CSOM/PL
A Virtual Machine Product Line

Michael Haupt, Stefan Marr, Robert Hirschfeld

Technische Berichte Nr. 48

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

Universität Potsdam

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 48

Michael Haupt | Stefan Marr | Robert Hirschfeld

CSOM/PL

A Virtual Machine Product Line

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.de/ abrufbar.

Universitätsverlag Potsdam 2011
http://info.ub.uni-potsdam.de/verlag.htm

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 4623 / Fax: 3474
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2011/5233/
URN urn:nbn:de:kobv:517-opus-52332
http://nbn-resolving.org/urn:nbn:de:kobv:517-opus-52332

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-134-9

CSOM/PL
A Virtual Machine Product Line

Michael Haupt
michael.haupt@hpi.uni-potsdam.de

Stefan Marr
stefan.marr@vub.ac.be

Robert Hirschfeld
hirschfeld@hpi.uni-potsdam.de

April 18, 2011

CSOM/PL is a software product line (SPL) derived from applying multi-dimensional separation of concerns (MDSOC) techniques to the domain of high-level language virtual machine (VM) implementations. For CSOM/PL, we modularised CSOM, a Smalltalk VM implemented in C, using VMADL (virtual machine architecture description language). Several features of the original CSOM were encapsulated in VMADL modules and composed in various combinations. In an evaluation of our approach, we show that applying MDSOC and SPL principles to a domain as complex as that of VMs is not only feasible but beneficial, as it improves understandability, maintainability, and configurability of VM implementations without harming performance.

1 INTRODUCTION

Implementors working on high-level language virtual machines (VMs) [36] typically face the characteristic problem of very high complexity, expressed in source code as intricately intertwined module dependencies. Even worse, even though logical modules such as memory management and emulation engine are perceivable, they can often hardly be identified as such in the code. The interdependencies lead to partial functionality realisations of logical modules being interwoven with other logical modules' code. This, in turn, is due to a lack of modular abstraction application in the domain of VM implementations.

A second difficulty with VM implementations is that they frequently need to be tailored to specific needs. Different dimensions of interest are relevant in this regard. The particular application domain might call for differently aggressive optimisation. For instance, Sun's HotSpot JVM features two different versions optimised for client- or server-specific applications, which use different just-in-time (JIT) compilers[1]. The optimisation to be used is chosen at VM startup time. The Jikes RVM[2] [5, 6] can employ a selection of two different JIT compilers that can moreover be combined with an adaptively optimising infrastructure [12].

Other dimensions of interest are, e. g., memory allocation behaviour, calling for different choices of garbage collectors (GCs) [22, 11]; availability of CPU cores, influencing the threading model (native or user-level threads, or hybrid scheduling); and the target platform, possibly imposing all kinds of limitations on the rest of the implementation (e. g., VMs for resource-constrained devices). Clearly, all of the choices have implications on the interactions of the different modules, in turn leading to more intricate relationships [20].

The notion of *service modules* was introduced [20] to address module entangling in VMs. A service module is a module with a *bidirectional interface*—in the fashion of open modules [2] or XPIs [19]—that can not only be sent requests, but that can also exhibit internal situations of interest to the outside. An initial proposal of an architecture description language (VMADL) was introduced, along with a proof of concept implementation, supporting the concepts of service modules at the programming language level.

The characteristics of the second problem suggest to regard the various VM subsystems and their variations as *concepts* and *features* in the sense of a software product line (SPL) [14]. This paper reports on the results achieved in combining the VMADL approach and SPL principles and applying them to the VM implementation domain.

In particular, we have applied these principles and techniques to CSOM[3] [21], a VM for a Smalltalk [18] dialect, which is used in teaching at the Hasso Plattner Institute. Focusing on understandability and clarity, CSOM is moderately complex, featuring a simple bytecode interpreter and a mark/sweep GC [22]. Despite its simplicity, CSOM exhibits characteristic crosscutting concerns [20]; increasingly so when extended with additional or alternative features, e. g., in coursework settings.

VMADL was used to modularise several extensions to CSOM that were previously introduced by hand. The extensions were of different kinds—garbage collectors, multi-threading implementations, optimised representation of integral numbers, and image persistence—and exhibited different crosscutting characteristics. Encapsulating these extensions in service modules allowed for turning CSOM

[1] java.sun.com/products/hotspot/whitepaper.html

[2] jikesrvm.org

[3] www.hpi.uni-potsdam.de/swa/projects/som

into an SPL, which we call CSOM/PL [4], enabling different combinations of modules to be chosen at compile-time.

In summary, the contributions of this paper are as follows.

- We present the first full version and implementation of VMADL. It differs significantly from the proof of concept [20] in that it has explicit constructs and extended support for service module combinations. Moreover, the proof of concept was replaced with a more stable implementation that applies AspectC++[5] [30], a production-quality AOP extension to C++.
- We show that an approach based on multi-dimensional separation of concerns *at source code level* alleviates programming in a complex domain with intricate crosscutting relationships. The beneficial effect of applying VMADL in the VM implementation domain consists in making architectural interdependencies explicit not only at the source code level, but abstractly so, by means of interactions between bidirectional interfaces.
- We demonstrate how the approach can be used to establish an SPL in this domain, fostering configuration and variability management as well as code reuse. The SPL includes *combinations* of features that were previously applied in isolation only. The CSOM product line was realised using *pure::variants*[6], a state-of-the-art tool for SPL development. By virtue of pure::variants, the CSOM/PL product space is consistently represented as a feature model, and products can be easily configured and validated. Once a product has been configured, corresponding build scripts can be generated by the SPL tool.

In the remainder of this paper, we first introduce the CSOM VM in the following section. We then, in Sec. 3, adumbrate the architectural principles at work in CSOM/PL, and give an introduction to the language VMADL, including a brief description of its implementation. The CSOM/PL results and how they were achieved is illustrated in Sec. 4. The evaluation of the obtained results is described in Sec. 5. Related work is discussed in Sec. 6, and Sec. 7 summarises the paper and gives future work directions.

2 THE CSOM VIRTUAL MACHINE

CSOM[7] [21] is a VM for a Smalltalk dialect designed for teaching purposes. Its precursor, SOM (Simple Object Machine) was implemented in Java at the University of Århus. CSOM is a port of SOM to C; that has been done at the Hasso Plattner Institute. There, CSOM has been used in two graduate courses on virtual machines in 2007 and 2008.

[4]A live CD image with the complete CSOM/PL is available at `www.hpi.uni-potsdam.de/swa/projects/som`. Due to license regulations, pure::variants cannot be included with the CD image.

[5]`www.aspectc.org`

[6]`www.pure-systems.com/pure_variants.49.0.html`

[7]Pronounced "*see*-som".

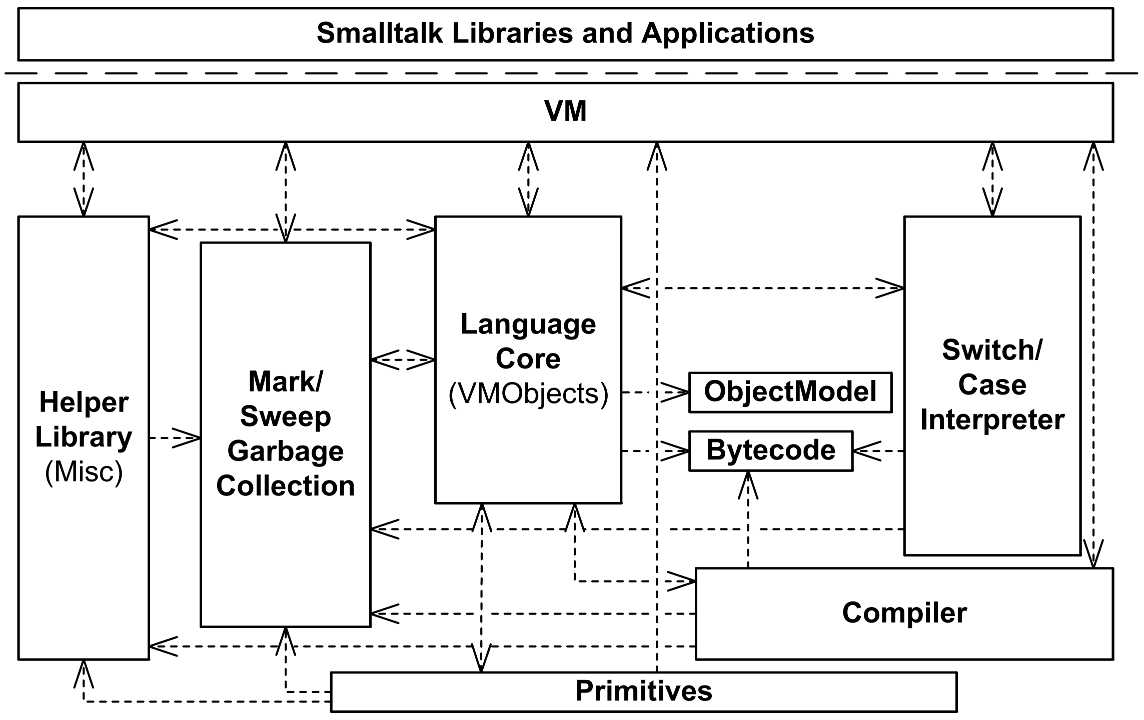

Smalltalk Libraries and Applications
VM
Helper Library (Misc)
Mark/ Sweep Garbage Collection
Language Core (VMObjects)
ObjectModel
Bytecode
Switch/ Case Interpreter
Compiler
Primitives

Figure 1: Architecture of CSOM.

Untypically, CSOM does not support images [18], but instead relies on text files containing Smalltalk code as input. The Smalltalk application to be run is passed as a command line parameter when the VM is started. If no application is given, the VM starts a Smalltalk shell.

The architecture of CSOM is deliberately simple to ease its employment in teaching. An overview about the architecture is given as block diagram in Fig. 1. The arrows between modules denote "uses" relationships. The standard implementation features a Smalltalk parser and compiler, a corresponding object model for representing Smalltalk entities, a simple bytecode interpreter, and a mark/sweep GC. The CSOM source code consists of 88 C files (43 `.c` and 45 `.h` files) accounting for 6,725 PSLOC [31] spread over seven logical modules represented by the folder structure of the implementation. The C implementation is accompanied by 568 lines of Smalltalk code in roughly two dozen files constituting its standard library. Additionally, a test suite and a set of benchmarks are available.

SOM, implemented in Java, exploited object-oriented programming (OOP) concepts to a large extent, using inheritance and interfaces. Also, the VM-level and language-level representations of core classes of the SOM Smalltalk standard library have parallel hierarchies. For instance, the Smalltalk implementation of the `Object` class is mirrored by a corresponding class on the VM side. The Smalltalk `Array` class inherits from `Object`, and so does the VM-level representation of Smalltalk arrays. This design is preserved in CSOM by using a macro-based emulation of OOP constructs in C. It supports single inheritance and a limited notion of traits [13], which is used to emulate Java's interfaces. Late binding is achieved by using a `SEND` macro to send messages and parameters to objects.

As already mentioned, CSOM has been used in teaching over the past few years. Several students have implemented extensions to CSOM to fulfil coursework assignments. So far, the following extensions have been developed.

Two alternative **multi-threading** approaches have been realised. *Native threading* uses the `pthreads` [25] library, whereas *green threading* implements scheduling and thread management within the VM itself.

For **memory management**, GCs applying mark/sweep and reference-counting [22] have been implemented. Since then, the mark/sweep GC is also part of the "standard" CSOM handed out to students.

As an **emulation engine optimisation**, a threaded interpreter [10] has been implemented. **Integer representation** was optimised using one-based tagged integers [18]. **Virtual images** [18], saving snapshots of application state, are common with Smalltalk and were provided for CSOM.

Each of the above was implemented as a stand-alone extension to CSOM, as the coursework groups were working separately. Hence, they were not concerned with clear modularisation and interoperability among the extensions. Thus, the implementations mentioned above are independent of each other, and represent custom-built products derived from a common code base.

The different extensions exhibit largely different crosscutting characteristics. For instance, mark/sweep and reference-counting GC both require the structural extension ("introduction" [23]) of adding a mark bit or reference count to objects. Behavioural crosscutting, however, is much different: while reference counting requires modifications at practically *all* pointer assignments (including parameter passing), mark/sweep GC is attached only to allocation requests. Another example is multi-threading. Native threading effectively requires the interpreter implementation to be thread-safe (i. e., the interpreter's global state must be turned into thread-local state). Conversely, green threading implies significant changes in the interpreter logic itself, as the interpreter is responsible for passing control to the scheduler, e. g., every N bytecode instructions.

All in all, the extensions realised so far constitute an interesting challenge with regard to modularisation. This holds even more when combinations of the aforementioned extensions are taken into account, e. g., a version of CSOM that features both a mark/sweep GC and native threading.

3 VIRTUAL MACHINE MODULARITY

In this section, we will first summarise the approach to VM modularisation [20] whose concepts VMADL implements. We will then give a tutorial on VMADL in its current state. It will include an extension to C that was necessary to allow for VMADL's application to the C programming language in the setting with CSOM. Finally, the VMADL implementation and tool chain will be briefly described.

3.1 Disentangling VM Architecture

In previous work [20], the architectures of different VM implementations were investigated. It was found that most of them exhibit no clear boundaries between subsystems perceivable as logical modules. The necessity of an architectural approach with support for reasoning about high-level modular structures in VM implementations was then motivated.

We would like to explain the notion of *architecture* that we adopt. There is no consensus on a definition for the terms "architecture" and "architectural description language" (ADL). A wide range of different interpretations of the terms [28] exists. On the one end of the spectrum, there are, e. g., graphical ADLs that enable an easier comprehension of system architectures to improve communication about systems. On the other, there are languages proposing formal semantics and tools for analyses, code synthesis, and run-time support, to allow for a formal evaluation of complex systems.

For VM implementations, system architecture needs to be supported at the source code level: architectural building blocks have no clear boundaries and are hence not cleanly modularisable. Consequently, modules and their interactions have to be described at a level that is close to the implementation language but still supports architectural abstraction in that it expresses larger-scale interdependencies. At the same time, the implementation language must not be constrained in its degree of control over low-level details.

The earlier introduced approach [20] modularises VM implementations into *service modules* with *bidirectional interfaces*. That is, a service module can not only be sent requests, but it can also signal internal situations of interest to the outside world. Other service modules can attach to these signals and react to them. These signals are called *exposed join points*, are defined using pointcuts, and constitute a module-specific join point model, elements of which can be quantified over by means of pointcuts.

To achieve these goals, VMADL provides a frame in which an implementation language and an aspect language can be combined. Consequently, VMADL is essentially agnostic as far as the particular implementation and aspect languages at work are concerned: it adds high-level modularity constructs that coordinate the interaction of the former two. The first VMADL proof of concept [20] was applied with C as the implementation language, and Aspicere2 [1] as the aspect language. In the present work, the implementation language is still C, but AspectC++ [30] is the aspect language.

3.2 VMADL: A Walkthrough

This overview of VMADL uses abbreviated actual code from the CSOM/PL implementation (cf. Sec. 4) to introduce the various features. A complete example of two service module definitions and their combination is given in App. A.

```
service Interpreter {
    void Interpreter_start(void);
}
service VMCore {
    void Universe_set_global(_VMSymbol*, _VMObject*);
}
service ObjectModel {}
service VMObjects {
    require ObjectModel;
    expose {
        pointcut initializer() = "void␣_VM%_init(...)";
    }
}
```

Listing 1: Service module definition in VMADL.

Lst. 1 introduces *service module definitions*. Four such modules are defined, and the `Interpreter` and `VMCore` module definitions demonstrate that the API is simply defined by declaring the corresponding C function. It is also possible to express that a given module is required by another, as in the `VMObjects` definition. The listing also demonstrates how *join point exposition* is achieved by using AspectC++ definitions: the `VMObjects` module exposes the `initializer` pointcut, which matches whenever a C function matching the name `_VM%_init` is executed. To express mandatory relationships between modules, the `require` statement is used. It ensures that the resulting configuration includes all mandatory service modules.

The specification of service module *interactions* is illustrated in Lst. 2. The construct used for this is called a *combiner*. The `combine` construct allows to implement module interactions at the same architectural level as service modules, but without touching module definitions. This separation enables developers to describe module interactions at a well-defined place in the source code. This allows an easier recognition of module relationships and dependencies. Furthermore, a combiner becomes part of the system only if all modules it refers to are part of the product configuration.

The first combiner in Lst. 2 avoids garbage collection during object initialisation by attaching an around advice to the `initializer` pointcut exposed from the `VMObjects` module. The second combiner shows how join point context information can be used in advice. It establishes management of a symbol table saved along with the Smalltalk virtual image.

When implementation languages such as C or C++ are used, the VM implementation most likely uses preprocessor macros. In the case of CSOM, whose implementation emulates object-orientation in C, macros are used to realise mes-

```
combine GCMarkSweep, VMObjects {
    advice execution(VMObjects::initializer())
     : around() {
        gc_start_uninterruptable_allocation();
        tjp->proceed();
        gc_end_uninterruptable_allocation();
    }
}
combine Image, VMCore {
    advice execution("void_Universe_set_global(...)")
     && args(name, value)
     : after (_VMSymbol* name, _VMObject* value) {
        // register key for symbol
        SEND(globals_dictionary_symbols,
                  addIfAbsent, name);
    }
}
```

Listing 2: Definition of service module interactions.

sage sending. The implementation of 1-based integer tagging (cf. Sec. 4.2.4) requires a redefinition of the SEND macro. As macros are not first-class values in the C programming language, some means for their redefinition was required.

```
service ObjectModel {
    SendMacro {
        #define SEND(O,M,...) ({ typeof(O) _O = (O); \
            (_O->_vtable->M(_O , ##__VA_ARGS__)); })
    }
}
service TaggedIntOne {
    #include <tagged-int-one/tagged-int-one.h>
    replace ObjectModel.SendMacro {
        #define SEND(O,M,...) ({
            typeof(O) _Org = (typeof(O))(O); \
            typeof(_Org) _O =
                (typeof(_Org))(INT_IS_TAGGED(_Org) ? \
            VMInteger_Global_Box() : _Org); \
            (_O->_vtable->M(_Org, ##__VA_ARGS__)); })
    }
}
```

Listing 3: Named section replacement.

Lst. 3 introduces the concept of *named sections*, i. e., parts of source code that can be referenced by name in VMADL. Named sections add structure to interface definitions and are used to support interface definition refinement. The listing shows how the SEND macro is defined in the SendMacro named section in the ObjectModel service module, and also its redefinition in the TaggedIntOne service module.

Since C is the implementation language, we had to introduce a language orthogonal to VMADL, which provides us with the necessary flexibility to describe crosscutting refinements of classes in our OOP emulation. Thus, we designed a small class definition language (ClassDL) as an add-on to C to be able to modularise the features of our VM product line completely. ClassDL is implemented as part of the VMADL tool chain used in our case study on CSOM.

ClassDL provides a C-like notation to define classes and traits according to CSOM's OOP emulation. ClassDL can be used to refine classes from other modules as well. From ClassDL definitions, the necessary implementation details like structure definitions for object layout and virtual method tables, including code for their initialisation, are generated. The ClassDL notation used to define fields conforms to field definitions in C structures. Respectively, method definitions conform to function declarations.

To support structural changes in VMADL service module combinations, an additional keyword was introduced to refine classes or traits from other modules. Within the scope of our case study, it was necessary to add methods and fields to existing classes due to the structural crosscutting exhibited by some features (cf. Sec. 2). For method introductions, simple definitions are given like in a normal class definition. As object layout must be controllable at a fine level of granularity—in particular, the order of fields in objects is important—, a field can be defined with an additional predicate specifying the position with respect to another field. These language constructs are sufficient to modularise the features under consideration.

```
service VMObjects {
    class VMObject {
        size_t    num_of_fields
        pVMObject fields[0]
    }
    trait VMInvokable : VMObject {
        pVMSymbol signature
        pVMClass  holder
        void      invoke(pVMFrame)
    }
    class VMArray : VMObject {}
    class VMMethod : VMArray, VMInvokable {
        pVMSymbol  signature
        pVMClass   holder
        bytecode_t get_bytecode(intptr_t)
        void       set_bytecode(intptr_t, bytecode_t)
        void       invoke_method(pVMFrame)
    }
}
service GCRefCount {
    refine VMObject {
        intptr_t   gc_field {before fields[0]}
    }
}
```

Listing 4: ClassDL object layout definitions.

Lst. 4 shows some ClassDL examples. It first presents how object layouts and interfaces for classes and traits defined in the `VMObjects` service module are specified.

It then shows how the reference-counting GC extends object layout in a controlled way by inserting the reference count field before the `fields` array responsible for storing actual object slots.

We would like to point out once more that ClassDL is entirely orthogonal to VMADL. Also, ClassDL is purely declarative: method implementations are not given. Its sole purpose is to help with the particularities of the OOP emulation approach chosen in CSOM. Had CSOM been implemented in a different language with dedicated support for object-oriented modularisation and heterogeneous crosscutting, ClassDL would not have been necessary.

A note on differences between the VMADL proof of concept [20] and the robust version of the language presented here is advisable. The proof of concept did *not* feature explicit combiners—service module interactions were defined in service module definitions themselves, prohibiting definitions of interaction facets when introducing new service modules. The proof of concept also lacked *named sections* and explicit module relationships expressed with `require`.

3.3 The VMADL Tool Chain

The VMADL compiler acts as a preprocessor for the actual implementation and aspect languages. It produces implementation files according to a given set of chosen service modules and identifies the required service module combinations automatically to provide the intended variability. The *product configuration* is given by passing the chosen service modules' names to the VMADL compiler. With this information available, the VMADL files are parsed, and AspectC++ `.ah` files as well as C `.h` files are generated, defining the function interfaces at the C language level. The ClassDL compiler also processes the respective `.c` files to generate initialisation code for the OOP emulation. It outputs a set of C header and implementation files as well, whose contents also depend on the chosen configuration and the actual implementation. The resulting files are processed by the AspectC++ and C++ compilers.

A VMADL file may contain a service module definition along with combiners, or just one or more combiners. Hence, it is possible to specify a newly arisen combination in a single new file without having to touch already existing ones.

The tool chain itself integrates into the CSOM build environment which is based on *make*. A sample invocation to generate a CSOM instance with mark/sweep GC, 1-based tagged integers and green threading looks as follows:
`./configure marksweep int-one green && make`.

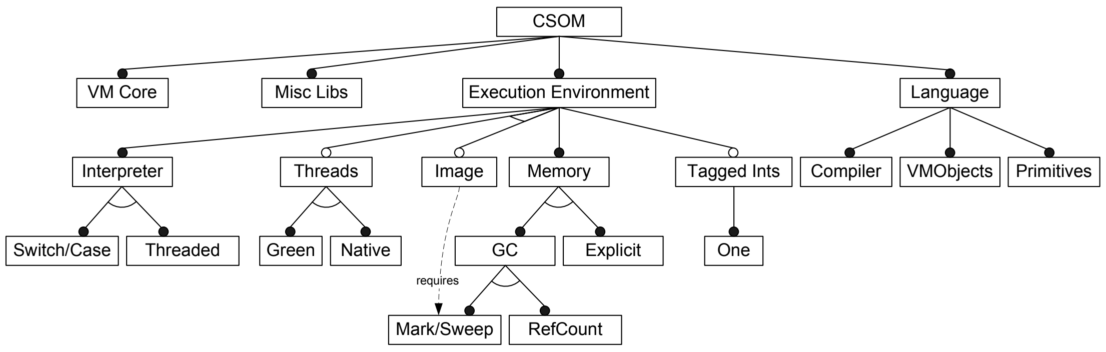


Figure 2: The CSOM/PL feature model as realised by the current implementation.

4 A VIRTUAL MACHINE PRODUCT LINE

This section presents CSOM/PL. The first part will discuss the product line's feature model, possible configurations, realisation using pure::variants, and overall benefits of our approach. Subsequently, we will discuss the language-level concepts used to modularise CSOM's features.

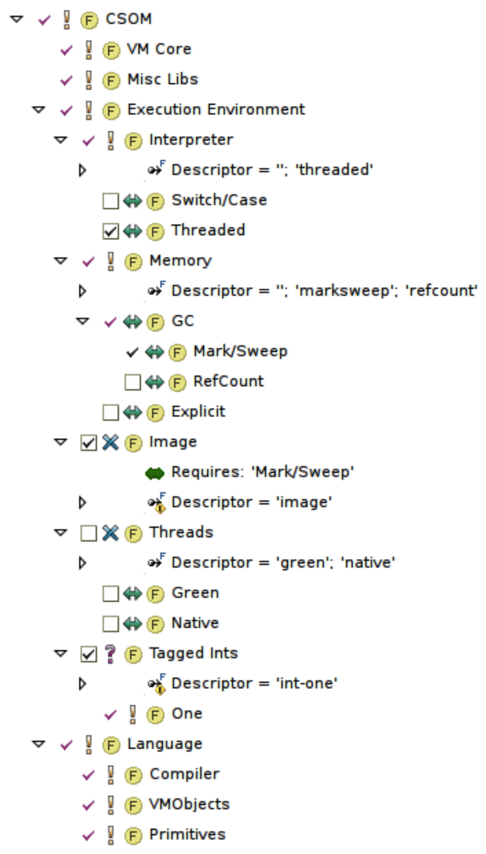
4.1 The CSOM/PL Feature Model

With the earlier proof of concept implementation of VMADL [20], it was possible to use the implementation of explicit memory management, mark/sweep and reference-counting garbage collection, and green as well as native threads for a case study. Each of these features was implemented as a mere add-on to CSOM; no combinations of features were provided. Some of the CSOM versions composed using the VMADL proof of concept were less robust than the CSOM extensions with the same features that were coded by hand.

In contrast, the present VMADL implementation enabled us to achieve significantly better results. On the one hand, we were able to use, in addition to the features mentioned above, implementations of Smalltalk virtual images, 1-based tagged integers, and threaded interpretation. On the other hand, feature combinations were achieved that were not even existent in hand-coded form before.

Fig. 2 shows a feature diagram representing the current status of CSOM/PL. Mandatory core assets of any given instance of the product line are a memory manager and an execution engine, i. e., interpreter. Both can be instantiated by using either of the options mentioned in Sec. 2.

Each of the 16 concrete features has been realised as a VMADL service module, and all of the achieved product line instances have been realised using VMADL combiners. The latter not only make it possible to create actual CSOM/PL products, but also make the architectural relationships between the features (cf. Fig. 1) explicit at the source code level. Based on the given module names (cf. Sec. 3.3), the VMADL compiler decides which interactions are actually required to instantiate a given product, and generates code only for those. More detailed descriptions of the various feature implementations are given below.



```
▽ ✓ ! Ⓕ CSOM
    ✓ ! Ⓕ VM Core
    ✓ ! Ⓕ Misc Libs
  ▽ ✓ ! Ⓕ Execution Environment
    ▽ ✓ ! Ⓕ Interpreter
      ▷     Descriptor = '': 'threaded'
        ☐ Ⓕ Switch/Case
        ☑ Ⓕ Threaded
    ▽ ✓ ! Ⓕ Memory
      ▷     Descriptor = '': 'marksweep': 'refcount'
      ▽ ✓ Ⓕ GC
          ✓ Ⓕ Mark/Sweep
          ☐ Ⓕ RefCount
        ☐ Ⓕ Explicit
    ▽ ☑ ✕ Ⓕ Image
            Requires: 'Mark/Sweep'
      ▷     Descriptor = 'image'
    ▽ ☐ ✕ Ⓕ Threads
      ▷     Descriptor = 'green': 'native'
        ☐ Ⓕ Green
        ☐ Ⓕ Native
    ▽ ☑ ? Ⓕ Tagged Ints
      ▷     Descriptor = 'int-one'
        ✓ ! Ⓕ One
  ▽ ✓ ! Ⓕ Language
    ✓ ! Ⓕ Compiler
    ✓ ! Ⓕ VMObjects
    ✓ ! Ⓕ Primitives
```

Figure 3: A concrete CSOM/PL configuration in pure::variants.

The feature diagram exhibits a constraint imposed on virtual images: they require combination with the mark/sweep GC as they are not compatible with reference counting. This is due to the *Image* feature's relying on all objects being laid out in a single contiguous memory area, which property explicit memory management and reference counting do not guarantee. The feature diagram also excludes combinations of virtual images with multi-threading. This is simply because the *Image* feature was not adapted for thread safety. Note that these combinations are, in principle, achievable but require some more implementation effort (cf. Sec. 7).

The feature model has been realised in software using the pure::variants tool, which features a model editor, product configurator, validity checker, and a rich generator infrastructure. Fig. 3 shows a screenshot from the product configuration view, where the entire feature model tree of CSOM/PL has been expanded. The selected configuration represents a CSOM VM with a threaded interpreter, mark-sweep GC, one-tagged integer representation, virtual images, and no multithreading support. The mark-sweep GC has been selected automatically by pure::variants as the *Image* feature was included in the product. From such a product configuration, the SPL tool generates a build script which is used by `make` to compile a CSOM/PL instance.

4.2 Feature and Product Implementations

We will now give brief examples how VMADL was used to implement the CSOM/PL features as service modules, and how those were combined to instantiate products. It is interesting to note that the CSOM "base implementation" did not have to be adapted to meet the needs of any of the extensions that were added. All combinations could be expressed using the abstraction capabilities of VMADL and the embedded aspect language, AspectC++. Throughout this section, we will only give brief examples. A more elaborate example is given in the Appendix. It shows the definition of the *native multi-threading* service module and its combination with the *interpreter* and *mark/sweep GC* modules. This combination was chosen because it significantly influences the involved service modules.

4.2.1 *Memory Management*

The three different service modules representing concrete memory management features each have a particular implementation of a common interface. The explicit memory management service module does not provide any additional functionality but relies on the interfaces offered by the *VMCore* and the *VMObjects* service modules.

```
1  service GCMarkSweep {
2    refine VMObject {
3      int gc_field { before fields[0] }
4    }
5  }
6  combine GCMarkSweep, VMObjects {
7    advice execution(VMObjects::initializer()) :
8    around() {
9      gc_start_uninterruptable_allocation();
10     tjp->proceed();
11     gc_end_uninterruptable_allocation();
12   }
13 }
```

Listing 5: Combine Mark/Sweep with VMObjects.

The implementation of the **mark/sweep GC** is almost as transparent as that of explicit memory management. Just a few parts are adapted in other service modules by refinement or combiners. An example is given in Lst. 5 for the introduction of a mark field into the `VMObject` by a **`refine`** statement of ClassDL. It inserts the mark field before the first field containing a member slot.

Furthermore, a combiner describes the interaction of the *mark/sweep GC* and the *VMObjects* service modules. It introduces a guard for object initialisation. This avoids dangling pointers resulting from only partially initialised objects which could be caused by a GC run during object creation. The corresponding code is shown in lines 6–13 in Lst. 5.

The modularisation of the **reference-counting GC** is, at first, quite similar. A ClassDL **`refine`** statement introduces a field for the reference count in the `VMObject` class of the *VMObjects* service module. Other than with mark/sweep GC, the nature of reference counting demands a high number of interactions with other modules, since almost every assignment of an object reference has to be tracked. Thus, combiners have to be defined for all service modules the reference-counting GC has to be used with. These combiners are typically straightforward. They increase the reference count of the new object before the actual execution and decrease the reference count of the old value afterwards.

4.2.2 *Multi-Threading*

From the modularisation perspective, **green threading** is quite undemanding. The service module interacts with the *primitives* service module to register primitives for the `Scheduler` class and with the *VMCore* service module to enable the shell to use threads as well. Another combiner is used to adapt the *interpreter* service module to signal when it reaches a safe point in execution to allow thread pre-emption as shown in Lst. 6.

```
service Interpreter {
  expose {
    pointcut safe_points() =
      execution("void␣send(...)");
  }
}
combine GreenThreads, Interpreter {
  advice Interpreter::safe_points() : before() {
    ++scheduler_return_count;
    Scheduler_insert_scheduler();
  }
}
```

Listing 6: Pre-emption for green threads.

Some additional combiners are necessary to support the combined usage of multi-threading with, e. g., the different GCs. For configurations using the *mark/sweep GC*, the combiner implements an extension to the GC's mark phase to add the `Scheduler`-internal list of available threads to the GC's root set. Were this not done, all but the currently running thread would not be regarded as live objects. The case is similar for reference counting: assignments to `Scheduler` data structures have to be handled like all other assignments to ensure reference counts are updated correctly.

For **native threads**, the changes are more fundamental than for green threads (cf. Sec. 2). The major task is to achieve thread-local execution of interpreters. This is achieved by adapting the global frame pointer to be a thread-local one. Since most service modules are implemented without global state, this adaptation need is very low.

The aforementioned assignment adjustments were done to enable the combination of the *native multi-threading* and *reference-counting GC* features. With *mark/sweep GC*, this is more challenging. The scheme that was implemented in the feature combination found in CSOM/PL is a stop-the-world solution [22]. This is implemented entirely inside a combiner and will therefore become part of an instance of the CSOM product line *only* if both service modules—mark/sweep GC and native threads—are chosen.

4.2.3 *Execution Engine*

Interaction with the two possible interpreters (switch/case and threading) is realised using the common *interpreter* service module interface. The **threaded interpreter** requires some interaction with other service modules; e. g., a combination with the *VMObjects* service module achieves the translation of method bytecodes into threaded code [10] after method assembly by the Smalltalk compiler. Bytecode index handling is also adapted. The original design implies a local bytecode index in every `VMFrame` object. For threaded interpretation, this needs to be changed, since it relies on a global pointer to the bytecode handler executed next.

4.2.4 *Integer Representation*

When integers are implemented as "ordinary" objects, i. e., *boxed* integers, there is no difference between sending a message to an `Integer` instance or to another object: the virtual method table (VMT) is accessed and the message implementation resolved. Conversely, tagged integers do not have a multiple-slot representation in memory, and do not reference a VMT. Instead, a global "surrogate object" exists via whose VMT messages sent to tagged integers are dispatched.

In CSOM, adopting this change is challenging because sending messages to objects is realised via C macros, which cannot normally be redefined. However, as VMADL features *named sections* (cf. Sec. 3.2), redefining the `SEND` macro infrastructure is done by providing a replacement for the corresponding named section in the *ObjectModel* service module.

For this case study, we used **one-based tagging** as in Smalltalk-80 [18].

4.2.5 *Image Persistence*

The Smalltalk **virtual images** implementation relies predominantly on the abilities of ClassDL to refine classes and add methods. This is used to update references after loading an existing image. The change in the startup process of the VM to load an image instead of initialising the VM from source files is done by simple adaptations of initialisation routines implemented with a service module combination.

5 EVALUATION

When turning a set of hand-crafted extensions to a base system into a cleanly encapsulated set of service modules forming an SPL, there are two points of view from which the results should be evaluated. First of all, it is important to assess the impact of modularisation and combination on performance. This is especially interesting in the domain of VM implementations, where performance is crucial. CSOM has a set of benchmarks (cf. Sec. 2) that can be used to evaluate the performance of hand-crafted extensions versus automatically combined products. The second perspective is that of code complexity. We have evaluated the source code by applying several source code metrics to it. Modularity improvements were also considered.

Below, we will elaborate on these assessments, and conclude the section with a brief discussion of our approach in the light of Parnas' modularity criteria [32], SPL development tool support, and how the two coincide in CSOM/PL.

5.1 Virtual Machine Performance

Performance measurements were run on a Dual Xeon PC (2.5 GHz clock rate, four cores overall) with 6 MB cache and 8 GB RAM. The operating system was Debian Linux 4.0r3 with a 64-bit kernel (version 2.6.18) and 32-bit user land. The used compilers were GNU C++ 4.1.2, and `ac++` 1.0pre4/`ag++` 0.7 for AspectC++. For compiling CSOM, standard compiler settings were used, the optimisation flag given was `-O3` (highest optimisation).

To obtain performance results, each benchmark was run ten times as a standalone application in a dedicated CSOM instance; the average running time was taken as the result. The time required to start up the VM is not contained in these values; they reflect the sheer time required to actually run the benchmarks. Measurements were run for pairs of CSOM versions with the same features. We compare the hand-crafted implementations with the corresponding CSOM/PL configuration generated from VMADL service modules. This allowed for assessing the performance impact of using VMADL on a single CSOM/PL product.

Performance measurement results are displayed in Fig. 4. For each CSOM configuration, the relative performance of the VMADL version to the hand-crafted build is shown. VMADL employment does not bring about severe impacts. Interestingly, half of the VMADL-generated products exhibit small improvements. This is because the compiler applied different optimisations in the presence of code constructs generated by AspectC++. We comment on the more significant penalties observed for the reference-counting GC, virtual images, and threaded interpretation products.

First of all, in case of the reference-counting GC, the fact that the VMADL implementation performs worse than the hand-crafted one is due to that the reference-counting service module is *more accurate* than the hand-crafted code. This is because the pointcuts used to quantify over pointer assignment join points are more robust than an approach where all pointer assignments have to be identified and

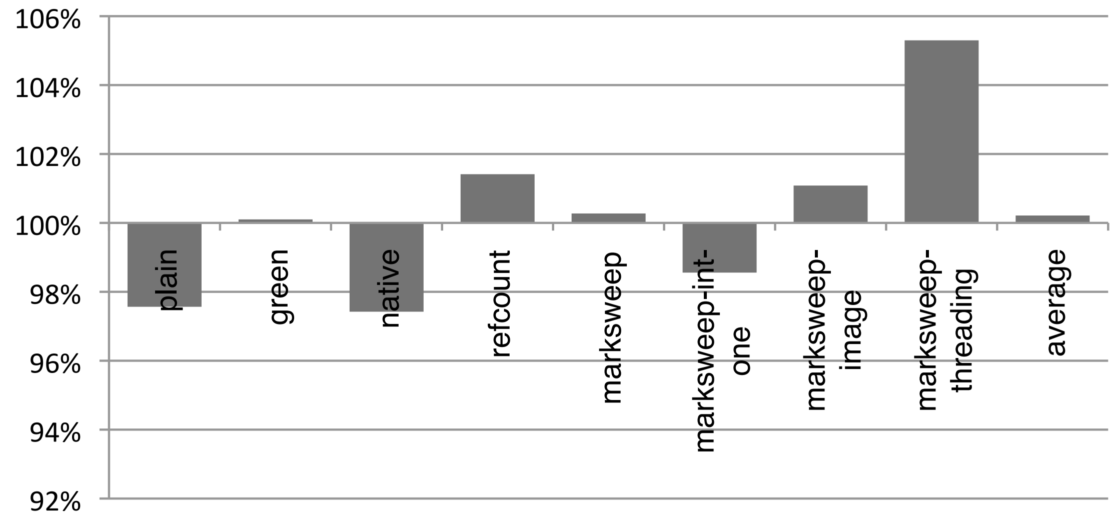


Figure 4: Performance measurement results: relative performance of VMADL versions to hand-crafted feature combinations.

instrumented by hand. Thus, the VMADL reference-counting GC implementation is slower just because it is correct.

For virtual images, the reason for the observed slight decrease in performance is due to differences in method localisation between the hand-crafted and the VMADL versions. The hand-crafted image feature had added various methods to different C source files. Since they belong to the *image* logical module, they were all collected in one place in the VMADL version, namely in the corresponding service module. That is, instead of being scattered over different compilation units, they are now found in a single one. The GNU C++ compiler does not support inter-module optimisation and thus fails to inline the methods from the single implementation file into the code where they are actually used. This leads to the observed penalty for the cleanly modularised implementation of the image feature with VMADL.

Threaded interpretation is the implementation where the implementation based on service modules exhibits the most significant penalty (about 5 %). This case in fact illustrates that VMADL requires a feature it is currently lacking: control over the order in which features apply. In this product, the mark/sweep GC and threaded interpretation features interact at certain points. Manual modification of the advice application order in the AspectC++ code generated from the VMADL preprocessor led to a performance impact of only 1.4 %. This remaining degradation is owed to the usage of `cflow` constructs [23] which could not be avoided.

The average performance over all considered CSOM/PL configurations is about 100.2 % compared to the hand-crafted versions. So, even though there can be performance penalties caused by the usage of aspect-oriented means, overall performance is not influenced when applying VMADL.

5.2 Source Code Complexity

To assess source code complexity, several metrics were applied. Notice that only the *actual* CSOM/PL source code was taken into account. For some of the extensions, e. g., multi-threading, the Smalltalk libraries had to be extended as well to provide APIs for the new features. As those do not belong to the VM *as such*, they were not regarded.

The metrics were applied to the entire corpus of CSOM source code including *all* extensions. Hand-crafted versions represented reference values, to which results obtained from service module source code were put in relation.

5.2.1 *Lines of Code Results*

The *physical source lines of code* (PSLOC) [31] metrics counts all lines of code that are not empty and do not solely consist of comments. Applying VMADL resulted in a moderate increase of 2.1 % (143 lines) in PSLOC. It needs to be noted that this value is adjusted as it does *not* account for the effect of also using ClassDL. The employment of ClassDL resulted in a *decrease* of 738 PSLOC. As we want to assess the sheer effect of VMADL, we elided ClassDL's effect.

5.2.2 *Modularity Results*

To determine feature locality and modularisation, we identified changed implementation modules as well as modified *functions and structures* at the sub-module level. CSOM's base configuration was compared to others with reference-counting and mark/sweep GCs, and green as well as native threading. Moreover, CSOM with mark/sweep GC was compared to configurations with virtual images, 1-based integer tagging, and threaded interpretation. That way, it was possible to determine, by feature, how many lines of code were added or modified, and how many files and function or structure definitions were affected. The results allow a comparison of hand-crafted and VMADL implementations.

Results from the modularity metrics are shown in Fig. 5. The shown values are cumulative: they represent the sums of the respective metrics from measuring them for *all* of the analysed feature implementations. In the figure, the left-hand *y* axis relates to the dotted large bars in the background; the right-hand *y* axis, to the bars in the foreground. The background bars, "new lines" represent the total number of added lines of code. The foreground bars present additional details, namely the number of lines that had to be removed, number of newly added files, changed files, and changed function or structure definitions. The "changed lines"

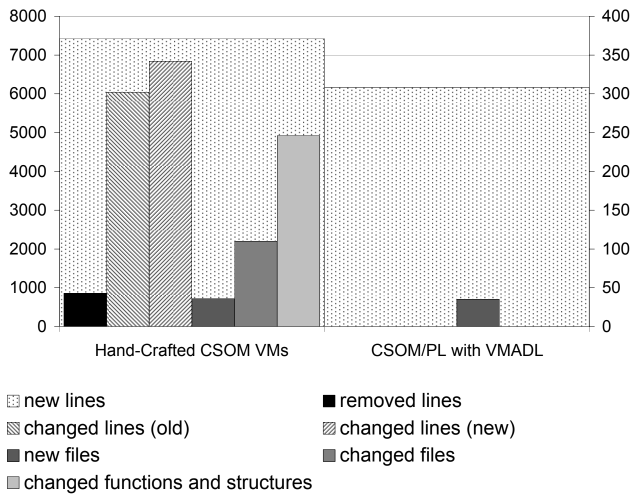


Figure 5: Modularity metrics results. Left y axis (background bars): number of new lines; right y axis (foreground bars): number of added/changed lines/files/functions/structures.

numbers express how many lines were *modified* in the transition in a way reported by a common `diff` tool.

The figure clearly shows that hand-crafted implementations exhibit a larger (more than 1,000 lines) implementation overhead. Since we did not change the inner modularisation of service modules, the number of added files remains constant, while eliminating crosscutting changes and thus reducing implementation overhead. Thus, the VMADL approach yields excellent modularity: while handcrafting involves a large number of *modifications* in existing code, using VMADL and service modules merely implies introducing new files, which contain all of the newly introduced code.

In a nutshell, this means that VMADL supports *real* modularity: extensions are not invasive in any way; they are completely encapsulated in dedicated files, which in turn results in a unified source base for the whole CSOM/PL.

5.3 Discussion and Conclusion

D. L. Parnas has formulated [32] a set of three criteria for modular programming. In short, an approach can be called "modular" if "separate groups [can] work on each module with little need for communication", "drastic changes [can be made] to one module without a need to change others", and "it [is] possible to study the system one module at a time". More recent elaborations on modularity, such as those by Meyer [29], add more detail to these criteria but basically imply the same. We will now briefly discuss our approach to modularising VM implementations along these criteria.

As *interfaces* are at the core of the VMADL approach, the benefits usually brought about by them are also benefits of VMADL. Communication among development groups can take place in terms of the interfaces of service modules. As long as changes to modules do not affect their interfaces, other modules do not have to be changed. When semantically meaningful sets of join points are exposed and given appropriate names (cf. Lst. 1), it is even possible to change their definition (i. e., pointcut) without having to change a client. In fact, it might even be the case that the details of a module interaction change, but the modules themselves do not have to be modified as the interaction is specified in a VMADL *combiner*. Studying a system as complex as a VM one module at a time is usually hard. Having a clear separation of the various services into distinct modules helps in this process as it clarifies module boundaries.

Utilisation of SPL tool support in CSOM/PL is less extensive than it might be expected: pure::variants is only used for feature model representation, product configuration and validation, but not for generating large amounts of source code required to build the product. Instead, there exist various cleanly separated modules with explicit bidirectional interfaces mapping directly to product line features.

From the results in the three different areas of interest described above, and from the considerations on Parnas' modularity criteria, we conclude that using VMADL is fruitful. Its employment has no negative impact on performance. It supports actual modularity. The lines of code count increases slightly, but for the greater good of module interdependencies' being made explicit in the code. The subjective impression of developers is that VMADL makes working with the CSOM source code more comfortable.

Finally, the strong modular characteristics of the product family code base enabled by VMADL result in an excellent direct mappability of product configurations to source code. This, in turn, significantly reduces the effort required to establish a collection of code fragments as input to the complex SPL generator infrastructure.

6 RELATED WORK

In the field of VM implementations, various projects have attempted to tackle the large complexity that is typical of the domain. Still, the strong focus on both architecture and modularity that we have adopted has not been chosen by any of them. Hence, the results from these projects do, however significant in their own right, not bring about the same improvements in terms of modularity and architecture perception at the source code level as ours.

The PyPy project [34] focuses on tool-chain based VM development. The core idea is to swap out implementation complexity to dedicated tools that are applied at certain times during VM code generation. The PyPy VM is implemented in Python at a very high level, allowing developers to use the object-oriented abstraction and dynamic language mechanisms that Python offers. Implementation takes place without regarding the fact that the ultimate VM will have a GC component. The GC is added later automatically during code transformation steps of the tool chain.

PyPy thus hides complexity away, easing development significantly. While this is appreciable, we do not agree with the idea of ignoring the presence of certain features. It is our goal to give VM developers full control over all features at the same level of abstraction. VMADL supports this approach by providing bidirectional interfaces and combiners that allow for dealing with complex interdependencies.

Metacircular VM implementations generally benefit from the modularisation techniques offered by the implemented language directly. The Jikes RVM [5, 6] and Maxine[8] are Java VMs implemented in Java.

Jikes is a magnificent platform for VM implementation research and supports a wide variety of choices among, e. g., GC implementations and JIT compilers. It makes use of code generation [9] to complete Java source file stubs for various features prior to compile-time. Memory management is performed by MMTk [11], which encapsulates GC complexity, but introduces hardwired interactions between GC logic and the VM, in either code base, leading to the kind of crosscutting concerns typical for the VM implementation domain. Opposed to use code generation for variability, To summarise, Jikes realises variability by code generation, as opposed to VMADL, which achieves the same using declarative means at the programming language level. Also, Jikes does not support disentangling the way VMADL does.

Maxine tries to improve modularity with the language features [10] offered by Java 5. Interfaces are used to encapsulate features, and a build-time configuration mechanism decides about feature implementations. Even though all feature interactions are done via interfaces, dependencies between feature implementations are not as obvious as with VMADL, since interactions are still scattered. Furthermore, the implementation does not achieve modularisation at the same degree as it would be possible with MDSOC techniques.

ClassDL was inspired by feature-oriented programming [33, 8]. Refining a previously defined class in the context of a specific feature effectively supports heterogeneous crosscuts. VMADL thus combines aspect- and feature-oriented approaches [9] in a more architecture-aware manner.

VMKit [17] is called a "substrate" for implementing VMs. It provides a common foundation that implementations of different instruction sets and programming

[8] `research.sun.com/projects/maxine`

[9] `jikesrvm.org/Building+the+RVM`

[10] `java.sun.com/developer/technicalArticles/releases/j2se15langfeat/`

languages can build upon. The substrate includes memory and thread managers as well as a JIT compiler. Implementing a VM on top of it involves providing certain callbacks to the substrate, and mappings from ISA or programming language constructs to the substrate's abstractions. VM implementation is thus significantly simplified.

While VMKit supports variability to the extent that implementation of different languages is simplified in this environment, it is restricted in the choices it offers at the substrate level. For instance, LLVM [24] is used as the JIT compiler infrastructure, and MMTk [11] as the memory manager. The latter provides particularly good variability, but the overall degree of control over feature variation is coarse-grained, compared to our approach.

Compared to other ADLs, VMADL is most closely related to ArchJava [3]. Like ArchJava, VMADL makes system architecture explicit in the source code itself. Other languages like WRIGHT [4] or Rapide [27] separate architecture description from actual implementation, which is problematic, since it implies the need to keep both synchronised. VMADL's bidirectional interfaces are related to principles found in nesC [16], an extension to C designed to structure systems into components with clear boundaries. The interfaces used in nesC declaratively describe component interactions using events and callbacks.

In contrast to VMADL, ArchJava assumes a dynamic architecture and multiple instances of a component at run-time. Components provide communication ports, and connections are explicit. This idea is similar to VMADL combiners but provides lower flexibility, since connections need to be explicit in component implementations. VMADL's combiners support module combinations at the interface level without changing their implementations. By using a pointcut language, our approach is more flexible.

MDSOC techniques offer various opportunities for building SPLs. Alves et al. [7] describe a methodology which uses aspect-oriented techniques to extract an SPL from an existing code base. The approach is similar to what we have done to create CSOM/PL on the basis of the different extensions available. Compared to it, we do not use additional aspects for the evolution to be able to add new products to the SPL, instead we chose to bring the adaption to an architectural level and describe it by means of module interaction.

One of the application areas of FeatureC++ [8] is the implementation of SPLs [35]. It regards feature composition as *refinement* of a basis implementation. Feature modules are represented as (aspectual) mixin layers defining such refinements. Conversely, VMADL service modules are complete implementations of features that are composed with others by connecting interfaces. VMADL does not as much regard the single *features* as crosscutting concerns as their *orchestration*, which it makes explicit in combiners.

Figueiredo et al. [15] investigated the influences on stability as an important SPL property. Their results suggest that SPLs decomposed with AOP are more stable regarding adaptions in optional or alternative features. We assume similar benefits for an SPL built with VMADL.

VMADL, in its current version, expresses explicit interactions between service modules. Some languages for modelling variability at an architectural level include constructs to model other types of relationships as well. One example in the field of product lines is the *Variability Modelling Language* [26]. This language is meant to be used on a more conceptual level and not embedded into the implementation. It aims to describe variability orthogonally to architectural descriptions. This approach would be beneficial to describe, for instance, service modules as alternatives. In addition to the variability already described with VMADL, some of the concepts of this language could be used to provide advanced means for the configuration of instances of CSOM/PL.

7 SUMMARY AND FUTURE WORK

We have presented CSOM/PL, a virtual machine product line implemented in C, AspectC++, and VMADL, and realised with pure::variants. VMADL, an implementation of which is one of this work's contributions, surpasses previously achieved modularisation in the VM implementation domain. It supports modular abstraction by means of service modules with bidirectional interfaces. Using VMADL allowed us to implement various of the features of the CSOM Smalltalk VM in *combinable isolated modules*—features that were previously realised as hand-crafted extensions. This also facilitated devising a product line. The evaluation of the approach shows that performance is not harmed, and that modularity in source code is significantly improved.

Regarding SPL tool support, VMADL represents a valuable tool that can be used to introduce modular SPL development in languages like C that do not inherently support modularity. Moreover, VMADL supports direct mappings from feature models to source modules, reducing the complexity of code preparation for consumption by generators.

Some perceivable feature combinations have not been realised yet (cf. Sec. 4.1). This is not because they are impossible to achieve; it is a matter of providing more service module interfaces and combinations to make them work. Our ongoing work is concerned with moving towards the goal of dropping all constraints shown in Fig. 2, which are not conceptually necessary. For instance, threading together with virtual images could be realised as well as virtual images independent from a particular garbage collection technique.

Performance measurements have shown that fine-grained control over feature application order is important. We will investigate how to make such control available in VMADL declarations without introducing uncalled-for complexity.

The ClassDL extension was necessary because CSOM, including its particular OOP emulation, is implemented in C, and because this led to a lack of declarative means for class (re)definitions. An implementation in C++, combined with AspectC++ and/or FeatureC++, would have eliminated this need. In fact, a port of CSOM to C++ is in progress, and will allow to use VMADL directly with AspectC++. The port will be an important cornerstone of future research in disentangling VM architecture.

In addition to our ongoing activities to evaluate, adjust and extend the CSOM VM in our research and teaching activities, we hope to transfer our results to other, more complex, VM implementations to gain more insights into the modularisation of full-scale VM implementations.

8 ACKNOWLEDGMENTS

The authors are grateful for the contributions of Bram Adams, Yvonne Coady, Celina Gibbs, and Stijn Timbermont to the ideas underlying the presented results. Special thanks go to Tobias Pape for his contributions to CSOM.

REFERENCES

[1] B. Adams and K. D. Schutter. An aspect for idiom-based exception handling: (using local continuation join points, join point properties, annotations and type parameters). In *Proc. SPLAT'07*. ACM, 2007.

[2] J. Aldrich. Open modules: Modular reasoning about advice. In *Proc. ECOOP'05*, volume 3586 of *LNCS*, pages 144–168. Springer, 2005.

[3] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *Proc. ICSE'02*, pages 187–197. ACM, 2002.

[4] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[5] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *Proc. OOPSLA'99*. ACM Press, 1999.

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

[7] V. Alves, P. Matos, L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho. Extracting and evolving code in product lines with aspect-oriented programming. In *Transactions on Aspect-Oriented Software Development IV*, volume 4640 of *LNCS*, pages 117–142. Springer, 2007.

[8] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proc. GPCE*, 2005.

[9] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: Aspects and features in concert. In *Proc. ICSE'06*. ACM, May 2006.

[10] J. R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[11] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *Proc. ICSE'07*, 2004.

[12] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Proc. Java Grande'99*, pages 129–141. ACM Press, 1999.

[13] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. *ACM SIGPLAN Notices*, 24(10):49–70, 1989.

[14] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[15] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. C. Filho, and F. Dantas. Evolving software product lines with aspects: An empirical study on design stability. In *Proc. ICSE'08*. ACM, 2008.

[16] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The *nesC* language: A holistic approach to networked embedded systems. In *Proc. PLDI'03*, pages 1–11. ACM, May 2003.

[17] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. VMKit: a Substrate for Managed Runtime Environments. In *Proceedings of VEE*. ACM Press, 2010.

[18] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[19] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.

[20] M. Haupt, B. Adams, S. Timbermont, C. Gibbs, Y. Coady, and R. Hirschfeld. Disentangling Virtual Machine Architecture. *IET Journal Special Issue on Domain-Specific Aspect Languages*, 3(3), June 2009.

[21] M. Haupt, R. Hirschfeld, T. Pape, G. Gabrysiak, S. Marr, A. Bergmann, A. Heise, M. Kleine, and R. Krahn. The SOM Family: Virtual Machines for Teaching and Research. In *Proceedings of the 15th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM Press, 2010.

[22] R. Jones and R. Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP'01*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[24] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2004.

[25] B. Lewis and D. J. Berg. *Threads Primer. A Guide to Multithreaded Programming*. Prentice Hall, 1996.

[26] N. Loughran, P. Sánchez, A. Garcia, and L. Fuentes. *Language Support for Managing Variability in Architectural Models*, volume 4954 of *LNCS*, pages 36–51. Springer, 2008.

[27] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[28] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE TSE*, 26(1):70–93, 2000.

[29] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[30] O. Spinczyk and A. Gal and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proc. TOOLS Pacific'02*. ACM, 2002.

[31] R. E. Park. Software size measurement: A framework for counting source statements. Technical Report CMU/SEI-92-TR- 20, ESC-TR-92-20, Software Engineering Institute, Carnegie Mellon University, September 1992.

[32] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[33] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. *LNCS*, 1241:419–434, 1997.

[34] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *Proc. OOPSLA'06*, pages 944–953. ACM, 2006.

[35] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Code generation to support static and dynamic composition of software product lines. In *Proc. GPCE'08*, pages 3–12. ACM, 2008.

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

A VMADL EXAMPLE

Note: the code displayed here was abbreviated. Irrelevant parts are not shown.

```
service NativeThreads {
    require Memory;
    require VM;
    require VMObjects;
    require Interpreter;
```

required interfaces from other service modules

```
    #include <pthread.h>
```

here starts the definition of the *NativeThreads* service module interface, including ClassDL definitions

```
    extern pthread_key_t tsg_frame, tsg_thread;
    pVMMutex  VMMutex_new(void);
    void*     VMThread_get_safe_global(pthread_key_t);
    void      VMThread_set_safe_global(pthread_key_t, void*);
    class VMMutex : VMObject {
        pthread_mutex_t  embedded_mutex_id
        pthread_mutex_t* get_embedded_mutex_id()
        void             lock()
        void             unlock()
        bool             is_locked()
    }
    class VMSignal : VMObject { ... }
    class VMThread : VMObject { ... }
}
```

the interpreter needs to be executed thread-locally; thus, its global state variables have to be made thread-safe

```
combine NativeThreads, Interpreter {
    advice execution("void Interpreter_set_frame(...)") && args(value) : around(_VMFrame* value) {
        VMThread_set_safe_global(tsg_frame, value);
    }
    advice execution("_VMFrame* Interpreter_get_frame()") : around() {
        pVMFrame frame = (pVMFrame)VMThread_get_safe_global(tsg_frame);
        *tjp->result() = frame;
    }
}
```

the *stop-the-world* GC scheme is implemented entirely in this service module combination

```
combine NativeThreads, GCMarkSweep {
    require Interpreter;
    #include <pthread.h>
    bool            stop_the_world;
    pthread_mutex_t mtx_do_collect;
    pthread_mutex_t mtx_gc_structure;
    advice execution("void gc_collect()") : around() {
        if (pthread_mutex_trylock(&mtx_do_collect) == 0) {
            stop_the_world = true;
            wait_for_all_threads();
            tjp->proceed();
            signal_proceed_to_all_threads();
            pthread_mutex_unlock(&mtx_do_collect);
        }
    }
```

this advice guarantees *stop-the-world* semantics:
– try to acquire a lock; if this fails, a GC run has already been requested in another thread
– when the lock was acquired, signal all threads and wait until they have stopped at a safe point; then proceed with the collection
– finally, signal all threads to continue

```
    advice Interpreter::safe_point_in_execution() : before() {
        if (stop_the_world) {
            gc_mark_reachable_stack_objects();
            wait_until_gc_completed();
        }
    }
```

safe point: suspend thread execution, mark all stack objects, and wait for the signal to continue

```
    advice call("% pthread_exit(...)") : before() { dec_thread_count(); }
    advice call("% pthread_create(...)") : before() { inc_thread_count(); }
    advice execution("void Universe_exit(int)") : before() { signal_exit_to_gc_thread(); }
    advice GCMarkSweep::reserve_and_get_entry() : around() {
        pthread_mutex_lock(&mtx_gc_structure);
        tjp->proceed();
        pthread_mutex_unlock(&mtx_gc_structure);
    }
    advice GCMarkSweep::split_and_reserve_entry() : around() {
        pthread_mutex_lock(&mtx_gc_structure);
        tjp->proceed();
        pthread_mutex_unlock(&mtx_gc_structure);
    }
}
```

management: counting threads, and ensuring GC structures are thread-safe

COLOPHON

This report was typeset by LaTeX 2ε with pdf/ε-TeX using KOMA-Script. The body text is set 12/15 pt on a 36.2 pc measure. The body type face is *Palatino* by Hermann Zapf, brought to TeX as Type 1 PostScript font *URW Palladio L*. The listing type face is *Bera Mono*, based on the *Vera* family by Bitstream, Inc.; Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr. *—Tobias Pape*

ISBN 978-3-86956-134-9
ISSN 1613-5652