Towards an Aspect Weaving BPEL engine [*]

Carine Courbis
University College London
Department of Computer Science
Adastral Park - Martlesham
IP5 3RE, UK
carine.courbis@bt.com

Anthony Finkelstein
University College London
Department of Computer Science
Gower Street, London
WC1E 6BT, UK
a.finkelstein@cs.ucl.ac.uk

ABSTRACT

This position paper proposes the use of dynamic aspects and the visitor design pattern to obtain a highly configurable and extensible BPEL engine. Using these two techniques, the core of this infrastructural software can be customised to meet new requirements and add features such as debugging, execution monitoring, or changing to another Web Service selection policy. Additionally, it can easily be extended to cope with customer-specific BPEL extensions. We propose the use of dynamic aspects not only on the engine itself but also on the workflow in order to tackle the problems of Web Service hot deployment and hot fixes to long running processes. In this way, composing a Web Service "on-the-fly" means weaving its choreography interface into the workflow.

Keywords

BPEL engine, dynamic aspect, visitor design pattern, Web Service, SOA.

1. INTRODUCTION

Increasingly, applications are built from existing components or services at a coarser-grain level than manipulating classes. The advantage of using Web Services in comparison to components is to enable the development of loosely coupled distributed business applications that are highly interoperable and cross organisational boundaries. This paradigm is called Service-Oriented Computing[1] (SOC). There is a fundamental shift toward a Service-Oriented Architecture (SOA) supported by the use of standards: WSDL (*Web Service Description Language*) to describe the business interfaces of the services (i.e. the contracts), UDDI (*Universal Description, Discovery and Integration*) to publish and discover them, and SOAP (*Simple Object Access Protocol*) to exchange messages between them, independently of the underlying communication protocol.

[*]This research is supported by the Generative Software Development project funded by BT Exact.

[1]The first international conference on SOC has just taken place in Italy (see http://www.unitn.it/convegni/icsoc03.htm).

The Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) March 2004, Lancaster, UK

With Web Services, there is a layer of abstraction above the components that makes it possible to integrate a wide variety of incompatible systems (interoperability) to build an application. This layer of abstraction is often called the orchestration layer. The most well established orchestration technology for Web Services is BPEL (*Business Process Execution Language*) [1], originally created by BEA, IBM, and Microsoft, and currently submitted for standardisation to the OASIS consortium. This XML-based language is rather small [10] but sufficient to handle variables with scopes, loops, conditional branches, synchronous and asynchronous communications, concurrent activities with correlated messages, transactions, and exceptions. With this language, a business process can be described by gluing different Web Services together, creating a new Web Service. This process description is interpreted by a BPEL engine.

In our view, the BPEL engine should be minimal but easy to configure and extend to cope with new requirements and features. But orthogonal functionalities such as execution monitoring do not need to be enabled at each BPEL interpretation as they are themselves performance-inefficient. Selecting Web Services is another example of a possible adaptation. Instead of choosing at design or deployment time which Web Service to use, the engine can choose one at runtime, in accordance with specified criteria, on the first occasion the service is invoked.

As BPEL is an extensible language (that is new instructions can be used in a process description to cope with user-specific needs), its engine also needs to be extensible to integrate new behaviours for user-specific instructions. To build a flexible BPEL engine, we uses two techniques: Aspect-Oriented Programming (AOP) [7] and the visitor design pattern [5, 12]. By using these techniques, the engine can be extended both statically[2] by inheritance and dynamically by aspects.

We intend to apply dynamic aspects not only to the engine itself but also to the BPEL process to tackle the problems of Web Service hot deployment and hot fixes to long running processes. For example, it can be useful to add an unforeseen Web Service at runtime. We have also been investigating a specific case in which services are used to support a large Grid-based computational chemistry application. In this application, there is a need for steering, in

[2]Dynamically, if the engine supports dynamic class loading.

other words changing the end of the workflow depends upon results identified in earlier stages.

Our BPEL engine will manipulate different types of aspects; it can be seen as an aspect weaver that orchestrates Web Services.

The aim of this paper is to explain how such an adaptable BPEL infrastructure engine can be created taking advantage of the visitor design pattern and dynamic aspects. It is organised as follows. Section 2 presents a brief overview of related work. Section 3 describes the design of our BPEL engine. We conclude the paper in Section 4.

2. RELATED WORK

It is important that "systems infrastructure" software such as application servers, virtual machines, middleware, compilers, and operating systems be open and adaptable. Otherwise no user-specific feature or requirement can be added after implementation time. To integrate new functionalities requires redevelopment of the whole software. For example, Gilad Bracha *et al* have developed their own Java compiler to create a superset of the Java programming language, *GJ*, with generic types and methods [4]. It was not possible to adapt the Java compiler to cope with this language extension. Other examples are the VM-based runtime MOPs (*Meta-Object Protocols*) such as *Guaraná* [11] that have developed their own JVM (*Java Virtual Machine*) to intercept operations at runtime (with a VM-based solution, the passage from the base level to the meta level is invisible to the programmer).

One solution to build more adaptable system infrastructure software is to use dynamic aspects. They are appropriate for run-time adaptations in service architectures [15] and more precisely as hot fixes. By contrast with static aspects such as the ones used in *AspectJ* [6], dynamic aspects can be woven or unwoven into/from a program "on-the-fly". Sato *et al* present [16] a good introduction to dynamic AOP. They also describe their dynamic weaver, *Wool*, that is a hybrid of two aspect implementation approaches. At runtime and on demand, it either embeds hooks into a class for executing the advices and reloads it into the JVM, or inserts hooks as breakpoints into the JVM. At least two dynamic aspect systems, *JAC* [14] and *Handi-Wrap* [3], use static code translation on the byte-code, statically inserting the hooks at all the potential join points. Using aspects on SOAs will make it possible, for example, to check constraints (design by contracts), such as the ones proposed in the Web Service Offerings Language (WSOL) [18], or to monitor the execution with agents.

Using design patterns when implementing systems make them more flexible. In the case of interpretors, the visitor design pattern is very often chosen. Recently, this pattern was used, for example, to implement *Joeq* [19], a virtual machine and compiler infrastructure.

The existing service description languages and Web Service flow languages address business process dynamics and non-functional properties poorly. For example, in the current BPEL version, it is not possible to add on demand or replace a Web Service (a partner) at runtime; the workflow needs to be stopped to be adapted. The idea of using aspects for dynamic workflow adaptations or execution controls has been outlined in [2]. With aspects, new activities can be added or replaced, the control flow modified, the policy resolution to assign resources to activities changed or extended, and resource invocations replaced.

To address the problem of dynamic selection and composition of Web Services, DAML-S [9], an ontology of services, proposes the use of semantic descriptions. These descriptions will then be manipulated by different agents or software such as a semi-automatic composer of Web Services [17]. With the latter, compositions on demand are based on semantic descriptions and are validated by human controllers. Daniel Mandell and Sheila McIlraith describe in [8] how to augment BPEL with Semantic Web technology.

3. AN OPEN, EXTENSIBLE, AND CONFIGURABLE BPEL ENGINE

To have more flexible BPEL processes, we have chosen to design and implement an open, extensible, and configurable BPEL interpreter. Its core logic will be rather small as the language does not contain many instructions, but we plan to enrich it with new features such as:

- To easily extend or modify its behaviour;
- To select or replace Web Services after deployment time;
- To plug or unplug aspects in/from the engine on demand;
- To hot-fix the workflow; for example, to compose on demand new Web Services;

The advantages of these features and how we plan to implement them are now presented. At this end of this section, we also briefly put together the architecture technical details of our language interpreter.

3.1 Engine behaviour extension or modification

BPEL is a language that can be extended with new user-specific instructions such as launching an executable, or replacing a Web Service. This means that its engine needs to be easy to extend. Also it would be useful to have the capability to modify the engine behaviour to take into consideration user-specific requirements. The visitor design pattern meets these requirements as it separates the data structures and the semantics. The behaviour of each BPEL element is represented as a visit method and the set of these methods contained in a class (the visitor). As the engine code will be modular, it will be easy to understand, maintain, extend by inheritance, and modify by visit method overridings.

3.2 Selection and replacement of Web Services

Selecting a Web Service can depend on different criteria and constraints: QoS (*Quality of Service*), price, the result of a request, the trust you have in the provider, etc. In the well-known Web Service example, the travel agency, the selection policy for the airline company can be to take the lowest fare from London to Morocco at Christmas time, or the quickest trip without a stop, or to take British Airways for the frequent flyer points. To find the lowest fare, each airline company needs to be invoked; the selection policy can be a minimal business process. Each partner (Web Service) involved in a business process can have a different selection policy. The selection may be performed at runtime on the first occasion the service is invoked or at replacement request, not at design or deployment time. We plan to accept one selection policy per partner and a generic one if none is provided. This policy will be used at runtime by the engine to select, from an UDDI registry, a Web Service that is signature and constraint compliant.

There is also a need to be able to replace, at runtime, a Web Service that is slow, unresponsive, or no longer useful for the current iteration. In this way, the workflow can be adapted to improve performance or QoS, to avoid termination because there is no answer from one partner, and to use another similar service in a loop or on user demand. The substitution can only occur if the new Web Service is service-signature compliant (same WSDL description as there is no service adaptor) and if the service to be replaced is in a stable state (not in a transaction, and without an initialisation or one that does not impact on other partners).

To be compliant with the specifications, the core logic of our BPEL engine (the visitor) should contain no Web Service selection or replacement code. The solution is to set a hook before service invocations (the `invoke` visit method) to add these functionalities. In this way, services can be selected and even replaced at post-deployment, as well as selection policies.

3.3 Orthogonal concerns

It can be useful to enrich the core logic of the engine with different concerns at post-implementation. In this way, the engine is more modular, adaptable, and easy to maintain. As some of the concerns can be impact-performant, such as execution profiling or debugging, their corresponding aspects should be enabled to be woven or unwoven on demand during execution. With these dynamic non-functional aspects, the engine can, for example, be controlled by agents that monitor the execution and take actions if one service provider is not responding. Such a concern can be useful especially for long running processes. We can also identify the need for functional aspects between two service invocations to perform local code execution such as converting the data into another format.

We have defined an aspect BPEL-specific language using XPath as a pointcut language to identify the join points (matching the BPEL document) and Java as the advice language[3]. In our first version, we have statically set hooks to execute advices at all the potential join points; that is before and after any BPEL instruction (visit method) such as `invoke` or `receive`, and at any process variable modification. Plugging in an aspect means registering it on the current process and also selecting the different nodes of the process document (AST - *Abstract Syntax Tree*) identified by XPath expressions to annotate them with the aspect name and the name of the advice to execute. Before and after interpreting an instruction, our system checks if there is any annotation and calls the method to execute (advice) if this aspect is still registered. Unweaving an aspect only means removing the aspect from the registry.

3.4 Hot fixes applied to the workflow

For long running processes, adapting a workflow, according to earlier results, by stopping it is not acceptable. There is a need to modify, at runtime, the end of the workflow by adding new computational instructions, and replacing or deleting some instructions. This can also be seen as BPEL aspects, using XPath to identify the join points but BPEL as the advice language (instead of Java for the aspects on the engine). As these aspects act upon the workflow (functional aspects) and have their advice in BPEL, we plan to directly transform the process AST. These transformations can only be applied to the workflow at some precise points and under certain conditions that we need to identify to ensure the stability of the system. For example, deleting a BPEL sequence can only occur if the engine has not started interpreting it.

[3]The implementation language of our engine is Java-based.

An important example of such hot-fixes is the composition, on demand, of a new Web Service and thus the addition of its choreography interface. The dynamic aspect technology is our solution to address dynamic composition of Web Services: the choreography interface (BPEL instructions) can be seen as advices and where to weave them as pointcuts. Composing a new Web Service means transforming the AST workflow to integrate the piece of its choreography interface (BPEL advice). With this capability, the workflow can be extended to meet unforeseen post-deployment requirements and user needs.

3.5 Architecture technical details

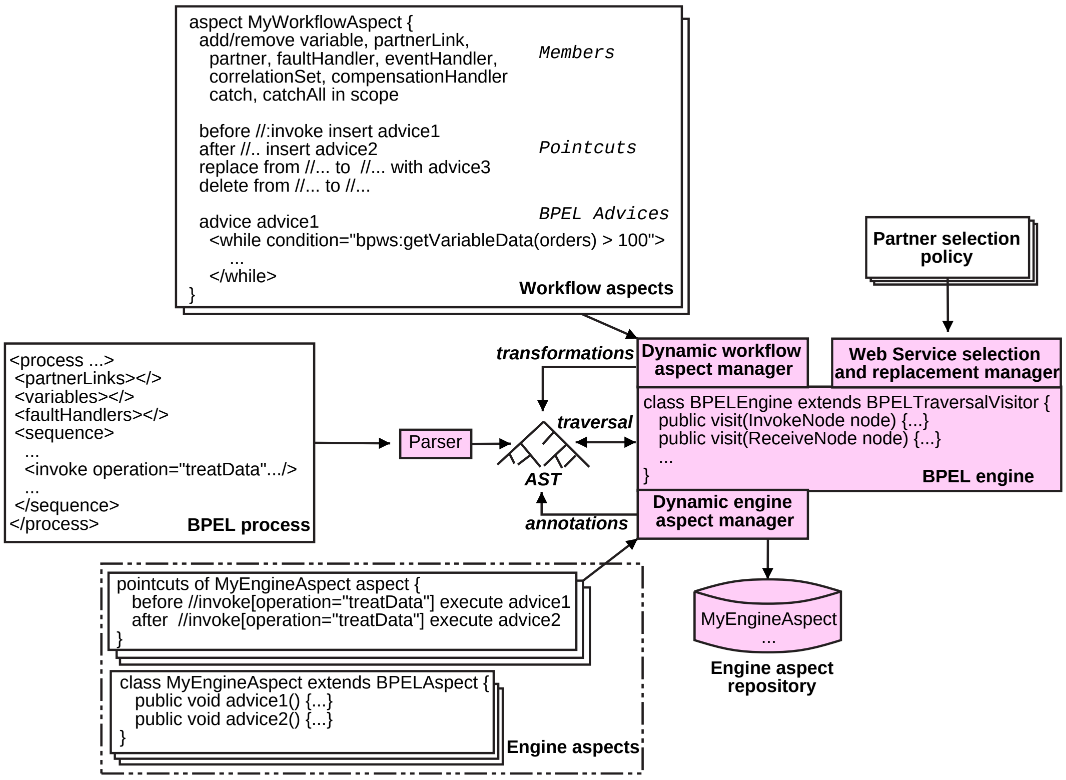
The core logic of our system (see Figure 1) is the BPEL interpretor, implemented using the visitor design pattern. It contains one visit method for each BPEL instruction and traverses the typed structures, the BPEL trees, from top to bottom. These trees are not only strictly typed to meet the pattern requirements but are also based on the DOM API to enable XPath selections of their nodes, which is useful for the implementation of our BPEL aspect languages.

The code to handle the selection and replacement of Web Services, and the engine aspects is represented as two aspects respectively that we can plug in or unplug from the BPEL interpretor. In this way, the interpretor can be used alone (faster) or extended, at runtime, with these functionalities. Its code is independent from the BPEL aspect and the Web Service selection code, and is compliant with the BPEL specification. This possibility of plugging aspects is due to our visitor design pattern implementation that checks before and after each visit method call to see if some advices need to be executed. More precisely, this check is done in the visit method dispatcher (in our case, a generic visit method instead of the different accept methods implemented in each BPEL element class). More details about the visitor design pattern implementation we are using can be found in [13].

The workflow aspect manager is also code independent from the engine. It just needs to suspend the engine when performing the transformations on the interpreted BPEL document at some stable points and to get access to its data environment to add or remove members (variables, partners, etc.). Additionally, the annotations of the engine aspects already plugged in should be propagated onto any new BPEL instruction added by insertion or replacement.

4. CONCLUSION

In this position paper, we argue that the visitor design pattern and dynamic aspects can be used to implement an extensible and adaptable BPEL engine, thus in SOAs. The benefit of using the visitor design pattern is to write modular code that is easy to extend by inheritance. This characteristic, in the context of an extensible language such as BPEL, is important so as to ease the incorporation of new instruction behaviours into the interpreter. Involving aspects into the



```
aspect MyWorkflowAspect {
  add/remove variable, partnerLink,
    partner, faultHandler, eventHandler,
    correlationSet, compensationHandler
    catch, catchAll in scope

  before //:invoke insert advice1
  after //.. insert advice2
  replace from //... to  //... with advice3
  delete from //... to //...

  advice advice1
    <while condition="bpws:getVariableData(orders) > 100">
      ...
    </while>
}
```

```
<process ...>
 <partnerLinks></>
 <variables></>
 <faultHandlers></>
 <sequence>
   ...
   <invoke operation="treatData".../>
   ...
 </sequence>
</process>
```

```
class BPELEngine extends BPELTraversalVisitor {
   public visit(InvokeNode node) {...}
   public visit(ReceiveNode node) {...}
   ...
}
```

```
pointcuts of MyEngineAspect aspect {
   before //invoke[operation="treatData"] execute advice1
   after  //invoke[operation="treatData"] execute advice2
}
```

```
class MyEngineAspect extends BPELAspect {
   public void advice1() {...}
   public void advice2() {...}
}
```

Figure 1: Overview of the BPEL engine architecture

engine makes it possible to separate, in a modular way, the different concerns, to focus only on its core logic in the first place, and to rapidly integrate unforeseen concerns into it in a non-invasive way. For greater flexibility, we have chosen to have dynamic aspects to be able to weave and unweave them into/from at runtime.

We also argue that dynamic aspect techniques can not only be used in the engine itself but also on business processes to address the well-known problems of Web Service hot deployments and hot fixes. Additionally, we believe that the BPEL engine should be customised with different selection policies as Web Service selection should be done after deployment, and with Web Service replacement capability.

We have started the development of our system, using SmartTools [13], a DSL (*Domain-Specific Language*) development environment, to quickly prototype tools for our different languages (BPEL, the engine aspect language, and the workflow aspect language). Later, we will need to refine our aspect languages by identifying which pointcuts are needed for advices either in BPEL (for the hot fixes such as the choreography interface compositions) or in Java (for orthogonal concerns).

5. ACKNOWLEDGEMENTS

The authors want to thank David Lesaint and George Papamargaritis from BT Exact for the fruitful discussions, as well as Ben Butchart from UCL.

6. REFERENCES

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, and I. Trickovic. Business Process Execution Language for Web Services version 1.1. Technical report, BEA, IBM, Microsoft, SAP, Siebel Systems, May 2003.
http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

[2] B. Bachmendo and R. Unland. Aspect-Based Workflow Evolution. In *Tutorial and Workshop on Aspect-Oriented Programming and Separation of Concerns*, Lancaster, UK, August 2001.
http://www.comp.lancs.ac.uk/computing/users/marash/aopws2001/papers/ba

[3] J. Baker and W. Hsieh. Runtime Aspect Weaving Through Metaprogramming. In *First International Conference on Aspect-Oriented Software Development*, pages 86–95, Enschede, The Netherlands, April 2002. ACM.
http://www.cs.utah.edu/~wilson/papers/handiwrap-aosd02.pdf.

[4] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In *Proceedings of OOPSLA'98*, Vancouver, Canada, October 1998. ACM Press.
http://www.cis.unisa.edu.au/~pizza/gj/Documents/gj-oopsla.pdf.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Pub Co, January 1995. ISBN 0201633612.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knuden, editor, *Proceedings of European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–355, Budapest, Hungary, June 2001.

http://www.cs.ubc.ca/~gregor/kiczales-ECOOP2001-AspectJ.pdf.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira, and J.-M. Loingtier. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag. http://www.cs.ubc.ca/~gregor/kiczales-ECOOP1997-AOP.pdf.

[8] D. J. Mandell and S. A. McIlraith. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In D. Fensel, K. Sycara, and J. Mylopoulos, editors, *Proceedings of the Second International Semantic Web Conference*, number to appear in LNCS, Sanibel Island, USA, October 2003. Springer-Verlag. http://www.ksl.stanford.edu/people/sam/iswc2003sam-djm-FINAL.pdf.

[9] T. D. S. C. D. Martin, M. Burstein, G. Denker, J. Hobbs, L. Kagal, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. DAML-S: Semantic Markup For Web Services 0.9, 2003. white paper available online at http://www.daml.org/services/daml-s/0.9/daml-s.pdf.

[10] N. Mukhi. Reference guide for creating BPEL4WS documents. Technical report, IBM, November 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-bpws4jed/.

[11] A. Olivia and L. E. Buzato. The Design and Implementation of Guaraná. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, San Diego, USA, May 1999. http://www.ic.unicamp.br/~oliva/guarana/docs/desimpl.ps.gz.

[12] J. Palsberg and C. B. Jay. The Essence of the Visitor Pattern. In *Proceedings of COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, August 1998. http://www.cs.ucla.edu/~palsberg/paper/compsac98.pdf.

[13] D. Parigot, C. Courbis, P. Degenne, A. Fau, C. Pasquier, J. Fillon, C. Held, and I. Attali. Aspect and xml-oriented semantic framework generator: Smarttools. In M. van den Brand and R. Lämmel, editors, *ETAPS'2002, LDTA workshop*, volume 65 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, Grenoble, France, April 2002. Elsevier Science. http://www.elsevier.nl/gej-ng/31/29/23/117/52/33/65.3.009.pdf.

[14] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns: Third International Conference, Reflection'01*, volume 2192 of *LNCS*, pages 1–24, Kyoto, Japan, September 2001. http://jac.aopsys.com/papers/reflection.ps.

[15] A. Popovici, G. Alonso, and T. Gross. Just-In-Time Aspects: Efficient Dynamic Weaving for Java. In *Proceedings of the 2nd international conference on Aspect-Oriented Software Development*, pages 100–109, Boston, USA, March 2003. ACM Press. http://www.lst.inf.ethz.ch/research/publications/publications/AOSD_2003/AOSD_2003.pdf.

[16] Y. Sato, S. Chiba, and M. Tatsubori. A Selective, Just-in-Time Aspect Weaver. In Springer-Verlag, editor, *Proceedings of Generative Programming and Component Engineering (GPCE'03)*, number 2830 in LNCS, pages 189–208, Erfurt, Germany, September 2003. http://www.research.ibm.com/trl/people/mich/pub/200306_gpce2003.pdf.

[17] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic Composition of Web Services using Semantic Descriptions. In *ICEIS2003, Web Services: Modeling, Architecture and Infrastuture workshop*, Angers, France, April 2003. ICEIS Book. http://www.mindswap.org/papers/composition.pdf.

[18] V. Tosic, B. Pagurek, and K. Patel. WSOL - A Language for the Formal Specification of Various Constraints and Classes of Service for Web Service. In *The International Conference On Web Services, ICWS'03*, pages 375–381, Las Vegas, USA, June 2003. CSREA Press. http://www.sce.carleton.ca/netmanage/papers/TosicEtAlResRepNov2002.pdf

[19] J. Whaley. Joeq: A Virtual Machine and Compiler Infrastructure. In *The Workshop on Interpreters, Virtual Machines, and Emulators*, pages 58–66, San Diego, USA, June 2003. ACM SIGPLAN 2003. http://www.stanford.edu/~jwhaley/papers/ivme03.pdf.