Dynamic Aspects for Runtime Fault Determination and Recovery

Jeremy Manson, Jan Vitek, Suresh Jagannathan
Department of Computer Science
Purdue University
{jmanson,jv,suresh}@cs.purdue.edu

ABSTRACT

One of the most promising applications of Aspect Oriented Programming (AOP) is the area of fault tolerance and recovery. In traditional programming languages, error handling code must be closely interwoven with program logic. AOP allows the programmer to take a more modular approach - error handling code can be woven into the code by expressing it as an aspect.

One major impediment to handling error code in this way is that while errors are a dynamic, runtime property, most research on AOP has focused on static properties. In this paper, we propose a method for handling a variety of run-time faults as dynamic aspects. First, we separate fault handling into two different notions: *fault determination*, or the discovery of faults within a program, and and *fault recovery*, or the logic used to recover from a fault.

Our position is that fault determination can be expressed effectively as dynamic aspects. We propose a system, called Rescue, that exposes underlying features of the virtual machine in order to express faults as variety of run-time constraints. We show how our methodology can be used to address several of the flaws in state of the art fault fault handling techniques. This includes their limitations in handling parallel and distributed faults, their obfuscated nature and their overly simplistic notion of what a "fault" actually may comprise.

1. INTRODUCTION

It is extremely difficult to write code that handles faults efficiently and effectively. They are a reasonably high percentage of source code generated: one study indicates that up to 5% of program text is contained in fault handling code, and that anywhere up to 46% of a given program is reachable from that code [9].

We take an even broader view of faults: instead of saying a fault is a situation where something drastic has occurred, we simply define it as a situation that arises because of unexpected change in resource availability or performance assumptions. This notion encompasses the traditional notion of program/node failures, as well as quality of service and other non-functional characteristics of distributed program behavior. For example, under this model, excessively high load on a processor might be characterized as a fault.

The current state of the art mechanism for dealing with faults in software treats them as *exceptions* which can be *thrown* by a line of code, and then *caught* by an exception handler whose dynamic scope encloses the code that threw the exception. Some form of exception handling is available in most modern languages, including Java, C++, C#, Visual Basic, Ada, ML and Haskell. Their use in this context stretches back to the CLU programming language [4]. However, exceptions have a number of limitations.

The first problem that we encounter with the use of language-level exceptions is that they tend to foster a binary view of faults: either the program is out of memory (for example), or it is not. This does not allow for graceful resolution of resource exhaustion – if we want to perform some emergency cleanup task when a certain percentage of memory is available, we are out of luck.

Exceptions also obfuscate code greatly. One or two thrown exceptions are simple to catch, and the resulting code is simple to read. However, in large systems, with calls to many different libraries, code can throw many different exceptions. This leads to a software engineering challenge: there will be many clauses in the program that catch these exceptions, all of which are interlaced with program logic that has nothing to do with fault handling and tolerance.

This problem is exacerbated when dealing with faults; faults incorporate unexpected as well as erroneous conditions. Since fault criteria can be arbitrarily complex, using standard exception handling mechanisms to express more general faults leads to further complexity.

It is also unclear how exceptions raised on one segment of a distributed application can be handled cleanly in another. If, for example, code running on one node fails, an exception is propagated up the stack for that node, but not necessarily propagated to other nodes which require the information.

Our conclusion is that is desirable to have a way *to separate*

the definition of an application-specific fault from the mechanics of fault book-keeping. Once a fault is identified, the infrastructure should provide *facilities for orderly recovery and compensation.*

This paper discusses a novel programming language based approach to determining the presence of faults, called RESCUE. The language is designed to be implemented on both stand-alone and distributed systems; it is an extension to AspectJ [3], the most widely used AOP language for Java. RESCUE would provide a wide spectrum of ways for users to define mechanisms for determining when faults occur in their programs. Once they have declared how the faults occur, they can either define their own compensatory code, or use one of a number of built-in mechanisms (such as transactions, or distributed replication). The code they define may change behavior across the entire infrastructure; thus, the resulting system would provide a pervasive system for defining and handling faults in a clear, clean and efficient way.

2. FAULT DETERMINATION IN RESCUE

In distributed systems, just as in a parallel program, there is a large set of events that need to be monitored in order to detect "exceptional" situations. RESCUE provides infrastructural support for *fault determination*, the act of recognizing such conditions. An informal taxonomy of the faults RESCUE addresses is given below:

- *Internal v. External.* An external event (for example, memory exhaustion or a network outage) exists outside of the control of the program and can be monitored either within the virtual machine or in middleware. An internal event corresponds to a fault in the application: for example, when a thread raises an exception as a result of a condition that compromises the integrity of other concurrently executing tasks.
- *Fine- v. Coarse-Grained.* Some events, such as a measurement of CPU usage or a hardware interrupt, have high frequencies. Other events, such as node failure or file operations, are less frequent. The granularity impacts the performance of monitoring.
- *Builtin v. User-defined.* Builtin events include events supported by the VM, like CPU or memory usage. User-defined events may include something as simple as a counter incremented every time an element is added to an array; a fault might occur if the count gets too high.
- *Local v. Distributed.* An example of a local event would include a change in CPU usage on a given node. An example of a distributed event would be a failure on a different node.

In order to support the full taxonomy listed above, RESCUE requires functionality that goes well beyond the current state-of-the-art. Faults can be complex to express, and the logic to keep track of them may not be easily localized. In the case of fine-grained event monitoring, such as timers or hardware interrupts, it is essential to guarantee that the overhead of fault determination can be bounded.

2.1 Meters and Plans

RESCUE includes a declarative language for fault determination; it is integrated into Java and designed to allow for an efficient implementation. The two key concepts in the language are *plans* and *meters*. Plans are specifications of fault conditions as a predicate over meters. Meters provide an interface to monitoring events. Typically, plans are written by an application developer, as they describe the condition which requires intervention. Meters, on the other hand, are designed to be provided by the infrastructure: i.e. either the virtual machine or libraries. User-defined meters can also be defined using a programmatic API.

2.1.1 Meters

Meters provide the flexibility of join points for faults. A meter is attached to a resource, and can be used to tell the user how much of that resource remains. For example, a meter may give information as to how much memory remains, how many threads are active in a system, or how much network bandwidth is available.

```
plan NetworkUnavailable:
  meter bandwidth < 128kbps or
  meter latency < 100ms;
```

Built in memory meters can also be used to determine that an application is running out of memory. For example, the following plan can be used to trigger a memory fault:

```
plan MemoryFault:
  meter memory < 100k;
```

The obvious alternative to this approach would be to have application code poll for memory at regular intervals. Doing this in an automated way is clearly not very efficient in the case where many applications are running on the same host, trying to monitor the same fine-grained fault. Meters, on the other hand, have compiler support and are registered with the environment. So, for instance, in the case of memory, the virtual machine could insert checks in the memory allocator (and try to optimize these checks).

Additionally, meters may be defined in library code. A meter in an I/O class may, for example, tell the user how much space is left in a buffer, so that it can avoid buffer overflow/underflow. In this case, the programmer explicitly controls the frequency of checks. Obviously, this would also be a case where more traditional aspect-oriented approaches might suffice.

2.1.2 Plans

Once a meter is in place, it can be used to indicate the presence of a fault by placing it in a *plan*, possibly aggregated with other meters. In the following example, the plan is triggered when network bandwidth is low or the latency (perhaps of the last remote request) is high.

```
after(): within(Task) && NetworkUnavailable {
  spawnTasksOnLocalHost = true;
}
```

The above code deals with the spawning of tasks on a distributed platform. It specifies that when the control flow is in the Task class and the network becomes unavailable, the task should be spawned locally. The example above is not quite complete, as we should specify the *rate* at which plans are evaluated. Without a rate, the `MemoryFault` plan would be triggered at every allocation. The user can allow the compiler to determine the rate (as mentioned earlier), or can control it with explicit annotation. This can be done with a `period` argument, e.g. `period == 5s`, or with the `once` predicate which ensure that plan can only evaluate to true once.

The expressive power of the meter predicates is determined by the meters themselves. So, for instance, the memory meter only allows for constant arguments and simple inequalities. This is because the implementation of the meter needs to order all meter predicates and be guaranteed that the value of the meter expression does not change.

User-defined meters can include a `evaluate(exp)` predicate to evaluate user code. For instance, it is possible to associate a meter with the filesystem so that every I/O operation evaluates the user defined `check()` function. To reduce the granularity of events, inclusion of the `period` predicate ensures that `check()` will be invoked at most once per second.

```
plan Tick:
  meter file evaluate(check()) &&
        period == 1s;
```

We can directly associate the execution of method with a plan by writing the following (`when` is a new keyword introduced by RESCUE as a shorthand for a more thorough aspect-like syntax):

```
void takeAction() when Tick {
  ... implementation ...
}
```

2.2 Discussion

Meters monitor thread and system state. When a plan is triggered, it may in turn affect the threads affected by the fault, and update system resources as part of a recovery process.

Not all plans need to be built-in runtime environment constructs like network bandwidth or CPU usage. The user may also define her or his own plan. As an example, consider the fact that in distributed and cluster environments, there may be intermittent failures in one or more nodes. It is frequently useful to provide a "timeout" function: a component will register the fact that it is alive, and the timeout will expire if they have failed, or are unreachable. In the Java-based network services specification Jini [6], this timeout is called a *lease*. Unlike Jini, leases in RESCUE are simply user-defined instantiations of plans and meters, and not primitive.

Using RESCUE, a library that holds a resource may implement leases by simply placing a meter on the time elapsed since it was contacted by a given lease holder. An example of this can be seen in Figure 1. A meter monitors the acquisition of leases by a thread. Whenever an acquired lease exceeds a bound (in this case 5 minutes) a cancel plan is triggered, and the executed code notifies the lease manager to cancel the lease. This simple example could be effectively implemented by incorporating timer information within the lease manager itself. The advantage of using meters and plans, however, lies in added flexibility: the criteria for canceling a lease can be application-specific, and may include conditions beyond simple timer expiration, e.g., thread priority.

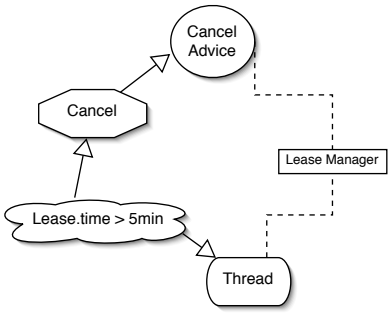

Figure 1: Using plans and advice to express a lease.

Meters and plans are partially inspired by *event-condition-action* (ECA) rules ("triggers"), an approach used mainly by database systems [11]. ECA rules check for the occurrence of events; at the event, if the condition is satisfied, the action takes places. Both academic and commercial databases use trigger mechanisms.

2.3 Synchronization

One of the problems with this form of aspect is that execution is somewhat non-deterministic; specifically, it is difficult to predict exactly when (if ever) the advice will be executed. On the other hand, advice should not execute unless any structures it is operating on are in a stable state. For example, if the advice is going to alter the representation of a data structure, it is important to ensure that no other code tries to alter that data structure while the advice executes. To put it another way, because the advice is executed asynchronously, our techniques brings along with it all of the difficulties of concurrency.

One way to formulate this problem is as an attempt to execute advice asynchronously *while also ensuring that the execution of the advice is transparent to the rest of the program.* There has been some work in ensuring that asynchronously executing code can be executed transparently [8], but only for introducing additional concurrency to single-threaded code, not for code that was originally intended to execute asynchronously.

AspectJ does not typically encourage synchronization in aspects. It offers a `Coordinator` package, which allow traditional Java synchronization to be applied to a pointcut of the program. Our proposed functionality is similar to this: the `mutex` qualifier on a plan prevents that plan from executing

concurrently with its parameter, a join point.

Our system uses a **mutex** qualifier to ensure that it is not executing concurrently with a given join point (or set of join points). It should be stressed that **mutex** is not the same as `synchronized`; the `synchronized` modifier would not only prevent the advice from executing concurrently with the join points, it would also prevent the join points from executing concurrently with each other.

The following code is an example of a plan that can only occur when the methods `o.foo` and `p.bar` do not execute:

```
plan Tick:
  meter file evaluate(check());
  mutex o.foo(), p.bar();
```

3. WEAVING

Obviously, there is a major problem with weaving an aspect that is intimately tied to the dynamic state of the program. Few of the meters we have discussed here can be resolved statically; it is impossible, for example, to determine high load levels when a program is not executing. Meters, therefore, may be triggered at any point in the program. However, it is obviously undesirable to check every meter after any instruction is executed.

This problem is not exclusive to our work. More traditional dynamic aspects can use (for example) `cflow` pointcuts, which execute advice based on whether the control flow of the program is within a given join point, require checks to indicate whether or not they are, in fact, in that join point. Exceptions in Java provide a similar problem: because most bytecodes can throw exceptions, it is necessary to check for exceptional conditions frequently.

Our first implementation is ongoing work; it involves weaving dynamically. Typically, static optimization is used to reduce the number of checks in a program. For dynamic aspects, Masuhara *et al.* [5] employ a technique based on partial evaluation that can limit the number of unnecessary run-time checks. There has also been a great deal of work in reducing the number of necessary checks for exceptions in Java [2, 10].

Unfortunately, the properties that we are checking for cannot, in general, even be approximated statically. For example, a plan that executes on high CPU load might be triggered at any point in the program. Fortunately, most of the potential meters do not require such broad support. For example, a check for low memory can simply be triggered on each allocation (when a check for low memory must be performed in any case).

4. FAULT RECOVERY IN RESCUE

As insufficient as the approaches for fault determination in modern programming languages are, the mechanisms available for fault recovery are worse: in fact, they are generally non-existent. All of the responsibility for failure recovery is in the hands of the programmer. However, it is generally accepted that programmers are very bad at planning for failure recovery [9]. This has a dramatic effect on software quality; in real environments, errors occur, and programs are not prepared to deal with them adequately. Other than doing nothing or killing a computation, applications have relatively few choices on how to recover from faults, especially those triggered by outside events. In distributed and grid environments, load balancing monitors can trigger offloading computation from one node to another. A more aggressive strategy, and one rarely supported by most implementations, is to alter the computation so that it adapts to the fault. For example, if a program has a memory-intensive data structure, and is running low on memory, it may be worthwhile to alter the data representation so that it is more compact (perhaps at the expense of another resource, such as CPU usage). Little support for these recovery mechanisms is available in modern programming languages. In fact, the programmer usually must implement them manually. This provides a great deal of flexibility. For example, if a computation must be undone, it is usually possible to undo only what needs to be undone, saving a great deal of time with checkpointing and logging the program's effects.

We provide some ideas here about how to support taking the burden of recovery off of the programmer, in combination with the programmatic techniques discussed in Section 2. Our research focuses on several different strategies: (i) support for efficient checkpointing; (ii) transactional features; (iii) task replication. We discuss (i) and (ii) below; the description extends naturally to (iii). We discuss this work in the context of the Ovm virtual machine [7], an implementation of the Real-Time Specification for Java [1].

4.0.0.1 *Checkpointing.*

One way to provide fault recovery is to checkpoint program state periodically. When done naïvely, checkpointing can be expensive, potentially requiring the copying of large amounts of state. Aspects provide one way to checkpoint relevant data *selectively*. For example, consider an application performing an iterative fixpoint calculation over a complex data structure D. Modifications to D may occur by different threads and in different parts of the program. Nonetheless, every such modification is guaranteed to be relevant. Checkpointing D whenever it is updated would thus allow tasks that fail when faults arise to restart with latest checkpointed version of D.

The code fragment shown below captures this functionality. A *write barrier* allows the execution of specified code whenever a memory store occurs. Write barriers can be prefixed with the name of the object to be monitored. A meter can be associated with a write barrier. When a write occurs on file `f` of a particular class, but before it takes effect, the current version of the structure is saved; this version can be restored if the computation must be restarted:

```
onwrite int ClassName::f when
  within(fixpointTask)  {
    save(old(f));
    return f = new(f);
}
```

This interface is already available in the Ovm customizable virtual machine, but it requires deep knowledge of the in-

ternals of the VM to use. We will make it more accessible. Read barriers are also supported. In both cases users should be extremely careful with the use of this API due to obvious performance implications. Another example of barriers for implementing replication is as follows:

```
onwrite int *::_ {
  buffer[i++]= old(_);
  if (i==buffer.length) {
    atomic { flush(); }
  }
}
```

This barrier is triggered for any integer field write. The function `flush()` tries to propagate the changes to other machines in the cluster. It is only called when the buffer is full. The `atomic` keyword is a low-level feature of the VM which essentially turns off scheduling.

4.0.0.2 Transactions

A transaction is a sequence of operations that is performed atomically: either all of it is seen to have been performed, or none of it is. If it completes successfully, it *commits*; otherwise, it *aborts*, in which case none of the updates it made are seen.

RESCUE should provide direct support for transactions. In the scope of the transaction, if a write occurs, the original value of the heap location is written to a log. When a fault occurs, an earlier program state can be re-established by restoring the original values from the log. Program flow can be resumed either from the beginning of the transaction (with some new compensatory code in place) or continued as if the error did not occur.

Transactional execution is supported in RESCUE by using lightweight language-based transactions. A RESCUE program could use aspects to attach transactional support to a computation and has programmatic control over their semantics. To give an example, consider a method `evaluate()` in a class `Task`. This method is called once for every task and encapsulates most of the computation it performs. In the case that the system runs out of memory, we would like to abort the task. This can be done by a combination of transactions and plans. The first piece of the puzzle is to attach a transaction to the code. This can be done by attaching pre- and post- actions to the `evaluate()` method.

```
rescue.transaction transaction;

before(): Task.evaluate() {
   // start a transaction before each call
   transaction = new Transaction(solution);
   // to evaluate()
}

after(): Task.evaluate() {
  if (!transaction.aborted())
    transaction.commit();
}
```

These methods start a transaction. Note that we give an argument to the constructor, the object `solution`, to indicate the transactional root. All objects transitively reachable from that root will be logged. We then define a method that gets triggered when the system gets low on memory. This method obtains the transactional log, *extracts* the contents from the log (i.e. the objects that have been modified during the transaction), and inspects their values. Finally the transaction is aborted.

```
void handleOOM() when} LowOnMemory {
 Object root = transaction.inspect();
   // obtain the contents of the log
 copyPartialResults(root);
   // user procedure to capture
   // partial results
 transaction.abort();
   // throw away changes
}
```

The default semantics of our transactional mechanism is to log all reads and writes, modulo compiler optimizations. More discriminate policies can be implemented with the low-level read/write barrier interface.

Fig. 2 illustrates how transactional support interacts with RESCUE's recovery mechanism. The figure depicts several meters and plans that monitor memory usage. When available memory becomes low (less than 5MB), thread actions are monitored are recorded in a transaction log. If memory use continues to increase, and falls below a critical threshold, an abort action is triggered, and the effects of the thread are discarded using the contents of the transaction log to restore original values. If memory use falls and exceeds a safety threshold (here $>$ 10MB), the contents of the log are committed. RESCUE's design uses plans and advice, combined with transactional mechanisms, to allow computation effects to be propagated only when safe to do so; this is a distinguishing characteristic and central to the contributions of the proposed research.

Our proposal is designed to support fault recovery in RESCUE by using transactions in conjunction with programming language support for plans and meters. We will also investigate a number of other fault recovery mechanisms, including replication and task migration.

5. CONCLUSION

In this paper we have presented a method for expressing hooks into a virtual machine to provide fault determination and fault tolerance for user code. The resulting language infrastructure allows for a more nuanced approach to handling faults; in essence, we provide an asynchronous construct that can be woven into code at runtime and gracefully handle exceptional conditions. We discuss how to combine this approach with fault recovery mechanisms that may take much of the burden of fault recovery off of the programmer.

6. REFERENCES

[1] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The*

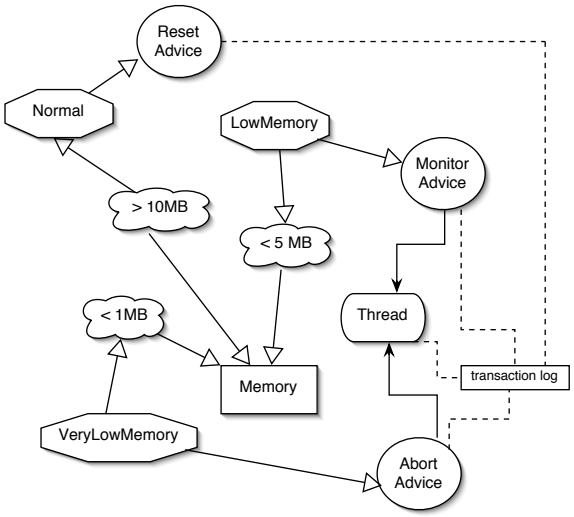

Figure 2: Advice and transactions. Clouds represent meters that monitor resources for particular conditions. Octagons represent plans that are triggered when their associated meters are satisfied. Circles denote advice associated with plans.

Real-Time Specification for Java. Java Series. Addison-Wesley, June 2000. `www.javaseries.com/rtj.pdf`.

[2] Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. Effective Null Pointer Check Elimination Utilizing Hardware Trap. *SIGPLAN Notices*, 35(11):139–149, 2000.

[3] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*. Springer-Verlag, June 2001.

[4] Barbara Liskov and Alan Snyder. Exception Handling in CLU. *IEEE Transactions on Software Engineering*, SE-5(6):546–558, November 1979.

[5] Kidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In Gary T. Leavens and Ron Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Langauges Workshop at AOSD 2002*, number 02-06 in Technical Report, pages 17–26. Department of Computer Science, Iowa State University, April 2002.

[6] Sun Microsystems. Jini Network Technology version 2.0, http://www.jini.org, June 2003.

[7] Krzysztof Palacz, Jason Baker, Chapman Flack, Christian Grothoff, Hiroshi Yamauchi, and Jan Vitek. The OVM Customizable Intermediate Representation. *To appear in The Science of Computer Programming*, 2005.

[8] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent proxies for Java futures. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Appilcations (OOPSLA)*, pages 206–223, October 2004.

[9] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pages 419–431, October 2004.

[10] John Whaley. Dynamic Optimization through the Use of Automatic Runtime Specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.

[11] J. Widom and S. Ceri. Introduction to active database systems. In J. Widom and S. Ceri, editors, *Active Database Systems - Triggers and Rules for Advanced Database Processing*, pages 2–41. Springer, Berlin,, 1996.