A Generic Solution for Agile Run-Time Inspection Middleware

Wouter De Borger, Bert Lagaisse, and Wouter Joosen

Distrinet, Department of Computer Science, KULeuven, Belgium
{wouter.deborger,bert.lagaisse,wouter.joosen}@cs.kuleuven.be

Abstract. Contemporary middleware offers powerful abstractions to construct distributed software systems. However, when inspecting the software at run-time, these abstractions are no longer visible. While inspection, monitoring and management are increasingly important in our always-online world, they are often only possible in terms of the lower-level abstraction of the underlying platform. Due to the complexity of current programming languages and middleware, this low-level information is too complex to handle or understand.

This paper presents a run-time inspection system based on dynamic model transformation capabilities that extends run-time entities with higher-level abstract views, in order to enable inspection in terms of the original and most relevant abstractions. Our solution is lightweight in terms of performance overhead and agile in the sense that it can selectively (and on-demand) generate these high-level views.

Our prototype implementation has been applied to inspect distributed applications using RMI. In this case study, we inspect the distributed RMI system using our integrated overview over the collection of distributed objects that interact using remote method invocation.

1 Introduction

Run-time analysis and run-time inspection of software is required at various stages of the software engineering life cycle: from the early prototyping phases, over the debugging phases that are inherently present when preparing for release, to the deployment phases when the software is exploited in a production environment where profiling and monitoring are important for management purposes.

Many distributed software systems, for example based on the service oriented computing paradigm, or built using web service technology cause major and ineffective efforts to enable dynamic and run-time analysis. In such systems, the operational code is the result of composing and translating many building blocks, developed using different technologies at different abstraction layers. For example, in contemporary distributed systems, applications are represented at different layers of abstractions, ranging from business process management (BPM) support such as the Business Process Execution Language (BPEL [10]) over web services and enterprise component models, to plain object oriented

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 451–470, 2011.
© IFIP International Federation for Information Processing 2011

artifacts and possibly native code. The run-time application typically consists of byte code and data structures that cannot be used to observe higher level abstractions that the BPM developer, system operator or web service integrator might understand. In summary, the run-time representation of such a program is a complex synthetic structure that is not suitable for run-time monitoring by the system operator and that is foreign even to the original developer.

This trend is further evolving due to the versatile modeling and programming languages that can be combined in a single system. Meanwhile platforms and operating environments (cloud based systems, Internet of things etc) tend to become more heterogeneous. The need for run-time inspection and dynamic program analysis increases rapidly. Various stakeholders (developers, operators etc.) should be capable of building dynamic program analysis features that represent the abstraction and concepts that match their understanding of the software.

To enable inspection of such complex composed systems, we present an approach to run-time inspection, based on dynamic model transformation capabilities to extend run-time artifacts with higher-level abstract views, in order to enable inspection in terms of the relevant abstractions. Our solution is lightweight in terms of performance overhead and agile in the sense that it can selectively (and on- demand) generate these high-level views. We combine model transformation and reflective technology to enable a declarative specification of the relation between abstractions. This declarative specification is automatically converted into a mirror-based inspection system that reconstructs representations of higher-level abstractions [1].

The core element of our approach is a generator that is capable of converting the declarative specification of relationships between views on a system (at various abstraction levels) into an actual system that consumes information from lower-level reflective interfaces and implements the higher-level interface. We have developed a prototype implementation of such a system that is capable of translating the relationship between programming models into mirroring systems.

Our generator is validated in a middleware case-study. The generated inspection system automatically collects information from multiple machines, to offer a view on a collection of distributed objects that interact using remote method invocation (RMI hereafter). To enable intuitive inspection, distribution is made transparent, enabling navigation through remote relations with the same ease as local relations. Also distributed stack traces, that span multiple VMs, are represented as if they are local.

The remainder of this paper is structured as follows. The next section elaborates on the problem of inspection of middleware based applications. Section three discusses the requirements for middleware inspection. Section four gives a detailed overview of our solution. In section five, we validate our solution with four inspection cases and evaluate the performance overhead. Section six describes the related work and section seven concludes.

2 Problem Illustration

Application developers and integrators use middleware as development platform and abstraction layer. They use the advanced functionality provided by the middleware, but they are unaware of its inner workings. Middlewares and programming languages offer powerful abstractions, that allow programmers to focus on functionality while making abstraction of technical complexity.

However, when a middleware based software system is deployed, the middleware abstractions are no longer visible. The middleware and the application are composed together into a single synthetic system. The abstractions provided by the middleware are are no longer visible in the run-time structure. When inspecting the run-time structure, the programmer is faced with its full complexity.

When performing detailed inspections, with a monitoring or debugging tool, the developer or operator is faced with information in terms of the language abstraction. The middleware is no longer an abstraction layer, but a complex synthetic structure. The application is no longer represented in terms of middleware abstractions, but entangled in a complex low-level system.

For distributed middleware, the situation is further complicated by distribution. The run-time structure is not only hard to understand, but it is also scattered over different machines. When the middleware is capable of automatic deployment, without human interaction, it is not even known up-front which information is located where. Only the middleware itself knows where the various components have been deployed.

As an example, we will look at remote method invocations in Java. RMI creates a notion of distributed objects and distributed threads. While this is a very basic middleware feature, its run-time structure is already hard to understand without tool support. A distributed object consists of a stub and a proxy. The stub listens on a network port for remote calls. When it receives a remote call, it passes them on to an actual local object. A proxy acts as a local object, but sends all calls it receives over the network to its stub. To support remote invocations, proxy objects are passed on between the different hosts. When a method is invoked on a proxy, the stub on the remote host is contacted and a thread is created on the remote side of the call. This thread receives the request and invokes the correct local method. The caller thread is blocked until the remote call is complete. As such RMI creates many threads, on different machines. What looks like a single thread of execution to a programmer is actually a collection of many different threads on different machines.

In the current state of the art, several tools exist to examine the state of RMI applications. Tools such as JMX for example can provide general overview information, such as the number of threads, memory usage and garbage collector information. A Java debugger can be used for detailed inspection of the individual nodes.

However, there is currently no tool capable of presenting a detailed overview. When an RMI application exhibits undesired behavior, there is no convenient way to inspect it. Current tools don't support distributed objects or distributed

logical threads of execution through the various machines. In practice it would require manually decoding all stack traces on all machines, to find the various local threads that make up the distributed stack trace. Conversely, there is no convenient way to find a stub for a specific proxy.

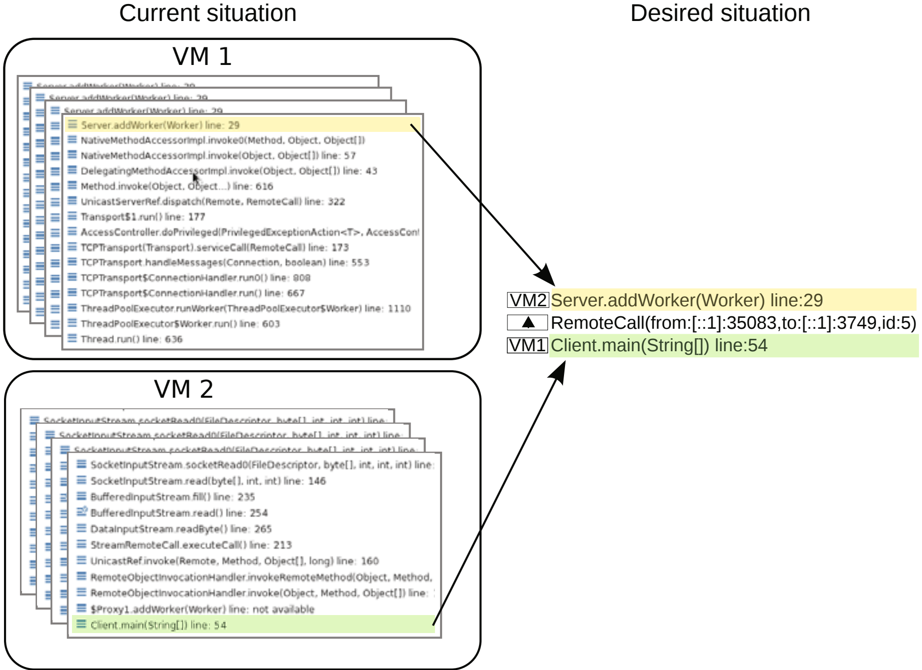

Fig. 1. example of an RMI stack trace

Consider Figure 1. On the right is the conceptual overview: a client performs a remote call to a server. On the left we see the view current tools can offer: on each virtual machine, there is a number of threads. Even when the correct pair of threads is isolated, the view is still polluted with synthetic code. In this case, each of the stack traces contains only one line of actual application code. Furthermore, the information of where the call comes from and where it goes to is not apparent from these synthetic stack traces.

3 Requirements and Approach

When inspecting middleware based applications, the abstraction offered by the middleware should be maintained. For RMI in particular, this means that remote objects and logical distributed threads should be visible.

However, it is also important to acknowledge that openness to inspection is not the core functionality of middleware. Inspection tools should not place a

burden on the normal structure and operations of the middleware. As such, the inspection tools should not overly influence the middleware's core design or operations.

As such we propose three requirements:

1. *Information should be extracted from existing sources.* By reusing existing sources of inspection information, we make sure the middleware's execution structure can be optimized for concerns other then inspection. It enables creation of inspection tools for middlewares that are already in production. No extensive instrumentation or modification is required to support inspection.
2. *What is to be inspected should be specified independent of how it should be inspected.* Middleware experts are not necessarily inspection experts. As such, a middleware expert should be capable of conveying his knowledge about the middleware abstractions without being concerned with the technicalities of distributed inspection. When a middleware expert writes an independent specification of the abstractions, he has the freedom of expressing all facts he knows, without having to think about the efficiency of the resulting inspection tool.
3. *Lazy inspection should be supported.* Inspection should interfere with normal operations as little as possible. Inspection tools should support precise scoping, in which only the required information is extracted. Detailed inspection often focuses on a specific part of the system and explores from there outwards. To support such a restricted focus, the system should support dynamic, on-demand inspection. It should only inspect the part of the system that is requested by the user.

To fulfill these requirements we propose the use of model transformations to build abstract views on top of existing inspection tools. Model transformations enable declarative specification of the relation between the existing inspection interfaces and the desired inspection interfaces in a declarative and natural way. In our solution, these specifications can be automatically converted to middleware components that support dynamic inspection.

As such, our approach for presenting the run-time state leverages on a declarative specification of the transformation that restores the middleware abstractions. This specification is purely declarative and free of technical details about run-time inspection. It describes how the run-time structure, that can be observed through existing inspection interfaces, relates to the conceptual structure, that we want to observe. The declarative specification is automatically converted into an implementation of the high-level reflective interface as a component that consumes a lower-level reflective interface and provides a high-level reflective interface.

Our approach can be divided into four steps.

1. **Modeling:** the existing inspection interfaces and the desired high-level inspection interface are represented by models. For the low-level interface this is usually a trivial conversion of the existing interface into a textual model. Modeling the desired high-level interface requires some design efforts, as they

define which information should be presented to operators or developers. For more information about the design of such interfaces, we refer to Bracha et al [1].
2. **Intermodeling:** the relations between the low-level source models and the high-level target model are specified as a model-to-model transformation. This requires a very precise understanding of the run-time structure of the middleware. However, it requires no special knowledge about reflective systems.
3. **Generation:** the model-to-model transformation is automatically converted into an inspection component that consumes low-level inspection information and produces the higher level information.
4. **Deployment:** the generated component is connected to the actual system.

In the intermodeling phase, a model-to-model transformation language is used to relate the existing structure to the desired structure. Generic model transformation systems exist in the form of rule engines and model-to-model transformers [19,21,4,19,11]. However, both types of systems have no support for lazy execution. These existing systems fundamentally assume that the entire system must be transformed at once. These systems apply an eager strategy, that makes lazy evaluation impossible. Due to the size of the run-time state of software systems, this eager strategy is too slow to support effective reflective transformations. *As such, we base the syntax and semantics of our transformation on the existing QVT-r language[19], but provide an alternate, lazy execution strategy.* We named this QVT-r dialect dynamic QVT-r or QVT-dr.

4 Detailed Solution

This section explains the technical details of our solution. First the declarative description of model-to-model transformations is presented. Then, we describe how such a declarative specification can be converted to an executable form. Finally, we discuss the advantages of this approach.

4.1 Declarative Specification of Model to Model Transformations

In general, a model-to-model transformation expresses the relations between a source model and a target model. The model transformation defines how entities in the source model are related to entities in the target model and vice versa (See Figure 2). For our approach, the source model is an existing inspection system. The source meta-model is the interface of this inspection system. In analogy, the target model is the inspection infrastructure we wish to provide. Its interface is described by the target meta-model. The transformation definition describes the relation between the two interfaces, while the transformation engine is the component we generate, which implements the target model by consuming the source model.

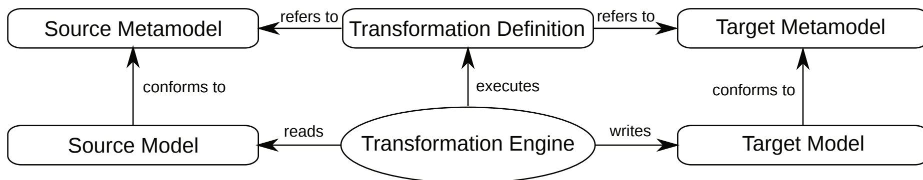

Fig. 2. Overview of model to model transformations. Based on [4]

Meta-modeling. More concretely, for RMI, the *source meta-model* is the Java Debugging Interface, JDI [27,1]. It contains entities such as classes, objects, threads and stack frames. Each of these entities has properties. Classes for example have a name, instances and a reference to their virtual machine. The *source model* is an instance of JDI, that connects to an actual running Java virtual machine (JVM). It provides instances of the types defined in the meta-model, which represent the actual classes and the actual objects present in that JVM. The *target meta-model* is the remote java debugging interface (RJDI), which contains all entities present in JDI, but also all RMI abstractions, such as stubs, proxies and distributed logical threads. Like in the source-meta model, the entities have properties. Proxies for example have an associated stub, instances and a reference to their virtual machine. The *target model* is provided by our generated inspection component: it implements the RJDI interface, based on the JDI interface. The model transformation itself defines how the RJDI interface can be implemented.

Model Transformations. A declarative model transformation describes all relations between elements in the source and target model. Any QVT-(d)r transformation consists of a set of relations, where each relation models the relation between two specific entities. A relation defines the conditions an entity in the source model must fulfill to be transformed into a specific entity in the target model. Listing 1.1 shows the relation between Java (JDI) classes and RMI (RJDI) proxy-types.

To relate both entities, all their properties are bound to a set of shared variables. All variables bound in the target model can be derived from the variables bound in the source model. The relation also defines a set of preconditions to which the source entity must comply. These preconditions are demarcated with the keyword `when`. When an entity of the correct type is found in the source model for which all preconditions hold, we say the relation holds.

If the relation holds, the target model must contain the target entity. Furthermore, the relation can also define a set of post-conditions (demarcated with the keyword `where`). If the relation holds, all post-conditions must hold.

As such, Listing 1.1 defines a single relation, relating two entities. It states that if an entity of type `ClassType` exists in Java (JDI), and it has `$Proxy` in its name and its superclass has as name `java.lang.reflect.Proxy`, then this entity corresponds to an `RmiProxyType` in RMI (RJDI), which belongs to the

```
1   relation ObjectReferenceTypeToProxyType{
2     domain JDI in:ClassType{
3      instances = instI;
4     };
5     domain RJDI out:RmiProxyType{
6      instances = instO;
7      //other properties omitted
8     };
9     when{
10      VmtoVm(in.virtualMachine,out.virtualMachine);
11      in.name.contains("$Proxy");
12      in.superclass.name = "java.lang.reflect.Proxy";
13      //more complex preconditions omitted
14    }
15    where{
16      ObjectReferenceToProxy(instI,instO);
17      //additional post-conditions omitted
18    }
19  }
```

Listing 1.1. Concrete example of a model transformation relating RMI-proxies to their Java equivalents

corresponding virtual machine. The instances of the `RmiProxyType` can be derived through the `ObjectReferenceToProxy` relation.

In this example, we omitted the more complex pre- and post-conditions that are used to extract more information from the middleware, such as how to find the stub associated with this proxy. These parts of the pattern are analogous to what is already presented, but require a more intimate knowledge of the internals of RMI.

Using this approach, a description of the run-time structure of the most important RMI concepts is created. Stubs and proxies can be found based on these patterns both on the heap and on the stack. They are transformed into a representation that hides their internal complexity but exposes their internal state.

4.2 Dynamic Execution of Declarative Model Transformations

Such declarative model transformations are not directly executable. The next section describes how the declarative specifications can be converted to an executable form. It describes the internal mechanism of the generator that transforms the declarative specification to an inspection component that dynamically and lazily transforms low-level information.

The model transformation defines how information flows between the source and target model. As such, the main task of the generator is to infer an implementation for all operations of the target model, based on the operations in the source model, in such a way that lazy evaluation is supported. The generator is based on a four step process.

1. **Parsing.** The model transformation definition and related meta-models are compiled into a data-flow graph and type-checked. The data-flow graph directly represents all relations defined in the transformation definition.
2. **Inference of the control flow.** Directions are added to the flow of information. Given the fact that the input node is known and the characteristics of all other nodes, a sat solver is used to compute all valid flows of information through the model. If multiple alternatives exist, a heuristic is used to choose the most optimal.
3. **Model partitioning.** The graph is partitioned into three parts: a part containing all preconditions, a part containing all postconditions and a part containing the rest.
4. **Code generation.** Based on the partitioned information flow graph, code is generated.

The remainder of this section provides more information about these steps.

Parsing and Checking. The declarative model transformation definition is parsed into an abstract semantic graph (ASG). The ASG is a typed and labeled graph (e.g. Figure 3), representing the structure of the source meta-model, target meta-model and the transformation between them. First we discuss the general structure of the ASG and then illustrate it based on the ASG segment in Figure 3.

In general, for each relation (such as the one in Listing 1.1), each entity presented in it becomes a pattern node in the ASG. Each pattern node is also bound to a node in the source or the target meta model that indicates its type. All relations between pattern nodes become edges in the ASG. The type of the edge indicates the type of the relation. Entities bound to a variable have their pattern node bound to a variable node.

For example, consider figure 3. It represents the following part of Listing 1.1:

```
in:ClassType{virtualMachine=temp1,instances=instances}.
```

The variable `temp1` is implicit in Listing 1.1. In this pattern, the variable `in` is bound to a pattern node of type `ClassType`. This node is indicated with a bold border. The type `ClassType` has three operations: `virtualMachine`, `instances` and `name`, with as types respectively `VirtualMachine`, `ObjectReference` and `String`. The pattern binds the operations `virtualMachine` and `instances` to pattern nodes that are bound to the named variables `temp1` and `instances`.

Inference of the Control Flow. To support the generation of an implementation, the direction of the information flow through the pattern must be inferred. When considering individual statements, information can flow in either direction. For example: in Listing 1.1 line 3 and 6 are identical statements but information flows through them in the opposite direction. Line 3 assigns the value of `in.instances` to the variable `instI` while line 6 provides a result for the operation `out.instances` through the value of `instO`.

As such, the information flowing through any node depends on the flow through any other node in the same relation. To derive a valid information

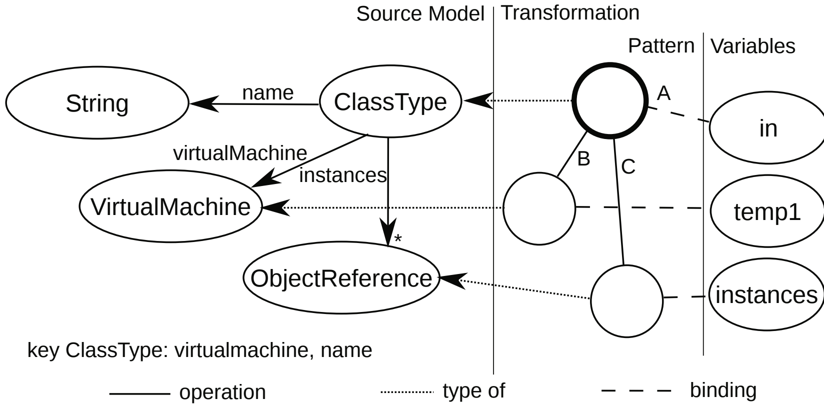

Fig. 3. Part of the ASG

flow, the flow analysis adds a direction to each edge, indicating the flow of information. The analysis is based on a model that defines when a node in the pattern has sufficient information flowing in to calculate all other edges.

While information flow is best explained in terms of graphs, it is more convenient to use a SAT solver to efficiently derive an optimal information flow. Therefore the ASG model is translated into logic predicates. Each edge becomes a boolean variable indicating the flow of information. Each node becomes a list of predicates, defining when the node has sufficient incoming edges to calculate all other edges.

For example, take the pattern from Figure 3. The bold node is defined if either 1) information is flowing in from the variable `in` (i.e. the entity is defined elsewhere) or 2) all it's operations (`virtualMachine`, `name` and `instances`) are defined (i.e. a complete definition of the entity is present and it can be constructed here). However, in this case the operation `name` is not bound in the pattern. As such the overall logical predicate becomes $(false \wedge B \wedge C) \vee A$, when we assume *true* means *the edge is incoming*. As a consequence A must be true and the corresponding edge must be incoming. The bold node thus receives information from the variable `in`.

For this simple example pattern, there is only one possible information flow (A is incoming). However, in general, each node can have multiple solutions. This require a more global analysis process, taking into account all nodes in the relation. By converting all nodes and edges into predicates, a SAT solver can be used to derive all valid information flows through the relation. A search heuristic can then be used to select one solution.

Model Partitioning. In the information flow graph, some nodes in the pattern have more incoming edges then strictly required. We call these nodes overconstrained. For example, on line 11, `in.superclass.name` is defined because the variable `in` is defined. It is also defined because it is bound to a constant. When information flow is overconstrained, pattern matching may fail on that node. i.e.

If both sources of information produce a different value, the relation doesn't hold. Any overconstrained node forms a condition that must be checked to determine if the relation holds or not.

For code generation, overconstrained nodes must be identified. Therefore, the ASG is partitioned. First it is divided into conditional nodes and non-conditional nodes. Conditional nodes are either overconstrained or they directly depend on an operation that may fail, such as a call to another relation. The conditional nodes are then partitioned again into nodes that have been marked as assertions and others.

In this way, there are three partitions: the guard nodes (conditions that are not assertions), the assertion nodes and the non-conditional nodes. The guard nodes are all the nodes that must be checked to see if the pattern matches. These nodes (and all nodes providing information to them) must always be evaluated eagerly (and thus not lazy).

The assertion nodes can be discarded, as the assertion should always hold. However, they can also be passed to the code generator, to produce more robust code, that checks all (or some) assertions.

Code Generation. In the implementation (see Listing 1.2) each relation becomes a method, that takes as an argument a source-model entity. When the method is called, all conditions in the guard partition are checked. If all conditions hold, the relation holds. Then an object of the desired target type is constructed. Each method of this object corresponds to an operation in the target model. Each operation contains the part of the pattern that provides the operation according to the inferred control flow. In practice, most operations use other patterns to create other target-model entities. As such, when a first target-model entity has been created, the rest of the target model can be explored using its operations. Each operation will lazily collect entities from the underlying inspection interface, as required by the patterns.

4.3 Discussion

Our approach enables automatic generation of an inspection component out of a declarative specification. Apart from the earlier mentioned requirements, it has two important advantages:

1. Overdetermined specifications don't result in a slower system. When a model transformation defines many ways of deriving any given operation, this doesn't make the execution of the pattern less efficient. The generator can choose any sufficient implementation, while discarding redundant information. At the other hand, the generator can also be configured to check all assertions. As such, the generator can create either more robust or more efficient code, without any manual rewriting.
2. Different tools can reuse the same transformation. As in any compiler, the use of a central intermediate representation decouples three roles: language user, optimization writer and back-end developer. The important consequence is

```
rmi.RmiProxyType objectReferenceTypeToProxyType(jdi.ClassType in){
 rmi.VirtualMachine temp1 = vmToVm(in.virtualMachine);
 if(temp1 == null) return null;
 if(!in.name().contains("$Proxy")) return null;
 if(!in.superclass().name().equals("java.lang.reflect.Proxy"))
  return null;
 //more complex guards omitted
 return new ObjectReferenceTypeToProxyType(in,temp1);
}
class ObjectReferenceTypeToProxyType implements rmi.RmiProxyType{
 private jdi.ClassType in;
 private rmi.VirtualMachine virtualMachine;
 private List<rmi.ObjectReference > instances;

 ObjectReferenceTypeToProxyType(jdi.ClassType in,
   rmi.VirtualMachine virtualMachine){
       this.in = in;
       this.virtualMachine = virtualMachine;
 }

 public rmi.VirtualMachine virtualMachine(){
  return virtualMachine;
 }

 public List<rmi.ObjectReference > instances(){
  if(instances != null)
    return instances;
  instances = ObjectReferenceTypeToProxyType(in.instances());
  return instances;
 }
 //other properties omitted
}
```

Listing 1.2. Implementation of the pattern in Listing 1.1 without assertions

that replacing the back-end stages yields a different type of inspection infrastructure. Inspection can be used in-program (reflective), but also for debugging or even post-mortem debugging (debugging of systems that have already crashed). Each of these styles requires a very different tool, but suffers from the same abstraction gap. If tools are required to support N languages and M styles, it is no longer necessary to build M*N tools, but N specifications and M back-ends. By decoupling the back-end and front-end the complexity of the problem has been reduced from multiplicative to additive. Furthermore, as any component is reused more often, it will mature faster.

5 Evaluation

To validate the use of such a high level inspection tool, we use an example application and compare inspecting it with existing tools against our solution. First, a number of use cases are discussed, then we evaluate the performance overhead of our solution.

The application is a work scheduling server. Jobs, consisting of several tasks are queued on the server. The server then schedules the tasks on worker nodes. The server also passes a callback remote object to the worker, by which the worker can report its progress. When a worker node has completed its task, it

signals the server through the callback. The server then schedules the next task in the job on a worker.

On the application, we test four inspection scenarios.

1. **A control flow problem:** when a worker signals a task is done, the next task is scheduled from within that thread. This causes all tasks in the job to be in the same logical thread of execution. The logical thread starts from the server, then goes to the first worker node and then back to the server, then back to the next worker and so on. This means that each job consumes 2n+1 threads, with n the number of tasks in the job. This has two side effects: the server and worker consume a massive amount of threads and network sockets and, when one worker node fails, all jobs that used this worker before crash on completion.
2. **An information flow problem:** when a worker receives a monitor callback, it exports the callback. When a job is done, it returns the callback to the server. The server then hands this callback to the next worker. Conceptually, this is the same callback-reference. However, because the callback has been exported on the worker, all calls to the remote object are routed over the worker node. This makes message propagation slow and very sensitive to failure of worker nodes.
3. **A deployment problem:** a worker node has been deployed to a wrong host. This causes two worker nodes to share the same virtual machine. At application level the workers look different. RMI makes abstraction of distribution, so the server doesn't know they are on the same host.
4. **The cost of inspection:** different work scheduling servers are working for different organizations. They share the same infrastructure, but for security reasons, different servers should never share a worker node. Regular audits must ensure this.

The control flow problem has no apparent symptoms, until a worker node is taken out of the schedule and powered down. After some time, jobs start to fail unexpectedly. When connecting a local debugger to the server and browsing through all threads present, we find that many threads have a similar structure. On closer examination, we find that they share the segment depicted in Listing 1.3. On the worker nodes, a similar stack trace is found (Listing 1.4). A very experienced RMI developer may conclude from this information that the control flow is going back and forth between client and server. This conclusion is however far from obvious.

When using our generated inspection tool we can investigate the distributed logical stack trace from one of the jobs (Listing 1.5). This immediately shows that the distributed stack trace spans many different nodes. The individual frames can be inspected to trace this control flow and learn to understand what causes this behavior.

Similar to the first problem, the second problem has no apparent symptoms, until a worker node is taken out of service. However, in this case the local stack trace provide no clues. In the previous example, the long stack traces had a long lifetime. This made sure many abnormal stack traces were present on any

```
... (8 more frames)
RemoteObjectInvocationHandler.invoke()
$Proxy1.run()
MonitorImpl.runOn()
MonitorImpl.done()
GeneratedMethodAccessor5.invoke()
DelegatingMethodAccessorImpl.invoke()
 Method.invoke()
UnicastServerRef.dispatch()
... (13 more frames)
```

Listing 1.3. Server side stack trace caused by the control flow problem

```
... (8 more frames)
RemoteObjectInvocationHandler.invoke()
$Proxy2.done()
Client.run()
NativeMethodAccessorImpl.invoke0()
... (13 more frames)
```

Listing 1.4. Client side stack trace caused by the control flow problem

machine, making it easy to find them. However, in this case they are very short lived. When inspecting any local node, one is unlikely to find any abnormal stack traces. Also, these stack traces contain no application code. Placing break points in the application doesn't help in finding abnormal stack traces. Without appropriate tools, the only possibility of finding the root cause is source code analysis.

With our tool, when plotting all remote objects present in the system, it is immediately clear that most worker nodes are not directly referring to the server, but referring to other worker nodes (Figure 4). This can only be caused by a reexport of the remote reference on the worker nodes.

```
vm3 Client.run()
vm3 STUB 192.168.150.32:38175 57769 -7666749065146411512
vm1 PROXY 192.168.150.32:38175 57769 -7666749065146411512
vm1 MonitorImpl.runOn()
vm1 MonitorImpl.done()
vm1 STUB 192.168.150.31:39728 36651 -2921635400086109349
vm2 PROXY 192.168.150.33:39728 36651 -2921635400086109349
vm2 Client.run()
vm2 STUB 192.168.150.33:60307 52965 -2670173772187310440
vm1 PROXY 192.168.150.33:60307 52965 -2670173772187310440
vm1 MonitorImpl.runOn()
vm1 MonitorImpl.run()
vm1 MonitorImpl.run()
```

Listing 1.5. Stack trace of failing application

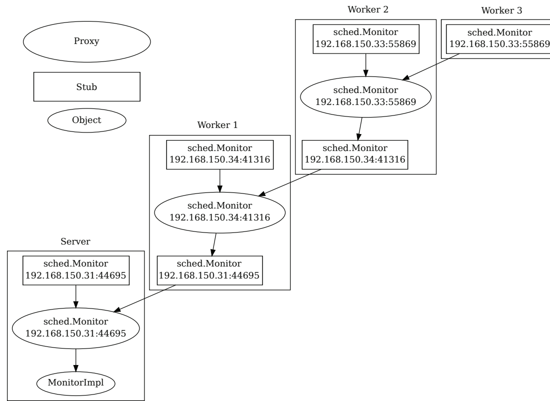

Fig. 4. Remote objects with information flow problem

The third bug has no immediately apparent symptoms. When looking at the load on the different hosts, it will be clear that one host is less heavily loaded. When looking at the performance of all hosts, two hosts will be underperforming. Putting these two facts together one may conclude that a node has been deployed on the wrong host.

When plotting the remote objects with our inspection component (Figure 5), it is immediately clear the server references two workers on the same host. It is also clear on which host the nodes are deployed.

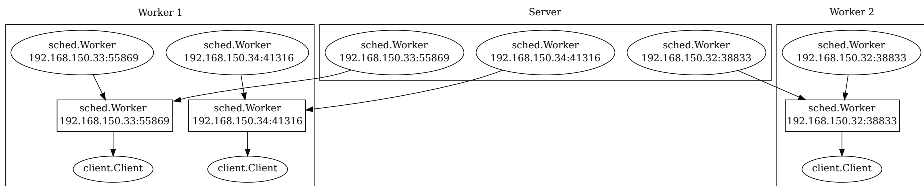

Fig. 5. Remote objects with deployment problem

The fourth problem is not a bug as such. It is an auditing requirement. When using ordinary inspection techniques, it is not cost efficient for an auditor to regularly dump the state of the application and start digging through in the hope of finding an irregularity in the communications pattern. Currently, two alternate solutions exist. The first solution would be to physically separate both infrastructures. This is more expensive, due to a lack of resource sharing. Alternatively, the server could be adapted to maintain an explicit list of hosts on which its workers are located. However, as RMI is location transparent, this would preclude the use of RMI. When using our inspection component, the physical placement and relation between hosts can be audited easily, as shown in the previous example.

To evaluate the performance overhead, we set up the following worst case scenario: worker nodes with a CPU intensive workload were subjected to continuous monitoring. Our set up consisted of one server with three worker nodes – all Pentium III single core machines. The inspection server continuously searched for all proxies and stubs on all machines and retrieved all their attributes. Once all information was retrieved, the cache was cleared and the search restarted. With all nodes executing CPU intensive tasks, the performance overhead was 34% compared to the case with no inspection. When the worker nodes were given less CPU intensive tasks, the performance overhead dropped under 1%.

6 Related Work

This work is founded in the reflection and model driven development (MDD) communities. The related work around MDD has already been briefly highlighted in Section 4. In this section, we highlight some influential work from the field of *reflection* and compare our approach to existing inspection approaches for *monitoring*, *debugging* and *reverse engineering.*

Reflection. Reflection is the ability of software systems to reason about and act on themselves. It encompasses inspection, but also self-modification and meta-programming. Our approach is inspired on the design principles for reflective systems, defined by Bracha et al [1]. These principles define that a reflective system should have ontological correspondence to the system it reflects on and that reflective systems should encapsulate their implementation.

Ontological correspondence means that the reflective interface should be structured according to the abstractions of the system it reflects on. This is also one of our key requirements for middleware inspection: all the middleware abstractions should be maintained when inspecting distributed software systems.

We also choose for strong encapsulation of the implementation. Our approach requires no modification of the underlying middleware and puts no constraints on the middleware's execution structure. The implementation of the inspection component is the model transformation, which is declarative and completely separated from the middleware. However, if the middleware has no interface or fixed internal structure, the inspection component may break when the middleware evolves. As such our approach doesn't require a stable interface, but it is more stable when a stable interface exists.

Reflective middleware systems [14,31,3] have the capability of reflecting on the middleware structure itself. This reflection goes beyond inspection and supports run-time adaptation of the middleware. In such middleware, the reflective infrastructure is always present. This adds a constant overhead to the execution. Reflective middleware systems can serve as sources of information for our approach. They can be used in the way we used JDI in this paper.

Monitoring. Monitoring systems keep track of a limited set of inspection targets over a long period of time. Monitoring consists of two main activities: information extraction and information aggregation.

The most common way of extracting information is *built-in monitoring*. The system is modified by hand to emit events that signal important changes [26]. The advantage of this approach is its simplicity, while the disadvantage is that the monitoring system always incurs an overhead – as it is part of the system. It can not evolve independently or be adapted at run-time. As such, this approach is used to expose small volumes of high-level information. When the middleware actively supports the emitting of events, such as in Google's Dapper [24], the event streams of different hosts can be put together to create a distributed trace. The information extracted from such monitoring probes can also be aggregated. For statistical aggregation of monitoring data, collection systems are already widely deployed [7,34].

A second way of extracting monitoring information is *instrumentation*. Instrumentation systems automatically modify a program so that it emits events. This enables dynamic fine-tuning of the monitoring and its associated overhead. However, dynamic deployment of monitoring probes is a technically complex operation, that requires support of the underlying platform. For many languages instrumentation support exists [8,18,20].

An advanced proponent of the instrumentation approach is described in [17]. This monitoring system dynamically instruments code ahead of the flow of control. It has properties comparable to our approach. Monitoring components are developed separately and deployed on demand. Also the performance overhead seems to be comparable. The major difference is that our system only supports structural inspection while Mirgorodskiy et al. only support event based inspection. As such, their system is capable of tracing change very efficiently, but incapable of inspecting state that doesn't change. Our system has the inverse properties: it can inspect existing state in great detail, but is incapable of perceiving rapid change. However, in the future we aim to integrate events into our model transformation approach.

Persistent query systems are another possible form of information aggregation. A query system is a reflective component that allows external systems to query its own state. A *persistent* query system is a query system capable of keeping the results of its queries up-to-date when the underlying system changes. It provides a form of continuous reporting [22,13,29].

Debugging. Debugging means searching for and remedying of software faults [35]. Interactive inspection is an important component of debugging, but not the only one. It also encompasses methodology, tools for automatic and semiautomatic detection, prevention and removal of faults as well as edit-and-continue technology. [23,35,8,33,32,12,25]

The current generation of inspection tools for debugging consists of two categories: debuggers for languages with a custom VM and debuggers for compiled languages. Debuggers for languages with a custom VM or languages with a strong reflective system provide debugging facilities by exposing their internal data structures. This provides a view of the running program in terms of the abstractions supported by the VM. Without need for transformations, the VM natively supports all abstractions.

For compiled languages, transformations are always required. The state-of-practice for such languages is to write the required transformations by hand. This lack of a disciplined approach – combined with the inherent complexity of pattern matching code – limits the capabilities of current debuggers. GDB [9], for example, is still unable to decode the heap of C programs. However, recently, efforts are being made to isolate the pattern matching into separate modules, to enable heap decoding [15].

For middleware few debuggers exist. One notable exception is a distributed debugger constructed by Mega and Kon [16]. It offers support for distributed logic threads, similar to our approach. As most debuggers, its transformation components have been built by hand, supporting both structural and event based inspection.

For a more elaborate explanation about the design trade-offs for the construction of higher level debuggers, we refer to [16,5].

Reverse Engineering. Reverse engineering (RE) tools enable dynamic rebuilding of software abstractions. RE tools rely on advanced visualization [6] and combined static and dynamic analysis [28,30,2]. From a modeling perspective, reverse engineering mostly uses containment relations. For example, instructions are grouped into blocks, blocks into methods, methods into classes. Currently, most RE tools use an event based approach for dynamic analysis. However, our generator makes state transformation components easier to build and may enable RE systems to incorporate them.

7 Conclusion

We presented an approach enabling inspection of middleware with full support for all abstractions offered by the middleware, that requires no modification of the middleware itself and is capable of limiting its overhead by dynamic, on-demand transformation.

Our generator technology can be used to construct detailed inspection systems for middleware. It decouples the role of middleware expert and inspection expert, to support modular development of inspection tools. Middleware experts can express their knowledge in a declarative way. This declarative specification is automatically converted into a usable and efficient inspection component. The generator is capable of automatically removing redundant information from the specification and can switch between generating either more robust or more efficient inspection components.

We have demonstrated the advantage of maintaining the middleware abstraction when inspecting in four use cases. We also showed that the overhead of the inspection system is acceptable, even in a worst case scenario.

In the future, we will focus on automatic generation of more complex inspection tools. We aim to add support for events in our generic inspection approach to detect run-time changes and act upon them. We will also integrate our approach into an IDE to enable broader, user-driven evaluation. Further validation and

evaluation of our approach in a broader monitoring and run-time management context will also provide more metrics about the effectiveness of lazy execution and the use of overdetermined specifications.

References

1. Bracha, G., Ungar, D.: Mirrors: design principles for meta-level facilities of object-oriented programming languages. In: Proc of OOPSLA, pp. 331–344 (2004)
2. Chan, A., Holmes, R., Murphy, G.C., Ying, A.T.T.: Scaling an object-oriented system execution visualizer through sampling. In: 11th IEEE Intl Workshop on Program Comprehension, pp. 237–244 (2003)
3. Clarke, M., Blair, G.S., Coulson, G., Parlavantzas, N.: An efficient component model for the construction of adaptive middleware. In: Proc of IFIP/ACM Intl Conf on Distributed Systems Platforms Heidelberg, pp. 160–178 (2001)
4. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–645 (2006)
5. De Borger, W., Lagaisse, B., Joosen, W.: A generic and reflective debugging architecture to support runtime visibility and traceability of aspects. In: Proc of AOSD, pp. 173–184 (2009)
6. De Pauw, W., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J., Yang, J.: Visualizing the Execution of Java Programs. In: Diehl, S. (ed.) Dagstuhl Seminar 2001. LNCS, vol. 2269, pp. 151–162. Springer, Heidelberg (2002)
7. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Transactions on software Engineering 30(12), 859–872 (2004)
8. Eaddy, M., Aho, A., Hu, W., McDonald, P., Burger, J.: Debugging Aspect-Enabled Programs. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 200–215. Springer, Heidelberg (2007)
9. Free Software Foundation, Inc. Gdb: The gnu project debugger (July 2009), `http://www.gnu.org/software/gdb/`
10. Jordan, D., and Evdemon, J. Web services business process execution language version 2.0 (April 2007), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
11. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
12. Ko, A.J., Myers, B.A.: Designing the whyline: a debugging interface for asking questions about program behavior. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 151–158 (2004)
13. Ko, S.Y., Yalagandula, P., Gupta, I., Talwar, V., Milojicic, D., Iyer, S.: Moara: Flexible and Scalable Group-Based Querying System. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 408–428. Springer, Heidelberg (2008)
14. Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., Pasquale, F.: Monitoring, Security, and Dynamic Configuration with the *dynamicTAO* Reflective ORB. In: Coulson, G., Sventek, J. (eds.) Middleware 2000. LNCS, vol. 1795, pp. 121–143. Springer, Heidelberg (2000)
15. Malcolm, D.: gdb-heap, `https://fedorahosted.org/gdb-heap/` (access: Febuary 17, 2011)
16. Mega, G., Kon, F.: An Eclipse-Based Tool for Symbolic Debugging of Distributed Object Systems. In: Meersman, R. (ed.) OTM 2007, Part I. LNCS, vol. 4803, pp. 648–666. Springer, Heidelberg (2007)

17. Miller, B.P., Mirgorodskiy, A.V.: Diagnosing Distributed Systems With Self-Propelled Instrumentation. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 82–103. Springer, Heidelberg (2008)
18. Navarro, L.D.B., Douence, R., Südholt, M.: Debugging and Testing Middleware With Aspect-Based Control-Flow and Causal Patterns. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 183–202. Springer, Heidelberg (2008)
19. OMG. Meta object facility (mof) 2.0 query/view/transformation, http://www.omg.org/spec/QVT/1.0/
20. Oracle. Java instrumentation, http://download.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html (access May, 2011)
21. Proctor, M. Drools documentation library (August 2009), http://www.jboss.org/drools/documentation.html
22. Rajamani, V., Julien, C., Payton, J., Roman, G.C.: Paq: Persistent Adaptive Query Middleware for Dynamic Environments. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 226–246. Springer, Heidelberg (2009)
23. Rosenberg, J.B.: How debuggers work: algorithms, data structures, and architecture. John Wiley & Sons, Inc., New York (1996)
24. Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a large-scale distributed systems tracing infrastructure. In: Google Research (2010)
25. Silva, J.: A Comparative Study of Algorithmic Debugging Strategies. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 143–159. Springer, Heidelberg (2007)
26. Sun Microsystems, I. Java management extensions (August 2009), http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/
27. Sun Microsystems, I. Java platform debugger architecture (June 2009), http://java.sun.com/javase/technologies/core/toolsapis/jpda/
28. Systä, T., Koskimies, K., Müller, H.: Shimba—an environment for reverse engineering java software systems. Software: Practice and Experience 31(4), 371–394 (2001)
29. Van Renesse, R., Birman, K.P., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Transactions on Computer Systems (TOCS) 21(2), 164–206 (2003)
30. Walker, R.J., Murphy, G.C., Freeman-Benson, B., Wright, D., Swanson, D., Isaak, J.: Visualizing dynamic software system information through high-level models. ACM SIGPLAN Notices 33(10), 271–283 (1998)
31. Wangham, M.S., Lung, L.C., Westphall, C.M., Fraga, J.S.: Integrating ssl to the jacoweb security framework: project and implementation. In: Proc of IEEE/IFIP Intl. Symp. on Integrated Network Management, pp. 779–792 (2001)
32. Weiser, M.: Program slicing. In: ICSE 1981, pp. 439–449 (1981)
33. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. SIGSOFT Softw. Eng. Notes 30(2), 1–36 (2005)
34. Zanikolas, S., Sakellariou, R.: A taxonomy of grid monitoring systems. Future Gener. Comput. Syst. 21(1), 163–188 (2005)
35. Zeller, A.: Why programs fail: a guide to systematic debugging. Morgan Kaufmann (2009)