodes, it may not have the actual current global state. Thus, the distributed monitor is causally correct as it maintains causally consistent data whereas a centralized monitor is able to assure that the monitor is sequentially correct by maintaining sequentially consistent data [25].

Throughout this section we will use the terms static and dynamic; by *static monitors* we refer to the fact that the installation of monitors must be done at compile time. *Dynamic monitors* means that the monitor may be installed or removed during runtime without redeploying the system. Unfortunately, the terms static and dynamic are somewhat overloaded–even a statically installed monitor observes the system at runtime. Moreover, the dynamicity of a monitor also depends on other factors: a monitor may adapt to changes in a distributed system, such as nodes joining and leaving system. Finally, we wish to address the ability of the monitoring system as a whole to adapt and cooperate through inter-monitor messaging.

Sen et al. introduce past-time distributed temporal logic, `PT-DTL`, as a language for defining monitors to specify restrictions on past or currently-known values at local or remote nodes [34]. Monitor code is generated from the logic requirements that is then woven into the code for deployment.

6.2 Past-Time Distributed Temporal Logic

PT-DTL is a logic for specifying passive monitor conditions using traditional logical and propositional operators. Past-time temporal operators for dictating previous states include previously, always in the past, happens-after, and at some time in the past. These sets of operators form past-time linear temporal logic, PT-LTL. In order to address distributed computation, the epistemic operators $@_{\forall J} F_J$, $@_{\exists J} F_J$, and $@_j(somefunction)$ are added to complete PT-DTL.

For example, the following formula tests for the safety of leader election:

$$@_i(\texttt{leaderElected} \rightarrow ((\texttt{state} = \texttt{leader}) \rightarrow (@_{\{\forall j | j \neq i\}}(\texttt{state} \neq \texttt{leader}))))$$

where the beginning of every PT-DTL statement is $@_i$, stating that the following constraint is being monitored at the local node. leaderElected, state, and leader are local keywords in the monitored process. Finally, the $@_{\{\forall j | j \neq i\}} \ldots$ is the epistemic component; evaluating based on the local knowledge of the most recently known states of other nodes in the system.

Although PT-DTL is a simple distribution of PT-LTL, its distributed nature means that it has a looser consistency model than PT-LTL or a centralized approach. PT-DTL must maintain not only the current values of all other nodes but also the evaluation of past-time logic expressions in order to maintain history.

In addition to the operators provided by PT-DTL, our implementation also provides a means of communication between monitors, allowing cooperation and synchronization. Using this added ability, monitors may perform model-based monitoring where the global monitoring scheme changes in response to the current system state. The new scheme is communicated via this monitor channel and each monitor adapts accordingly. Thus, we require a small logic extension to PT-DTL, which we hope to address in the near future.

6.3 Example Application

Suppose we have a network of distributed temperature monitors to assure a safe operating environment. We wish to ensure that the temperature at any one monitor is not greater than 110% of the average temperature. The PT-DTL formula for such a monitor would be:

$$@_i(\texttt{temp} < (1.10 * \texttt{avg}(@_{\{j | j \; is \; any \; process\}}(\texttt{temp}))))$$

Assuming the datatype holding the knowledge vector is sufficiently flexible to address joining and leaving nodes, this should maintain our operating constraints. However, as simple a change as modifying the alarm threshold to 125% requires flexibility of the monitor itself. In this case, such flexibility could be provided easily enough by using a variable to set the alarm tolerance.

However, using variables, flexible data types, and foresight can only address problems to a point before the amount of flexibility required pushes design requirements into the reflective realm. For example, if instead of monitoring the system as a whole, we wish to monitor each individual room:

$$@_i(\texttt{temp} < (1.10 * \texttt{avg}(@_{\{j | i, j \in room(J)\}}(\texttt{temp}))))$$

or the global system and each individual room with different tolerances:

$$@_i\ ((\texttt{temp} < (1.25 * \texttt{avg}(@_{\{j|j\ is\ any\ process\}}(\texttt{temp})))) \\ \wedge(\texttt{temp} < (1.10 * \texttt{avg}(@_{\{j|i,j \in roomJ\}}(\texttt{temp}))))$$

It quickly becomes evident that a more flexible system is necessary.

Using a computationally reflective system, it would be possible to modify methods instead of just modifying variable values. In the above example, the monitor method could be changed on the fly to accommodate for the changing requirements. Using a non-reflective system, the method change would have to be done in code, recompiled, and then deployed.

6.4 AOP in Distributed Monitoring

Many current distributed monitoring applications use aspect oriented programming to weave the monitor code in with the program code [12,34]. In these systems, monitors are often compiled from a monitor logic into aspects, which are then woven into the primary task's program code in appropriate places.

AOP keeps monitor code orthogonal to program code as stated in our case study requirements. Additionally, aspects have access to program internals such as state and variables allowing program monitoring and can even maintain their own state, which is useful for model-based monitors. Although the AOP approach can address monitoring requirements, it is not as flexible as other reflective approaches. Namely, since aspects must be woven into program code, they cannot be added, modified, or removed at runtime but are set at compile time.

6.5 Reflection in Distributed Monitoring

Reflection offers the flexibility necessary to address the distributed monitoring problem. Besides being dynamic enough to allow growing knowledge vectors, sufficient reflection can also allow in-place modification and removal of monitors.

Suppose a distributed system uses the actor framework, monitors could be implemented as meta-actors. Unmonitored actors would remain untouched while actors with installed meta-actors as monitors would have modified semantics. Note that meta-actors, being actors themselves, would use normal actor semantics unless they have installed monitors, and thus meta-actors, of their own.

Using the semantics given in Section 5.2, the meta-actor `transmit` semantics can be overridden to modify the outgoing message to include the knowledge vector, in the case of our monitors.

$$\texttt{transmit}\text{ (on monitor): } [R[transmit(t,m)],b]_{\hat{a}} \rightarrow [R[send(t,m'')],b]_{\hat{a}} \\ m'' = KVm(m, kv(a))$$

In order to deal with the knowledge vectors on a monitor, we simply remove $kv(a)$ and propagate the original message to the recipient actor:

$$\texttt{dlv}\text{ (on monitor): } [R[dlv(KVm(m,kv(t))],b]_{\hat{a}} \rightarrow [R[nil],b]_{\hat{a}}, < a \Leftarrow m >_{meta} \\ \text{where } meta(a) = \hat{a}$$

Monitors as meta-actors would have access to actors' internal states, also needed for the knowledge vectors. With all of this, the monitors would be able to track distributed system computation and check for errors. Because the meta-actors can reflectively modify themselves during runtime, monitors can be modified or removed on the fly without the need to redeploy monitored processes.

7 Implementation of Adaptive Monitors

A proof-of-concept implementation was done building upon Actor Architecture. Although the implementation language does not provide strong reflective capabilities, reflection using a library and indirection enabled us to achieve our goal of a dynamic distributed monitoring system.

7.1 Actor Architecture

The *Actor Architecture* (AA) is a Java actor framework providing a full actor implementation running on one or more systems [26]. AA handles message routing and actor migration and consists of two main parts: the platform, which provides all of the 'background' services, and the actor itself.

Actors in AA are simply Java subclasses of the `Actor` class. The `ActorThread` sleeps until a message is inserted into its mail queue by the local `MessageManager`. The `ActorThread` then processes the message by parsing the requested method and the corresponding arguments. If the method exists with the correct number of arguments in the actor object, the method is called.

7.2 Monitor Installation

Once an actor is created, a monitor is installed by sending a message. The message simply provides the monitor name and calls the actor class' `addMonitor()` method. This method uses Java's reflective facilities to create a new instance of the class named by the string argument. The newly created monitor object is then added to the list of this actor's monitors.

Although this installation method requires the monitor bytecode to be present on the system, there is no reason this is necessary. It is feasible to serialize the monitor and send it with the message to each actor. However, both schemes are equally flexible and only differ in when the monitor code is transferred. The class also has a method `removeMonitor()` to remove currently installed monitors.

7.3 Knowledge Vectors

The `KnowledgeVector` class is a collection of `KVEntry` objects. Each `KVEntry` contains only a value and a timestamp and the `KnowledgeVector` keeps a mapping between the entries and the associated actor names.

Scattered throughout the `Actor` and `ActorThread` code are calls to the `updateKV(KnowledgeVector)` method. This method call is necessary every time the local knowledge vector can change. Thus, every time a message is received at an actor, the local knowledge vector is compared with the message's piggybacked knowledge vector and necessary updates are incorporated.

After message processing and the corresponding call have completed, the actor's `updateLocalKV()` method is called in order to assure that the local knowledge vector contains the most current entries for local variables. Ideally, the knowledge vector update would be triggered by modification of any monitored variables but since actors are purely reactive, any modification must occur as a result of a message. Thus, updating the knowledge vector after completion of message processing guarantees that all variable modifications are recorded.

In order to update the local portions of the knowledge vector, the method iterates through each locally monitored variable and reflectively reads it. Java reflection is necessary in order to obtain references to the variables by only the string identifier. The `updateLocalKV()` method then records the current values and the current timestamp in the knowledge vector.

7.4 Actor Monitor

The `ActorMonitor` is an abstract class containing methods and variables related to the monitors. A monitor is created by extending the `ActorMonitor` class and implementing the `evaluateMonitor()` method.

Aside from the evaluate method, there are also two methods for getting and setting the list of monitored variables. Monitored variables are stored as a string that is the concatenation of the actor name or `local` and the variable name. The `local` keyword specifies that the variable will be evaluated at each actor locally whereas the standard actor names specify that the variable will only be evaluated at the specified actor.

```
public MyMonitor(){
    setMonitoredVariable("uan://127.0.0.1:2/counter");
    setMonitoredVariable("local/counter");
}
```

Fig. 5. The monitor's constructor showing manual listing of monitored variables

The `setMonitoredVariable()` modifier is only used within the constructor of the monitor to initially create the list of variables used in the monitor formula. The `getMonitoredVariables()` accessor is used in the above mentioned `updateLocalKV()` method to access the locally monitored variables.

Each time the knowledge vector is updated, the actor evaluates its installed monitors. The list of monitors is iterated through and each one is evaluated in turn using the newly updated values.

The `evaluateMonitor()` method does the actual evaluation of the monitor formula. The method pulls out the most currently known variable values from the local knowledge vector. Using these values, the formula is evaluated and a boolean value stating the result of the formula is returned. In the case where the knowledge vector is too sparse to evaluate the monitor, the method simply gives up and returns success. In addition to the knowledge vector, the local actor name is also passed in for determining which entries correspond to the local actor.

```
public boolean evaluateMonitor(KnowledgeVector kv, String locActorName){

    try{
        int myCnt = kv.get(localActorName + "/counter").getValue();
        int remoteCnt = kv.get("uan://127.0.0.1:2/m_iSum").getValue();

        if(myCnt > remoteCnt)
            System.err.println("MONITOR FAILS!!!! at " + localActorName);
        else if(myCnt > THRESHOLD)
            aboveTH = true;
        else
            aboveTH = false;
    }
    catch(NullPointerException e){
        // just ignore it if the knowledge vector is not full enough
    }
    return(new KVEntry(timestamp++, new Boolean(aboveTH)));
}
```

Fig. 6. The monitor's evaluate method. This monitor ensures that the counter value at every node is never greater than the counter value at actor 127.0.0.1:2.

The `evaluateMonitor()` method returns a KVEntry, which is then added to the node's knowledge vector. This provides means of communication between monitors allowing them to cooperate and communicate state information. In this way, monitors can cooperatively execute model-based monitoring.

7.5 Implementation Discussion

Currently, the `evaluateMonitor()` method and the monitored variables must be written by hand. However, we intend to incorporate automatic generation of these from a provided PT-DTL formula.

This implementation provides dynamic, adaptive distributed monitoring but is still constrained by Java's limited reflection. Objects can be created and passed around and their methods may be called but a more reflective language would allow even greater flexibility. For instance, implementation in Smalltalk would allow program modification in addition to program introspection.

With the Java implementation, any monitor adaptability has to be coded into the monitor. Monitors cannot simply be installed on any distributed application, the application must be designed with monitoring in mind. An analogy would be having to design an application for debugging versus being able to use a symbolic debugger on any application. Because of Java's weak reflective capabilities, all actors in our implementation must contain code to process knowledge vectors whether a monitor is installed or not.

With a sufficient level of reflection, any application can be monitored without any special design through reflectively overriding the appropriate methods. The send and receive methods would not need to handle knowledge vectors in the case where no monitor is installed but would be modified upon monitor installation. A Smalltalk implementation is currently underway that uses reflection and meta-actors. Using these, monitors may be installed on any actor implementing these two methods without any special design to accommodate monitors.

8 Related Work

We have used distributed monitoring as a case study to demonstrate the need for and different faces of reflection. As with any problem, there are many possible solutions, each with its own advantages and disadvantages. Many previous systems have offered various solutions to distributed monitoring utilizing different levels of reflection, allowing varying capabilities.

Chen and Rosu introduce *Monitor Oriented Programming* (MOP), which separates monitor specification from the program, much like AOP [12]. In addition to the program implementation, a formal specification is provided. This specification is translated into runtime monitors that are installed similarly to aspects. Should any constraints be violated, user specified code will run to recover from or report the error. However, also like AOP, only limited reflective capabilities are utilized. Indeed, the current implementation of MOP, JavaMOP, compiles the specifications into AspectJ code and thus is constrained by its limitations.

Another area in which reflection is frequently used is active monitoring. The monitors described thus far in this paper are all passive in that they in no way affect the regular operation of the system but monitor quietly in the background. Active monitors visibly affect and participate in the operation of their monitored nodes. *Synchronization constraints* are one form of active monitor used for distributed synchronization [21]. Meta-actors simply delay delivery of messages until a specified condition is met and then delivery continues as normal.

Similarly Aksit et al. introduce an AOP model permitting composition of multiple filters [2]. *Filters* are meta-level objects that intercept messages to program objects, apply specified conditions and then allow or disallow message delivery or perform a specified action. Program objects can be reified by a filter permitting observation or modification of the object. Filters can be layered to achieve complex functionality while still maintaining logical encapsulation and code reuse much like layering of meta-actors.

Broadway is an extension to C++ simplifying distributed systems development by providing an actor and meta-actor framework [36]. Since C++ is not a reflective language, reflection is accomplished through use of a library and changing function pointers. Using this scheme, a fully reflective actor system is created enabling redefinition of send, receive, and become methods.

Using Broadway, Sturman created *DIL*, a high-level language that can be used to define protocols as wrappers to actors [37]. These protocol wrappers are applied and enforced through meta-actors to capture and modify incoming and outgoing messages. Similarly, Astley introduced *Distributed Connection*

Language as a method of connecting distributed components through use of meta-actors to transform and route data [4].

9 Discussion

Reflective programming is, although a powerful tool, difficult to use and complex to reason and verify. Part of the reason reflective facilities are so limited in real languages is this complexity, for both the programmer and the runtime developer. Between this complexity and performance issues, reflection has had limited reach.

As mentioned in Section 3.2, most reflective languages limit the amount of reflection offered. Reflective facilities provide great flexibility but at a cost to performance, security, and ease of programming. Because of the necessity for program and metadata to be mutable, reflective runtime systems must be either interpreted or provide much indirection, both of which reduce performance and efficiency [17].

The flexibility and dynamicity provided by reflection also raise security and correctness concerns. Reflection and metaprogramming have been used to provide security and for verification purposes [7,18,2]. However, these start with the assumption that the reflective system itself is secure and correct. Much work remains to be done on verification and security of reflective systems themselves. Some security work has been done on Java's reflective system [11]. However, different security issues are raised by fully reflective systems, given their dynamicity and the possibility of interference between concurrent applications. Correctness and verification of reflective programs are especially important given the difficulty of programming these dynamic systems.

In general, fully reflective programming is useful for research and experimentation to determine what is useful and turn that into simpler programming tools and paradigms (e.g., AOP, TLAM). We believe that such simpler programming paradigms and tools will be essential to address the complexity of coordination in large-scale concurrent and distributed systems. Such systems are of increasing performance as web services and applications, sensor networks, and multicore architectures continue to grow in importance.

Acknowledgements

We would to thank Carolyn Talcott, MyungJoo Ham, Rajesh Kumar, and Sameer Sundresh for their helpful comments on an earlier draft.

References

1. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, MA (1986)
2. Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A.: Abstracting Object Interactions Using Composition Filters. In: Nierstrasz, O. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 152–184. Springer, Heidelberg (1993)

3. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(03), 329–366 (2004)
4. Astley, M. Customization and Composition of Distributed Objects: Policy Management in Distributed Software Architectures. PhD thesis, University of Illinois at Urbana-Champaign (1999)
5. Astley, M., Agha, G.: Customization and composition of distributed objects: Middleware abstractions for policy management. In: Sixth International Symposium on the Foundations of Software Engineering, ACM SIGSOFT (1998)
6. Astley, M., Sturman, D., Agha, G.: Customizable middleware for modular distributed software. Communications of the ACM 44, 99–107 (2001)
7. Bandinelli, S., Fuggetta, A.: Computational reflection in software process modeling: The SLANG approach. In: Proceedings of 15th International Conference on Software Engineering, pp. 144–154 (1993)
8. Bawden, A.: Reification without evaluation. In: LFP '88: Proceedings of the 1988, ACM conference on LISP and functional programming, pp. 342–349 (1988)
9. Brant, J., Foote, B., Johnson, R.E., Roberts, D.: Wrappers to the Rescue. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 396–417. Springer, Heidelberg (1998)
10. Cameron, R., Ito, M.: Grammar-Based Definition of Metaprogramming Systems. ACM Transactions on Programming Languages and Systems 6, 1 (1984)
11. Caromel, D., Vayssiere, J.: Reflections on MOPs, Components, and Java Security. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 256–274. Springer, Heidelberg (2001)
12. Chen, F., Rosu, G.: Java-MOP: A monitoring oriented programming environment for Java. In: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (2005)
13. Cointe, P., Amiot, A., Denier, S.: From (meta) objects to aspects: from Java to AspectJ. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 70–94. Springer, Heidelberg (2005)
14. Czarnecki, K., Eisenecker, U.W.: Generative programming. Springer, Heidelberg (2000)
15. Czarnecki, K., Eisenecker, U.W.: Components and generative programming. In: Proceedings of 7th European software engineering conference and 7th ACM SIGSOFT symposium on Foundations of software engineering, pp. 2–19 (1999)
16. Denker, G., Meseguer, J., Talcott, C.: Rewriting semantics of meta-objects and composable distributed services. Futatsugi [139], 407–427 (1999)
17. Deutsch, L., Schiffman, A.: Efficient implementation of the smalltalk-80 system. In: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 297–302 (1984)
18. Fabre, J.C., Perennou, T.: A metaobject architecture for fault-tolerant distributed systems: the FRIENDS approach. IEEE Transactions on Computers 47(1), 78–95 (1998)
19. Foote, B., Johnson, R.E.: Reflective facilities in Smalltalk-80. ACM SIGPLAN Notices 24(10), 327–335 (1989)
20. Friedman, D., Wand, M.: Reification: Reflection without metaphysics. In: Proceedings of the 1984 ACM Symposium on LISP and functional programming (1984)
21. Frølund, S.: Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. In: Madsen, O.L. (ed.) ECOOP 1992. LNCS, vol. 615, pp. 185–196. Springer, Heidelberg (1992)
22. Frølund, S., Agha, G.: A language framework for multi-object coordination. In: Nierstrasz, O. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 346–360. Springer, Heidelberg (1993)

23. Green, D.: Trail: The Reflection API. In: The Java Tutorial Continued: The Rest of the JDK (TM). Addison-Wesley Pub. Co., Reading (1998)
24. Hennessy, J.: Symbolic Debugging of Optimized Code. ACM Transactions on Programming Languages and Systems (TOPLAS) 4(3), 323–344 (1982)
25. Hutto, P., Ahamad, M.: Slow memory: weakening consistency to enhance concurrency indistributed shared memories. In: Proceedings of 10th International Conference on Distributed Computing Systems, pp. 302–309 (1990)
26. Jang, M.: The Actor Architecture Manual (2004)
27. Kim, W., Agha, G.: Compilation of a highly parallel actor-based language. In: The Fifth International Workshop on Languages and Compilers for Parallel Computing, pp. 1–12 (1992)
28. Lea, R., Yokote, Y., Itoh, J.-I.: Adaptive operating system design using reflection. In: HOTOS '95: Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V), p. 95 (1995)
29. Maes, P.: Computational Reflection. Springer, London (1987)
30. Mason, I.A., Talcott, C.: A semantically sound actor translation. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) ICALP 1997. LNCS, vol. 1256, Springer, Heidelberg (1997)
31. Meseguer, J., Talcott, C.L.: Semantic Models for Distributed Object Reflection. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 1–36. Springer, Heidelberg (2002)
32. Rivard, F.: Smalltalk: a Reflective Language. In: Proceedings of Reflection, pp. 21–38 (1996)
33. Sen, K., Rosu, G., Agha, G.: Runtime safety analysis of multithreaded programs. In: Proceedings of the 9th European software engineering and 11th ACM SIGSOFT symposium on Foundations of software engineering, pp. 337–346 (2003)
34. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient Decentralized Monitoring of Safety in Distributed Systems. In: Proceedings of the 26th International Conference on Software Engineering, pp. 418–427 (2004)
35. Smith, B.C.: Reflection and semantics in LISP. ACM Press, New York (1984)
36. Sturman, D.: Fault-adaptation for systems in unpredictable environments. Master's thesis, University of Illinois at Urbana-Champaign (1994)
37. Sturman, D.: Modular Specification of Interaction Policies in Distributed Computing. PhD thesis, University of Illinois at Urbana-Champaign (1996)
38. Sturman, D., Agha, G.: A protocol description language for customizing failure semantics. In: Proceedings of the 13th Symposium on Reliable Distributed Systems, pp. 148–157 (1994)
39. Talcott, C., Venkatasubramarian, N.: A Semantic Framework for Specifying and Reasoning about Composable Distributed Middleware Services (2001)
40. Venkatasubramanian, N., Talcott, C.: Reasoning about meta level activities in open distributed systems. In: Proceedings of the 14th annual ACM symposium on Principles of distributed computing, pp. 144–152 (1995)
41. Venkatasubramanian, N., Talcott, C., Agha, G.: A formal model for reasoning about adaptive QoS-enabled middleware. ACM Transactions on Software Engineering and Methodology (TOSEM) 13(1), 86–147 (2004)
42. Watanabe, T., Yonezawa, A.: An Actor-Based Metalevel Architecture for Group-Wide Reflection. In: Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages, pp. 405–425 (1990)