Towards a Domain-Specific Aspect Language for Virtual Machines

– Position Paper –

Yvonne Coady Celina Gibbs
University of Victoria, Canada
ycoady@cs.uvic.ca, celinag@uvic.ca

Michael Haupt
Darmstadt University of Technology / Hasso Plattner Institute for Software Systems Engineering, Germany
michael.haupt@hpi.uni-potsdam.de

Jan Vitek Hiroshi Yamauchi
Purdue University, USA
{jv,yamauchi}@cs.purdue.edu

Abstract

High-level language virtual machines, e. g., for the Java programming language, offer a unique and challenging domain for aspects. This position paper motivates the need for an aspect-oriented language designed precisely for this domain. We start by overviewing examples of some of the crosscutting concerns we have refactored as aspects in VMs, and then demonstrate how mainstream aspect-oriented programming languages need to be augmented in order to elegantly implement these and similar concerns. We believe current join point and advice models are not expressive enough for this domain. Predominantly this is due to the fact that the concept of a point in the execution of the VM requires the ability to explicitly specify subtle issues regarding system state and services. Finally, the paper outlines, based on a design view on virtual machines, the shape of a possible domain-specific aspect language for the implementation of such systems.

1. Introduction

A virtual machine—virtual machines in the context of this paper are always *high-level language virtual machines* [12] such as the Java virtual machine—can be seen as providing several services to the application it runs, such as memory management, execution (interpreted or JIT-compiled), adaptive optimisation, thread management, synchronisation, and so forth. These services often interact closely, and these interactions' work flows are non-trivial.

When adopting a view that regards each such service as being a *concern*, the crosscutting nature of the services and their interactions becomes perceivable. Further considering that future VMs could customise services on the basis of application-specific behaviour [18], it becomes clear that virtual machines call for employing aspect-oriented programming in their implementations.

As an example, take the cooperation of the *execution*, *organiser* and *controller* services in the Jikes RVM's [2, 3, 13] adaptive optimisation system [4]. The *execution* service is that part of the VM actually running an application; albeit it is not explicitly modelled as a dedicated service in the Jikes RVM, viewing it as such supports our notion of VMs as collections of services. *Organiser* services are responsible for collecting performance data and issuing optimisation suggestions. The *controller* gathers such suggestions and decides on whether they should be put into action.

In this setting, an organiser observing call edge hotness may be signalled by the execution service that a particular call edge has been executed so-and-so many times, and the organiser may issue an optimisation request when that call edge exceeds a certain threshold. Though optimisation based on edge hotness is straightforward to describe, its current implementation requires using several threads and queues for their communication. The logic cuts across the virtual machine and is expressed only in an implicit way, by means of attaching queues to threads appropriately.

Several attempts have been made to actually utilise AOP in implementing virtual machine services. We focus on two particular examples in this work within Java-in-Java virtual machines. The first example is *GCSpy*, a heap visualiser [16], which has been introduced to the Jikes RVM using AspectJ [14, 5]. The second example is an implementation of *software transactional memory* (STM) in the OpenVM [15].

Observing the examples' utilisations of AOP constructs, it is interesting to see that, in both cases, the implementors had to go to considerable length to realise their particular VM services as crosscutting concerns. The observation has led to the recognition of some shortcomings of existing AOP languages that in turn call for dedicated modelling mechanisms for crosscutting in the virtual machine implementation domain.

The structure of this position paper is as follows. The next Section will briefly describe the aspect-oriented realisation of GCSpy in the Jikes RVM and of STM in the OpenVM, respectively. Section 3 will describe the identified shortcomings. Section 4 will present an initial proposal for some characteristics of a domain-specific aspect language for virtual machine implementations. Not all of the proposed language mechanisms are necessarily *specific* to the virtual machine implementation domain: some merely introduce a higher level of abstraction over generally applicable AOP language mechanisms, albeit in a way allowing for the declarative expression of domain-specific requirements. Section 5 summarises the paper and outlines future work directions.

2. Case Study: GCSpy and STM

Our experience with aspects in the VM domain include two research systems developed in Java, the Jikes RVM and the OpenVM. In both cases, we used standard AOP mechanisms provided by AspectJ, as described in the high-level overview in the subsections that follow.

DSAL '06 October 23, 2006, Portland, Oregon, USA.

2.1 The GCSpy Aspect

GCspy is a heap visualisation framework designed to visualise a wide variety of memory management systems. A system as complex as a VM benefits greatly from non-invasive, pluggable tools that provide system visualisation while minimising the effects on that system. Visualisation tools such as GCSpy inherently have many fine-grained interaction points that span the system they are visualising, lending themselves to an aspect-oriented implementation.

GCSpy-specific code interacts with the base RVM in order to establish two things: (1) to gather data before and after garbage collection, and (2) to connect a GCSpy server and client-GUI for heap visualisation. Details associated with the GCSpy aspect are overviewed in the table below. The code touches 12 classes, has a 1:1 ratio of pointcuts to advice, and uses a collection that spans before/after/around advice and can be further characterised as a 'heterogenous' concern [8].

GCSpy in RVM	
Classes Involved	12
Pointcuts	15
Before Advice	5
After Advice	6
Around Advice with Proceed	2
Around Advice without Proceed	2
LOC in Aspect	126

2.2 The STM Aspect

Historically, concurrent access to shared data has been controlled using mutual-exclusion locks and condition variables where critical sections are identified with a language-specific demarcation. Software transactional memory (STM) provides a declarative style of concurrency control, allowing programmers to work with the high level abstraction of software-based transactions. The abstraction replaces critical sections with transactions, which can be in one of several possible states and can be manipulated with a set of well-defined operations.

Modifying the OpenVM to support STM involved a series of changes that lent themselves to an aspect-oriented implementation. The details of this AspectJ implementation are overviewed in the table below. The code touches 13 classes, has a 1:1 ratio of pointcuts to advice, and employs the heavy use of non-proceeding around advice.

STM in OpenVM	
Classes Involved	13
Pointcuts	22
Before Advice	1
After Advice	3
Around Advice with Proceed	4
Around Advice without Proceed	14
LOC in Aspect	114

3. AOP in VMs: Current Shortcomings

Modularity within the implementation of a VM is generally challenging [6]. Implementations tend to rely on the "black art" of fine-grained optimisation within a predominantly monolithic system. But more recent implementations of virtual machines have shown that modularity and performance can indeed co-exist, and bode well for a future were JVMs can be more easily customised according to application-specific needs [18].

In previous work, we have demonstrated how the Jikes RVM's modularity can be enhanced even with a naïve implementation of aspects, and how these aspects impact system evolution [11]. Here, we consider a more qualitative assessment of the representations of the aspects themselves, and the ways in which AspectJ could be augmented to better support the needs of crosscutting concerns in this domain. In our experience, we have determined the need to explicitly define principled points in the execution of the VM in terms of a combination of current system state and a composition of system services. The following subsections consider the ways in which domain-specific needs outstrip current join point, pointcut and advice models for AspectJ [14].

3.1 Join Points in VMs

The AspectJ joint point model was designed according to principled points in the execution of a program, such that join points remain stable under a stable interface. Similarly, a characteristic required of domain-specific join points is stability across inconsequential changes as well as being understandable to a typical VM programmer. It is our experience that the types of join points—both static and dynamic—exposed by most existing join point models are not sufficient for expressing the special needs of crosscutting concern composition in virtual machine implementations. We believe a domain-specific aspect language for VMs could adhere to the stability characteristic offered by traditional join points, while augmenting the model with further support for meaningful and much needed service composition. We believe one of the interesting challenges in this work is that, in the domain of VMs, this requires simultaneous attention to both higher-level abstractions and lower-level details.

For example, the current AspectJ-based implementation of GCSpy and STM services both require a largely 1:1 pointcut to advice ratio as shown in Sec. 2. There is little redundancy of advice as they apply to specific points in the execution of the system. As a result, the aspects are relatively large and arguably difficult to understand from a high-level perspective. Though they improve the ability for developers to reason about their internal structure and external interaction, the improvement is arguably less compelling than a higher-level representation may be able to achieve. To get a sense of what the AspectJ-based implementation looks like in terms of implementation, four fine-grained pointcut/advice pairs are required just to ensure GCSpy starts properly when the VM boots. A similar phenomenon exists in the STM aspect. An abstracted view of the GCSpy code follows.

```
before(): execution(* Plan.boot()) {
  Plan.objectMap = new ObjectMap();
  Plan.objectMap.boot();
}

before(VM_Address ref):
 args(...) && execution(* Plan.postCopy(...)) {
    Plan.objectMap.alloc(...);
}

before(VM_Address original):
 args(...) && execution(* Plan.allocCopy(...)) {
    Plan.objectMap.dealloc(...);
}

void around(VM_Address ref,..) :
 args(...) && execution(* Plan.postAlloc(...)) {
    if (allocator == Plan.DEFAULT_SPACE
        && bytes <= Plan.LOS_SIZE_THRESHOLD) {
      Plan.objectMap.alloc(...);
    else
      proceed(...);
}
```

We consider the possibility to define such a concern as a higher-level abstraction, at the granularity of a service. We envision this to include startup parameters that would specify information such as whether it is started in a separate thread, whether the application triggers it, or other threads under certain circumstances trigger it. This high-level abstraction shields the VM programmer from knowledge of the fine-grained points at which the service interacts with other services, shifting the complexity to the domain-specific aspect language processor. In the case of the startup of GCSpy, the GCSpy service interacts with the VM boot service. An example of GCSpy's interaction with the VM boot service is further illustrated in Sec. 4.

In terms of requirements along the lines of a finer granularity than currently allowed by AspectJ's pointcut model, another problem exhibited in both the GCSpy and STM aspects is the aggressive refactoring of the VM code they crosscut in order to expose appropriate join points. In [17], Siadat *et al.* provide results that suggest an intolerable amount of refactoring to expose sufficient join-points in systems code. Refactorings resulting from naïve aspect-oriented implementations yielded either (a) empty methods, or (b) new methods that do not necessarily enhance the system. They even break modularity by introducing methods for which no abstraction is required. We would prefer ways to accomplish more explicit fine-grained inter-service relationships within VMs. Our experience in this domain has lead us to conclude that current join point models are not fine-grained enough to be highly effective within VMs.

3.2 Pointcut Descriptors

Pointcut descriptors determine whether a given join point matches a point in the execution of the VM. In our experience, virtual machine services may expose some points where other services may interact and often inherently require access to dynamic, often shared, VM system state. For example, the execution concern may expose a point indicating that a certain call edge hotness has exceeded a given threshold, which might be interesting for the adaptive optimisation concern. Similarly, the generalisation of memory management systems monitored by GCSpy, or controlled by STM, also lend themselves to this scenario, where one service is interested in points in the execution of the composed system only if system state is appropriate.

The circumstances in which an optimisation request is to be generated are cumbersome to express using the pointcut language present in AspectJ. The pointcut must match *only* if the call edge hotness actually *exceeds* the threshold; as long as a call edge's invocation count is less than the threshold, the optimisation aspect is in the "do not optimise" state, which it leaves in the moment the count exceeds the threshold. As soon as another, greater threshold is exceeded, the aspect may choose an even higher level of optimisation for the call edge. This basically constitutes a stateful aspect [10] and could be expressed using the appropriate means, e. g., tracematches [1].

We would like to stress that this kind of pointcut is not specific to the domain of virtual machine implementations and hence does not actually constitute a need for a *domain-specific* language construct. It is rather the case that stateful aspects of this kind characteristically occur in the domain. Still, a declarative way of expressing them, other than a generic one as seen in the tracematch syntax, may be more viable by means of introducing greater abstraction.

3.3 Advice

Advice supply a means of specifying code to run at a join point. It is important for a domain-specific language to carefully consider the nature of the VM domain. It is unacceptable, within this domain, to introduce possibly prohibitive performance penalties, or dramatic increases in the system's memory footprint. Returning to the adaptive optimiser example mentioned above, if the execution service signals sufficient call edge hotness, the optimising service actually should not necessarily kick in immediately. In this case, it could harm the VM's performance to optimise every single call edge as the execution service sees it fit for being optimised. The VM should instead wait until there are ample time and resources available. Usually, this is implemented using separate threads and a queue storing requests (as in the Jikes RVM; cf. above). The two concerns interact in a detached way; they utilise *asynchronous* advice [7], as met in, e. g., the AWED language [9].

Asynchronous advice cannot be expressed directly using simple AspectJ mechanisms, as the required queues and associated state have to be introduced as explicit data structures. Although there is no such concept as an asynchronous advice in traditional AOP advice models, we believe this to be highly desirable in VMs. Given that there are very likely many VM services that do not interact synchronously, modeling such services as crosscutting concerns calls for providing a mechanism allowing for such definitions.

4. A DSAL for Virtual Machines

Based on our experience with aspects in VMs so far, we believe a domain-specific aspect language for virtual machines must address the following issues:

1. Many common services in a VM can be structured as crosscutting concerns. An according DSAL should provide a high-level view on VM abstractions and their corresponding implementation that allows for expressing virtual machine services as modules of their own, explicitly specifying their characteristics and relations to other services. The abstractions should be generalisable across multiple VMs, enabling the services to be generalisable as well.
2. Existing join point models are not sufficient to express the rationale and type of interaction between the concerns found in a VM. A DSAL for VMs should allow for exposing types of join points met in virtual machines, for expressing them in appropriate pointcuts, and for specifying advice that should run at those points in meaningful ways (synchronously/asynchronously).

Regarding the first issue, a VM service should be expressible as a single, configurable module in terms relative to core VM abstractions. For a simple example, a completely asynchronous service (running in a dedicated thread) could be succinctly expressed like this:

```
detached service AdaptiveOptimiser { ... }
```

or advice associated with a boot-time sequence might be expressed like this:

```
service GCSpy {
  during(VMbooting): { ... }
  ...
}
```

with the DSAL weaver knowing about the places to join to in terms of boot-time logic, as signified by `during`. Of course, there may be cases where it is necessary to specify the order of service startups, in case of possible conflicts.

Each service should also make clear which points it exposes to others, establishing a clear interface for crosscutting behaviour. For example, in the case of the STM aspect, it may be possible to apply either optimistic or pessimistic concurrency control strategies depending on the level of conflict in the system. The points at which these different concurrency control services could be applied should be exposed by the STM service.

As for adaptive optimisation, the circumstance that a method `m2()` should be inlined in `m1()` when it has been called therefrom

more than 100 times could be expressed like this, as part of an organiser service:

```
detached service CallEdgeOrganiser {
  whenever(VM_Method m1, VM_Method m2):
    edgehotness(m1,m2) exceeds 100 {
      inline(m2, m1);
    }
}
```

In this example, the `whenever` advice type means asynchronous advice execution, and the `exceeds` comparison operator implies that the `edgehotness` value must exceed the given threshold for the pointcut to match. The `edgehotness` value, by the way, is exposed from the execution service.

5. Summary

Our experience with aspects in VMs leads us to believe that this domain could benefit greatly from VM-specific AOP mechanisms. In this paper, we have argued this point based on our sample aspects in the RVM, the OpenVM, and additionally reasoning about a common optimisation scenario. We also have described the ways we believe the join point model in AspectJ could be augmented to suit this domain. Based on these observations, we have proposed to regard crosscutting concerns in virtual machines as a special domain of aspects, requiring support in the form of dedicated language expressions.

Future work in this area will focus on a close examination of crosscutting in high-level language virtual machines. Based on results from this analysis, a more complete critique of existing AOP models will be formulated, along with a more detailed version of the proposed domain-specific aspect language.

References

[1] C. Allan et al. Adding Trace Matching with Free Variables to AspectJ. In *Proc. OOPSLA 2005*, pages 345–364. ACM Press, 2005.

[2] B. Alpern, A. Cocchi, D. Lieber, M. Mergen, and V. Sarkar. Jalapeño—a compiler-supported java virtual machine for servers. ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS '99), May 1999.

[3] B. Alpern et al. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *OOPSLA 2000 Proceedings*, pages 47–65. ACM Press, 2000.

[5] AspectJ Home Page. `http://www.eclipse.org/aspectj/`.

[6] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ICSE*, pages 137–146, 2004.

[7] M. Cilia, M. Haupt, M. Mezini, and A. P. Buchmann. The convergence of aop and active databases: Towards reactive middleware. In *Proc. GPCE 2003*, volume 2830, pages 169–188. Springer, 2003.

[8] A. Colyer and A. Clement. Dlarge-scale aosd for middleware. In *AOSD '04: Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 56–65. ACM Press, 2004.

[9] R. Douence, D. Le Botlan, J. Noye, and M. Sudholt. Concurrent aspects. In *Proc of GPCE*. Springer, 2006.

[10] R. Douence, P. Fradet, and M. Sudholt. A framework for the detection and resolution of aspect interactions. In *Proc of GPCE*, pages 173–188. Springer, 2002.

[11] C. Gibbs, R. Liu, and Y. Coady. Sustainable system infrastructure and big bang evolution: Can aspects keep pace? In *Proc. ECOOP 2005*. Springer, 2005.

[12] J. E. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan-Kaufmann, 2005.

[13] The Jikes Research Virtual Machine. `http://jikesrvm.sourceforge.net/`.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. Lindskov Knudsen, editor, *Proc. ECOOP 2001*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[15] OpenVM Home Page. `http://ovmj.org/`.

[16] T. Printezis and R. Jones. Gcspy: an adaptable heap visualisation framework. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 343–358. ACM Press, 2002.

[17] J. Siadat, R. Walker, and C. Kiddle. Optimization aspects in network simulation. In *AOSD '06: Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 122–133. ACM Press, 2006.

[18] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 49–60. ACM Press, 2004.