Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

CSOM/PL
A Virtual Machine Product Line

Michael Haupt[a] Stefan Marr[b] Robert Hirschfeld[c]

a. Oracle Labs, Potsdam, Germany
http://labs.oracle.com/
(work performed while at Hasso Plattner Institute)

b. Software Languages Lab, Vrije Universiteit Brussel, Belgium
http://soft.vub.ac.be/

c. Software Architecture Group, Hasso Plattner Institute, University of Potsdam, Germany
http://www.hpi.uni-potsdam.de/swa/

Abstract 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.

Keywords Virtual machines, architecture, software product lines, multi-dimensional separation of concerns

1 Introduction

Implementors working on high-level language virtual machines (VMs) [SN05] typically face the characteristic problem of intricately intertwined module dependencies. 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, Oracle's HotSpot JVM features two different versions[1] optimised for client- or server-specific applications, which use different just-in-time (JIT) compilers. The configuration is chosen at VM startup time. The Jikes RVM[2] [AAB+99, A+00] can employ a selection of two different JIT compilers that can moreover be combined with an adaptively optimising infrastructure [BCF+99].

Other dimensions of interest are, e. g., memory allocation behaviour, calling for different choices of garbage collectors (GCs) [JL96, BCM04]; 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 [HAT+09].

Previous work [HAT+09] introduced the notion of *service modules* to address module entangling in VMs. A service module is a module with a *bidirectional interface*—in the fashion of open modules [Ald05] or XPIs [GSS+06]—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 *features* in the sense of a software product line (SPL) [CN02]. SPL development organises the different shapes of a software system in a particular domain along the lines of a *feature model.* A feature model comprises the possible variations in the shapes of the software system's features in a formalised way, e. g., as a *feature diagram.* Such a model represents all members of the resulting *product family.* Concrete realisations of the software system, according to choices made for the different possible variations, are called *products.* The formalisation allows for validating configuration choices, and for rejecting conflicting configurations.

This article 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] [HHP+10], a VM for a Smalltalk [GR83] dialect. CSOM is primarily intended for use in teaching, and consequently focuses on understandability and clarity. It is moderately complex, featuring a bytecode interpreter and a mark/sweep GC [JL96]. Despite its simplicity, CSOM exhibits characteristic crosscutting concerns [HAT+09]; increasingly so when extended with additional or alternative features.

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 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

[1] `java.sun.com/products/hotspot/whitepaper.html`
[2] `jikesrvm.org`
[3] `www.hpi.uni-potsdam.de/swa/projects/som`
[4] The code and live CD with CSOM/PL are available at `www.hpi.uni-potsdam.de/swa/projects/som/`. Due to license regulations, pure::variants cannot be included with the CD image.

significantly from the proof of concept [HAT+09] 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] [O. 02], 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. In Sec. 3, we sketch the architectural principles at work in CSOM/PL, and give an introduction to the language VMADL, including a 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] [HHP+10] 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 done at the Hasso Plattner Institute. There, CSOM has been used in two graduate courses on virtual machines in 2007 and 2008.

Unlike most Smalltalk VMs, CSOM does not support images [GR83], 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 bytecode interpreter, and a mark/sweep GC. The helper library contains dedicated implementations of data structures used throughout the VM.

The CSOM source code consists of 88 C files (43 `.c` and 45 `.h` files) accounting for 6,725 PSLOC [Par92] spread over seven logical modules represented by the folder structure of the implementation. The C implementation is accompanied by 568 lines of

[5] `www.aspectc.org`
[6] `www.pure-systems.com/pure_variants.49.0.html`
[7] Pronounced "*see*-som".

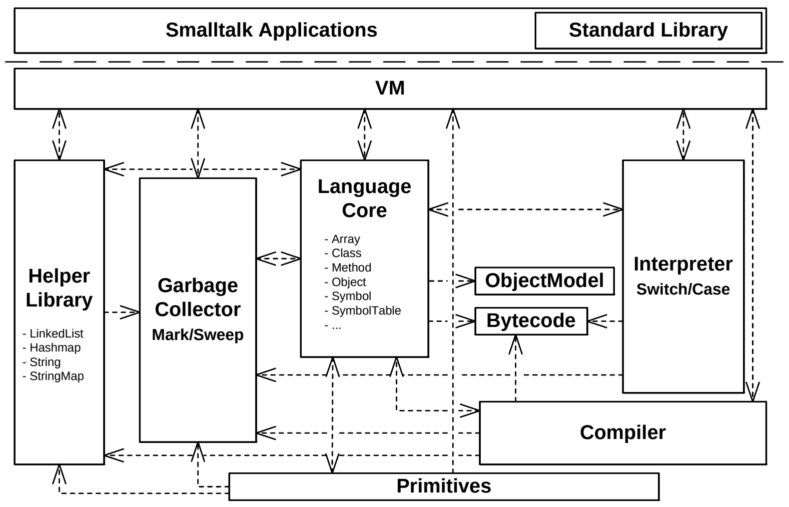


Figure 1 – Architecture of CSOM.

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, exploits 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 [CUL89], which is used to emulate Java's interfaces. Late binding is achieved by using a `SEND` macro to send messages and parameters to objects. C was preferred over C++ because it provides full low-level control over object layout in memory, including virtual method table placement.

As already mentioned, CSOM has been used in teaching over the past few years. Students have implemented extensions to CSOM to fulfil coursework assignments. Two alternative *multi-threading* approaches have been realised. *Native threading* uses the `pthreads` [LB96] library, whereas *green threading* implements scheduling and thread management within the VM itself. For *memory management*, GCs applying mark/sweep and reference counting [JL96] have been implemented. As an *emulation engine optimisation*, a threaded interpreter [Bel73] has been implemented. *Integer representation* was optimised using one-based tagged integers [GR83]. *Virtual images* [GR83], saving snapshots of application state, were provided for CSOM.

Each of the above was implemented as a stand-alone extension to CSOM. The separate coursework groups were not concerned with clear modularisation and interoperability among the extensions. The implementations 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" [KHH$^+$01]) of adding a mark bit or reference count to objects—an extension that is external to the actual GC logic. Behavioural crosscutting, however, is much different: while reference counting requires modifications at practically *all* pointer assignments throughout the VM implementation (including implicit ones like 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 first summarise the approach to VM modularisation [HAT+09] whose concepts VMADL implements. We then give an overview of the concrete language mechanisms necessary to realise the approach. Following some examples, we discuss the evolution of VMADL from the previous version, motivating its recent language features. Finally, we describe the VMADL implementation.

3.1 Disentangling VM Architecture

Previous work [HAT+09] investigated the architectures of different VM implementations, finding that they typically exhibit no clear boundaries between subsystems perceivable as logical modules. This insight motivated the necessity of an architectural approach with support for reasoning about high-level modular structures in VM implementations, and led to the proposal fo a first version of VMADL, which this article extends.

First, 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 [MT00] 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.

VM architecture needs to be supported at the source code level. 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.

VMADL modularises VM implementations into *service modules*, introducing a logical module structure into the code. Each service module can span several files containing source code contributing to the feature the module defines. Service modules have *bidirectional interfaces*, i. e., they can not only be sent requests, but 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*. They are defined using pointcuts, i. e., they can express complex internal situations whose occurrence can trigger reactions in client modules attaching to the exposed interface. This exposed interface constitutes 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 are concerned: it adds high-level modularity constructs that coordinate the interaction of the former two. The first VMADL proof of concept [HAT$^+$09] was applied with C as the implementation language, and Aspicere2 [AS07] as the aspect language. In the present work, the implementation language is still C, but AspectC++ [O. 02] is the aspect language.

3.2 VMADL Language Concepts

We now give an overview of the core mechanisms VMADL provides. As already mentioned, the main concept in VMADL is a *service module*. Each service module defines its bidirectional interface using the following constructs:

- **Type definitions:** Each service module defines a set of types or data structures, which can be used by other service modules or in module interactions.
- **Function definitions:** Each service module explicitly exposes a set of functions constituting its API.
- **Join-point definitions:** Each service module defines a set of internal situations that are exposed as join points to other modules. These are tightly coupled to the implementation of a service module, but abstract away from the concrete implementation, so that client modules are decoupled.
- **Exposed global variables:** Service modules define global variables that need to be exposed to other modules in their interface definition.
- **Module dependencies:** Required service modules are stated explicitly.
- **Structured interface and refinement:** Service module interfaces provide structured named sections which allow later cross-cutting refinement (cf. Sec. 3.3).
- **Startup and shutdown phases:** In complex systems like VMs, the phases of initialisation and termination introduce strong coupling and ordering requirements. To facilitate expressing those, service modules can provide specific functions and join points that are only accessible during these phases.

An architectural description also needs to define service module interactions. VMADL's *combiners* provide architectural-level implementation fragments describing all interactions that exceed API-like usage. Service modules depending on another service module can use the interface of that module in their implementation. Other forms of interactions, like advising join points, are expressed using combiners. They provide the following constructs:

- **Combined pairs of service modules:** A combiner always describes the interaction between a specific pair of modules which are explicitly named.
- **Module dependencies:** If combiners require other service modules to enable the interaction of the given pair of modules, these are stated explicitly.

- **Advice:** A combiner specifies advice to join points exposed by other service modules.
- **Startup and shutdown phases:** Similar to the service modules themselves, interactions can be specified that are only relevant during initialisation and termination phases.

Since VMADL is explicitly meant to be used on top of a set of implementation languages, these languages themselves need to fulfil some requirements to enable the desired degree of modularisation. The requirements are as follows:

- **Data type refinement:** Feature combinations in SPL development may require the definition of feature-specific, i. e., service module-specific, changes to types defined by particular modules. These are expressed using the underlying implementation language as part of the type definitions of a service module.
- **Control over object layout:** Specific to VMs is the requirement to have precise control over the layout of data structures in memory. Mechanisms like GCs need to be able to make certain assumptions about this, requiring modules to be able to put constraints on their refinements of data types.

3.3 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.

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. `VMObjects` declares a dependency on `ObjectModel` using the `require` keyword. 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. Note that `require` is used to express mandatory relationships between modules explicitly. This ensures that the resulting configuration includes all mandatory service modules.

```
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.

The specification of service module *interactions* is illustrated in Lst. 2. A *combiner* allows implementing module interactions at the same architectural level as service modules, but without requiring a direct modification of the module definitions. This separation enables developers to describe module interactions at a well-defined place in the source code, which in return enables 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.

```
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.

The first combiner in Lst. 2 avoids GC runs during object initialisation by attaching an around advice to the `initializer` join point exposed by the `VMObjects` module. The second shows an advice using join point context information. 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. From a feature-oriented perspective, preprocessor macros are problematic, since they cannot be refined in a composable way. However, their use cannot always be avoided. In the case of CSOM, whose implementation emulates object-orientation in C, macros are used to realise message sending. The implementation of 1-based integer tagging (cf. Sec. 4.2) requires a redefinition of the `SEND` macro.

To enable some form of refinement, we use *named sections* as demonstrated in Lst. 3. While named sections are meant to structure and document interface definitions, they also support refining interface definitions. 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.

```
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.

3.4 ClassDL

As stated in Sec. 3.2, VMADL requires the implementation language to provide mechanisms for data type refinement and explicit control over object layout. Since we used C as an implementation language, these features were not directly available.

Thus, we introduced ClassDL as a language orthogonal to VMADL, which provides us with the necessary flexibility to describe crosscutting refinements of classes in our OOP emulation. ClassDL is merely a simple layer on top of C that provides notation to define classes and traits. 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 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. It replaces C headers but relies on the implementation given in C source files (cf. Sec. 3.6).

ClassDL's sole purpose is to add the required capabilities of data type refinement and control over object layout to the used implementation language. Had CSOM been implemented in a different language with dedicated support for object-oriented modularisation and heterogeneous crosscutting, ClassDL would probably not have been necessary.

3.5 Evolution of VMADL

As already mentioned, the first version of VMADL was introduced [HAT+09] as a proof of concept. The version of VMADL presented here is has evolved beyond a mere proof of concept into a robust tool chain.

From a language point of view, three concepts were added. First, *combiners* were introduced to enable a clear separation between service module interfaces and service module interactions. This allows for a clearer modularisation and avoids polluting base modules with optional functionality. In the proof of concept, service module interactions were defined in service module definitions themselves, prohibiting definitions of interaction facets when introducing new service modules.

Second, VMADL as presented here introduces explicit module relationships using the `require` keyword. The reason for this is twofold. On the one hand, it documents relationships between service modules on an architectural level. On the other, it enables the tool chain to select required code fragments automatically.

The third feature introduced in VMADL presented here is the concept of *named sections*. While it is useful for documentation purposes, it also enables crosscutting refinement of service module interfaces, especially if the underlying implementation language lacks sufficiently powerful mechanisms.

3.6 The VMADL Tool Chain

The VMADL tool chain reuses standard tools wherever possible and introduces a custom compiler only to process the actual VMADL/ClassDL source code. We now give an overview of the work flow for the user, implementation details, and relevant limitations of the current tool chain as a whole.

Workflow An overview of the interaction of the involved tools, compilers, and artifacts is given in Fig. 2. As discussed in Sec. 4, *pure::variants* is used to build the feature model which is represented at the source code level by the different VMADL modules. Based on the feature model, the developer can select a *product configuration.* Pure::variants gives the set of selected service modules to the `make`-based build system. The build system in return invokes the different compilation steps to process the source artifacts and select the required implementation fragments to generate a binary for the desired configuration. It needs to be noted that pure::variants is optional. Users relying on their ability to select non-conflicting feature combinations can enter the `make` command line resulting from applying pure::variants themselves.

To ease development with the tool chain, it adheres to the standard conventions of the *GNU Compiler Collection* in terms of interface and error reporting. Thus, the VMADL compiler gives feedback to the user in the normal error format known from GCC. This enables IDEs like Eclipse and Xcode to parse warnings and errors and display them in line with the code, as is done for normal GCC error messages.

Furthermore, debugging is facilitated by utilising the standard `#line` preprocessor pragma, which enables compatible IDEs to debug directly on the original source file

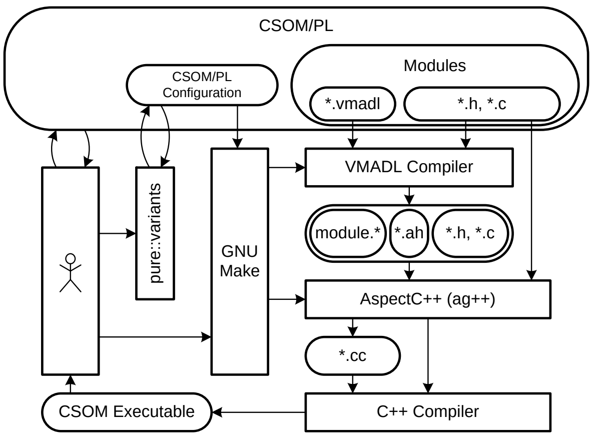


Figure 2 – VMADL tool chain overview

instead of the preprocessed intermediate version produced by the VMADL compiler.

Implementation The VMADL compiler is implemented in Java and uses ANTLR grammars to generate parsers for VMADL and ClassDL. Furthermore, it uses a simplified grammar for C to parse function definitions. The main purpose of the compiler is to preprocess the VMADL definitions and generate the actual implementation files for C and Aspect C++.

Pure::variants creates a simple build script containing a command line to invoke `make`. The VMADL compiler is parameterised with those. The main input for the compiler are thus a set of `*.vmadl` files containing the definition of service modules and combiners. These files are parsed and checked for consistency. Required VMADL modules are loaded on demand based on the initially given list of service modules. The simple C parser is used to parse the C implementation files and verify that all functions are available that have been defined in the ClassDL definitions.

When the consistency check fails, the error is reported back to the user immediately. In case it passes, the VMADL compiler will generated AspectC++ `.ah` files with all advice and pointcut definitions provided in the service modules. Furthermore, it will generate C header files that define the functions, types, and variables exported by a service module. The implementation of service modules is not provided inline with their definition, but comes as additional C files.

In addition to these VMADL specific implementation files, the ClassDL compiler (part of the VMADL compiler), generates a set of headers and C files defining the C representation of the defined classes and traits. The header files are split in two parts. The first header for each service module contains forward declarations of the necessary `struct`s, i. e., type information for the class, and the second header contains the definition of the actual `struct`s representing the classes/traits and their virtual method tables. Furthermore, a single common implementation file is generated, containing the virtual method table initialisers.

The compilers do not place any restrictions on file names or implementation file arrangement for the normal C code itself. The only requirement is that the compiler be able to identify the VMADL definitions based on names used for service modules.

This gives freedom to the VM implementation about how to arrange the different artifacts, e. g., how to implement a service module with actual C code. Furthermore, 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.

For convenience, the compiler verifies that the selected source tree contains an implementation for every function defined as a member of a class/trait. Otherwise, the developer would be notified about such an inconsistency only at link-time.

In the final step, the generated files are processed by the AspectC++ and C++ compilers to create the resulting CSOM binary. This final step works on the result of the previous preprocessing step; thus, all transformation/adaptations defined in VMADL—redefinition of named parts in service modules and modifications to classes—have already been applied at that point and are consistently visible to the compilers.

Practical Limitations The main limitation of the current tool chain is that it does not enforce consistency of feature model and implementation artifacts. Thus, it is possible that implementation and model get "out of sync" and valid or invalid configurations are not properly reflected by the model. We would like to note, though, that the tool chain is primarily a language processor, not a feature modelling tool.

Other limitations stem from the fact that C and AspectC++ have been used as implementation languages. As mentioned before, ClassDL could be replaced with another language (e. g., C++) since its sole purpose is to provide a conceptual model of classes and traits that can be the target of refinements by other service modules.

The current status of the AspectC++ implementation raises two relevant issues. The first is a limitation of AspectC++ regarding functions with variable argument lists. This is currently not fully supported by AspectC++; thus, our experiments had to avoid using this feature for functions that constitute interesting join-points.

Another AspectC++ characteristic is more critical, especially with regard to the performance-sensitive field of VM implementations. The issue is with advice that access and modify return values of advised functions. Here, the current state of C++ compiler optimisations fails to provide adequate performance for complex situations like interpreter loops. The workaround of replacing return values with reference parameters is not ideal, but allows avoiding performance impacts resulting from advice usage. However, this is a limitation of some of the elements of the current tool chain, but not of the overall approach of VMADL.

4 A Virtual Machine Product Line

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

4.1 The CSOM/PL Feature Model

With the proof of concept implementation of VMADL [HAT+09], it was possible to use the implementation of explicit memory management, mark/sweep and reference-counting GC, 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 feature combinations were provided. Some of the CSOM versions composed using the VMADL proof of concept

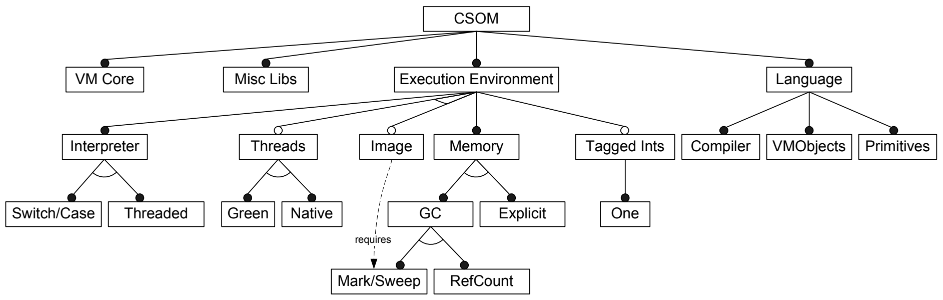


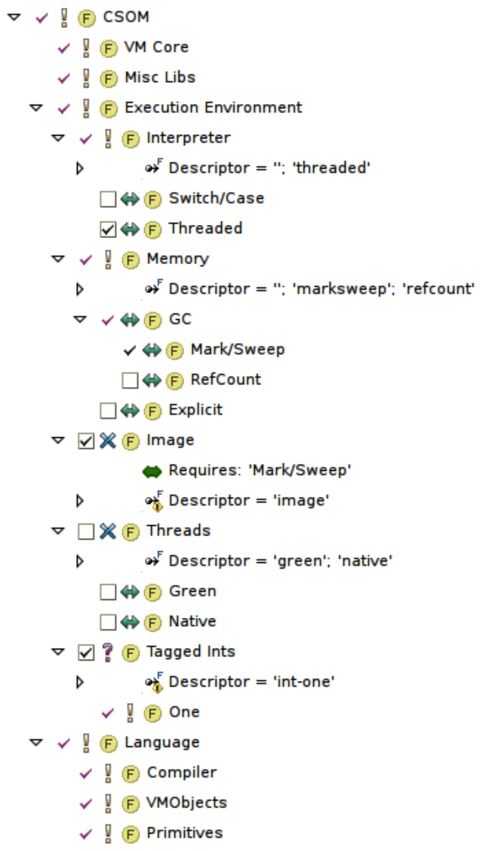
Figure 3 – The CSOM/PL feature model as realised by the current implementation.

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. 3 shows a feature diagram representing the current status of CSOM/PL. As usual with feature diagrams, boxes represent particular features, with the root representing the product line in question. Inner nodes denote features that can be realised in different ways; leaves indicate concrete shapes of certain features. The connections between features in the diagram point out dependencies. Mandatory features are represented by lines with a filled circle at the end; lines for optional features have an empty circle. An arc between lines indicates an alternative.

Each of the 14 concrete features in Fig. 3 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. 3.6), the VMADL compiler decides which interactions are actually required to instantiate a given product, and generates code only for those. More detailed descriptions of



```
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 4 – A concrete CSOM/PL configuration in pure::variants.

the various feature implementations are given below.

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, a property that is not guaranteed by explicit memory management and reference counting. 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 pure::variants, an industry-strength tool for variability modelling and management. It features a model editor, product configurator, validity checker, and a rich generator infrastructure. Fig. 4 shows a screenshot from the product configuration view, where the entire feature model tree of CSOM/PL has been expanded.

In the figure, yellow circles with an "F" denote features. An exclamation mark indicates a mandatory feature; a question mark, an optional feature; a green "X", conflicting features. Unboxed ticks denote automatic selection, tick boxes indicate possible configuration choices. Double green arrows next to feature circles mean that these features have dependencies, and hence, consequences for the overall configuration if selected. The various "Descriptor" nodes contain text fragments that are used to assemble the list of parameters passed to the `make` script generated from a configuration.

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 one-line build script that is used to drive CSOM/PL `Makefile` configuration (cf. Sec. 3.6). For the example given in Fig. 4, the resulting `make` invocation would look like this: `make threaded marksweep int-one image`.

4.2 Feature and Product Implementations

We will now give brief examples of how VMADL was used to implement the CSOM/PL features as service modules, and how those were combined to instantiate products. 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 only give brief examples. A more elaborate example is given in the appendix. It shows the definition of the *native multi-threading* service module (`NativeThreads`) and its combination with the `Interpreter` and `GCMarkSweep` modules. This combination was chosen because it significantly influences the involved service modules.

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, simply falling back to `malloc` and supporting no garbage collection, does not provide any additional functionality but relies on the interfaces offered by `VMCore` and `VMObjects`.

The *mark/sweep GC* implementation is almost transparent to the explicit memory management service module which acts as an underlying base module. Only 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 ClassDL `refine` statement. It inserts the mark field before the first field containing a member slot.

```
 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.

Furthermore, a combiner describes how the `GCMarkSweep` and `VMObjects` service modules interact, introducing a guard for object initialisation. This avoids dangling pointers resulting from 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* (`GCRefCount`) is, at first, quite similar, as the example in Lst. 6 illustrates. A `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.

```
service GCRefCount {
  refine VMObject {
    int gc_field { before fields[0] }
  }
}
combine GCRefCount, VMObjects {
  advice execution(VMObject::set_field(self, idx, val)) : around(self, idx, val) {
    pVMObject old_val = self->fields[idx];
    gc_inc_reference_counter(val)
    tjp->proceed();
    gc_dec_reference_counter(old_val);
  }
  advice execution(VMObject::set_class(self, idx, val)) : around(self, idx, val) {
    pVMObject old_val = self->clazz;
    gc_inc_reference_counter(val)
    tjp->proceed();
    gc_dec_reference_counter(old_val);
  }
  ...
}
combine GCRefCount, Interpreter {
  advice Interpreter::unwind_stack() : before() {
    pVMFrame frame = Interpreter_get_frame();
    gc_inc_reference_counter((pVMObject)frame);
  }
  ...
}
```

Listing 6 – Combine Reference Counting with VMObjects.

Multi-Threading From the modularisation perspective, *green threading* is quite undemanding. The service module `GreenThreads` 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. 7.

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

Listing 7 – Pre-emption for green threads.

Some additional combiners are necessary to support the combined usage of multi-threading with the different GCs. For configurations using `GCMarkSweep`, 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 `NativeThreads`, the changes are more fundamental than for green threads (cf. Sec. 2). The major task is to achieve thread-local execution of interpreters 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 `NativeThreads` and `GCRefCount` service modules. With `GCMarkSweep`, this is more challenging. The scheme that was implemented in the feature combination found in CSOM/PL is a stop-the-world solution [JL96]. This is implemented entirely inside a combiner (cf. App. A) 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.

Execution Engine Interaction with the two possible interpreters is realised using the common `Interpreter` service module interface. The *threaded interpreter* (`InterpreterThreaded`) 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 [Bel73] 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.

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. 3.3), 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 [GR83].

Image Persistence The Smalltalk *virtual images* implementation in the `Image` service module 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.

4.3 Feature Generalisation

The intertwined nature of VM implementations usually leads to ad-hoc defined interfaces. As can be seen from the previous section, this tendency exists also for service modules. However, once a number of features requires a certain set of interfaces and events, it becomes beneficial to generalise these features.

In VMs, a common problem is the interaction of all modules with GC logic. The characteristics for different GC approaches differ especially when it comes to performance characteristics. Thus, a VM product line should be able to provide different GC algorithms for specific purposes.

The major concern for many GC types are read and write barriers. Thus, the VM needs to make it explicit for the GC when values are read from the heap or stored into it. Also important are references stored in registers, on the stack, or in other globals that are not immediately known to the GC.

What we have seen above for the reference-counting GC are typical write barriers: on every write, the reference count of the old and new values have to be adapted. Furthermore, there are a couple of write barriers for storing into temporary locations on the stack necessary to be able to track the correct reference counts and avoid premature reclamation. Read barriers are very similar, but are not discussed here. One example of a GC algorithm using a read barrier is the pauseless GC [CTW05]. Here, the value of each reference that is read has to be checked and possibly updated.

With VMADL, concepts like read/write barriers can be reified and generalised on the architectural level. Instead of leaving the responsibility to the reference-counting GC to identify the points where reads/stores constitute a relevant event for a barrier, the service modules themselves can make these explicit. For example, see the pointcut in Lst. 8 which aggregates the relevant writes of `VMObjects`.

```
pointcut Interpreter::object_store() =
      execution(VMObjects::set_field(...))
   || execution(VMObject::set_class(...))
   || ...
```

Listing 8 – Generalised Write Barrier for the VMObjects.

This leads to the desired modularisation, since now the GC modules need only architectural-level information about the event, and a service module can expose this through appropriate pointcuts.

Name	Selected Optional Features		
Base	Switch/Case Interpreter	Explicit Memory	
RefCount	Switch/Case Interpreter	RefCount GC	
Mark/Sweep	Switch/Case Interpreter	Mark/Sweep GC	
Green	Switch/Case Interpreter	Green Threads	
Native	Switch/Case Interpreter	Native Threads	
Image	Switch/Case Interpreter	Mark/Sweep GC	Image
Tagged Ints	Switch/Case Interpreter	Mark/Sweep GC	One Tagged Ints
Threaded	Threaded Interpreter	Mark/Sweep GC	

Table 1 – CSOM/PL configurations used for evaluation (cf. Fig. 3)

5 Evaluation

Having turned a set of hand-crafted extensions to a base system into a set of cleanly encapsulated 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 metrics to it, also considering modularity improvements.

The hand-crafted CSOM variants were the results of student course-work, as noted in Sec. 2. These variants are used as the baseline for comparison. The derived VMADL-based product line is kept as close to the hand-crafted versions as possible. To avoid invalidating the results, the code bases have been synchronised to remove differences in functionality and execution semantics. The differences in the source code were minimised. The VMADL-based implementation uses configurations corresponding to the hand-crafted versions. The following section uses the selected optional service modules to indicate the exact configuration. For example, Green is the configuration including the (mandatory) base system and the green-threads feature, while Base is the base system without any optional service modules, i. e., the simplest possible VM configuration with a minimal feature set.

Below, we elaborate on the performance and code complexity assessments, and conclude the section with a discussion of our approach.

5.1 Virtual Machine Performance

Performance measurements were run on a 8-core workstation with two Intel Xeon E5520 processors (2.27 GHz clock rate, 16 hyperthreads overall, 8 MB cache) and 8 GB RAM. The operating system was Ubuntu Linux 10.10 with a 64-bit kernel (version 2.6.38-11) and 32-bit user land. The used compilers were GNU C++ 4.5.1, and `ac++` 1.0/`ag++` 0.8 for AspectC++. The C++ compiler flags used were `-m32 -O3 -flto`, i. e., to compile for 32 bit, with the highest optimisation level, and link-time optimisation turned on. The used benchmarks originate from the original SOM implementation and are a set of micro-benchmarks and kernels. They are listed with a brief description in Tab. 2.

For the performance evaluation, we followed the suggestions of Georges et al. [GBE07]. The benchmarks were executed using ReBench[8] on an idle machine. Each benchmark was executed 100 times to ensure statistical confidence in the results. Each

[8] `https://github.com/smarr/ReBench`

Micro-benchmarks		Kernels	
IntegerLoop	pure loop over integers	Bounce	physics simulation
Fibonacci	standard tree-recursive	Queens	eight queens puzzle
Dispatch	method dispatch	Storage	n-ary tree creation
Loop	nested loops	Sieve	simple prim sieve
Sum	object modification in loops	Towers	Towers of Hanoi
List	linked list creation/traversal	BubbleSort	standard sort
Recurse	self-recursive calls	QuickSort	standard sort
		TreeSort	binary search tree

Table 2 – Benchmarks used for performance evaluation

benchmark was executed in a dedicated CSOM instance. The benchmark runtime was measured without VM startup time. To reduce result variation, all benchmarks were executed on exactly the same core, and we regard only single-threaded 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 allows us to assess the performance impact of using VMADL on a single CSOM/PL product. We did not run performance measurements for feature combinations for which no hand-crafted VM exists as they cannot provide any insight on overheads induced by VMADL. Each measurement pair for a given benchmark and feature set was used to calculate the performance ratio of the VMADL-based version divided by the hand-crafted version. Thus, the ideal result would be a factor of 1, indicating identical performance properties. A runtime factor higher than 1 indicates an overhead induced by our approach.

The performance comparison results are displayed in Fig. 5. For each CSOM configuration, the relative performance of the VMADL version to the hand-crafted build is shown as an accumulation over all benchmarks. The individual benchmark results are left out for brevity; they neither show statistically relevant outliers nor diverging behaviour. The used graph is a standard box plot and indicates the distribution of the measurements. The desired result of 1 is indicated by a dashed line.

The first box, *average*, represents the average performance of all particular benchmark runs for all configurations. It shows that adopting VMADL does not entail a statistically significant overall impact on performance. On average, the benchmarks take, on the VMADL versions of CSOM, only about 99.18 % of the runtime they take on the hand-crafted versions. However, with a standard deviation of 3.3 % the difference is not significant.

Half of the VMADL-generated products exhibit small improvements. Only the Tagged Ints, Native, and Threaded configurations exhibit minor performance degradations of less than 2 %. Since these performance changes are within the limit of the performance changes observed for the other experiments as well, we attribute them to the effects of compiler heuristics. The woven code produced by AspectC++ introduces additional complexity for optimisations; so does the code organisation, in terms of C implementation files changed between the hand-crafted and the VMADL version. Thus, we assume those performance differences to be caused by differently applied compiler optimisations. This assumption was tested by running the benchmarks with other optimisation flags and compiler versions. The result is that the different configurations have different optimal compiler settings, fluctuating within the 2 % margin around the factor 1.

Note that there are currently restrictions with regard to the reliability of compiler optimisations and certain AspectC++ constructs. As mentioned in Sec. 3.6, the used

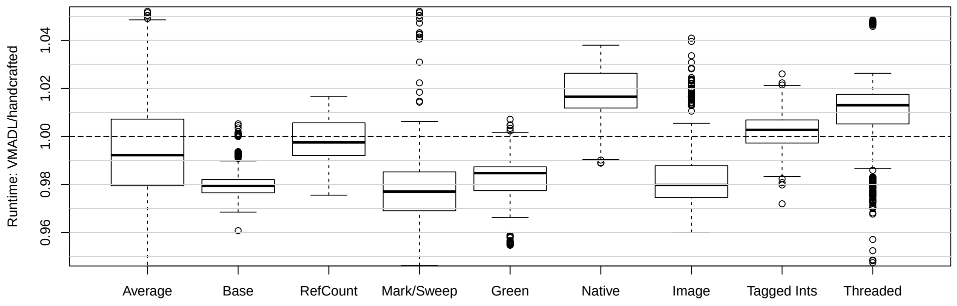


Figure 5 – Accumulated relative performance over all benchmarks of VMADL versions to hand-crafted feature combinations.

C++ compiler does currently not reliably optimise advised code modifying return values. To achieve a performance neutral implementation, performance critical parts on the interpreter loop had to be refactored to use reference parameters.

5.2 Source Code Complexity

To assess source code complexity, we applied several metrics. Notice that only the *actual* CSOM/PL source code was regarded. For some of the extensions, e. g., multi-threading, the Smalltalk libraries had to be extended as well, providing 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.

Lines of Code Results The *physical source lines of code* (PSLOC) [Par92] 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, which must be accounted to the use of combiners as they add a declarative yet somewhat more verbose syntax. 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 the ClassDL effect.

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 Base was compared to RefCount, Mark/Sweep, Green and Native. Moreover, CSOM's Mark/Sweep configuration was compared to Image, Tagged Ints, and Threaded. 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 Tab. 3. The table details the impact on the level of lines of code, files, and language. The impact on lines of code is

	Lines of Code							
	Added		Removed		Changed (new)		Changed (old)	
	HC	VMADL	HC	VMADL	HC	VMADL	HC	VMADL
RefCount	455	564	3	0	31	0	35	0
Mark/Sweep	541	788	4	0	68	0	32	0
Green	352	390	0	0	13	0	5	0
Native	1033	702	0	0	7	0	3	0
Image	3375	2429	30	0	94	0	133	0
Tagged Ints	222	296	0	0	33	0	23	0
Threaded	1447	1001	6	0	96	0	71	0
Total sum	7425	6170	43	0	342	0	302	0

	Files				Functions & Structs	
	Added		Modified		Modified	
	HC	VMADL	HC	VMADL	HC	VMADL
RefCount	2	3	14	0	37	0
Mark/Sweep	0	3	22	0	35	0
Green	4	3	6	0	9	0
Native	14	5	10	0	20	0
Image	6	11	38	0	85	0
Tagged Ints	1	3	10	0	29	0
Threaded	9	7	10	0	31	0
Total sum	36	35	110	0	246	0

Table 3 – Modularity metrics results. Comparing hand-crafted (HC) vs. VMADL

represented as newly added lines, removed lines, and changed lines. Changed lines are split into *new* and *old*, representing the amount of modification as reported by GNU `diff`. That is, the lines in a chunk of changes in the base version (old) compared to the changes in the final version (new). The numbers for changed lines do not include added and removed lines.

The changes on file level correspond better to modules than the fine-grained lines of code metrics and are given here with the number of new and changed files. For the intermediate level, we report the number of changed functions or structure definitions to give a better indication of the number of semantic changes.

The table 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 hand-crafting 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 Conclusions

Modularity An approach can be called "modular" [Par72] 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 [Mey97], add more detail to these criteria but basically imply the same.

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. In this regard, VMADL combiners, keeping the entire specification of service interactions *in one place* and referring to interfaces, certainly provide stronger means than simple constructs like macros or certain design patterns [Lad10, pp. 17ff][9] usually mentioned when discussing MDSOC.

One might criticise that combiners sometimes need to refine particular pieces of service modules to enable the interaction they specify. While this might come across as a violation of modularity, we would like to note that such intrusions are well encapsulated and strictly confined to the interaction at hand. They apply only in the setting described by the respective combiner and have no effect in other interactions not occurring in the same product configuration.

VMADL's name suggests a strong tie to the VM implementation domain. In fact, the name stems from the primary purpose the language was developed for. Consequently, VMADL can, in spite of its name, be used in other implementation domains as well. As shown, it can prove useful in C/C++ projects that have the potential to grow into a software product line.

JIT Compiler Integration The simplicity of CSOM, most notably the lack of a JIT compiler, gives rise to the question of how such a feature could be integrated using the VMADL mechanisms. Using VMADL constructs, it is not possible to directly reason about exposed and other join points in machine code generated at run-time. However, it is possible to use VMADL to express that JIT compiler code *generating code corresponding to such join points* be instrumented accordingly.

Consider the example of combining a JIT compiler and reference-counting garbage collector. Provided the JIT compiler exposes internal situations such as "generating pointer assignment" and "generating code for parameter passing", the reference counting GC module can interact by requesting the generation of reference counter adjustments along with the actual assignments.

SPL Tooling 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.

Conclusions 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.

[9] Also, `www.ibm.com/developerworks/java/library/j-aopwork15/`.

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 [RP06] 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 away the complexity of interactions between VM run-time and memory management, easing development significantly. While this is appreciable, it comes at a certain cost. The interpreter and other parts of the VM are represented by code actually describing a VM *implementation*, whereas memory management is represented by code that describes VM implementation *transformations*. There are two different kinds of abstraction at play in this setting, with memory management being extraneous to the actual VM. The goal we pursue with VMADL, conversely, is 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 [AAB$^+$99, A$^+$00] and Maxine[10] 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[11] to complete Java source file stubs for various features prior to compile-time. Memory management is performed by MMTk [BCM04], 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. To summarise, Jikes realises variability by template-based 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.

[10] `labs.oracle.com/projects/maxine`
[11] `jikesrvm.org/Building+the+RVM`

Maxine tries to improve modularity with the language features[12] 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 [Pre97, ALRS05]. Refining a previously defined class in the context of a specific feature effectively supports heterogeneous crosscuts. VMADL thus combines aspect- and feature-oriented approaches [ALS06] in a more architecture-aware manner, e. g., by making module relationships explicit in declarations using requirement statements.

VMKit [GTL$^+$10] is called a "substrate" for implementing VMs. It provides a common foundation that implementations of different instruction sets and programming 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, it restricts choices offered at the substrate level. For instance, LLVM [LA04] is used as the JIT compiler infrastructure, and MMTk [BCM04] 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 [ACN02]. Like ArchJava, VMADL makes system architecture explicit in the source code itself. Other languages like WRIGHT [AG97] or Rapide [LV95] 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 [GLvB$^+$03], 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 component instances at run-time. Components provide communication ports, and connections are explicit. This 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. [AMC$^+$07] 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++ [ALRS05] is the implementation of SPLs [RSSA08]. 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

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

features 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. [FCS$^+$08] 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 currently 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* [LSGF08]. 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 optionally representing the feature model 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 several CSOM Smalltalk VM features in *combinable isolated modules*—features that were previously realised as hand-crafted extensions. This also facilitated devising a product line. The evaluation shows that performance is not harmed, and that source code modularity is significantly improved.

Regarding SPL tool support, VMADL represents a valuable tool that can be used to introduce modular SPL development in languages lacking inherent modularity support. Moreover, VMADL provides 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. Our ongoing work is concerned with moving towards the goal of dropping all constraints shown in Fig. 3 that 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++ has been done and is considered for future research in disentangling VM architecture.

We also hope to transfer our results to other, more complex, VM implementations to gain more insights into the modularisation of full-scale VM implementations. Part of the ongoing work in the Maxine project at Oracle Labs is investigating this.

References

[A$^+$00] B. Alpern et al. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000. doi:10.1147/sj.391.0211.

[AAB$^+$99] 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.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: Connecting software architecture to implementation. In *Proc. ICSE'02*, pages 187–197. ACM, 2002. doi:10.1145/581339.581365.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997. Available from: http://portal.acm.org/citation.cfm?doid=258077.258078, doi:10.1145/258077.258078.

[Ald05] J. Aldrich. Open modules: Modular reasoning about advice. In *Proc. ECOOP'05*, volume 3586 of *LNCS*, pages 144–168. Springer, 2005. doi:10.1007/11531142_7.

[ALRS05] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proc. GPCE*, 2005. doi:10.1007/11561347_10.

[ALS06] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual mixin layers: Aspects and features in concert. In *Proc. ICSE'06*. ACM, May 2006. doi:10.1145/1134285.1134304.

[AMC$^+$07] Vander Alves, Pedro Matos, Leonardo Cole, Alexandre Vasconcelos, Paulo Borba, and Geber 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. doi:10.1007/978-3-540-77042-8_5.

[AS07] B. Adams and K. De 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. doi:10.1145/1233843.1233844.

[BCF$^+$99] 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. doi:10.1145/304065.304113.

[BCM04] 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. doi:10.1109/ICSE.2004.1317436.

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973. doi:10.1145/362248.362270.

[CN02] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[CTW05] Cliff Click, Gil Tene, and Michael Wolf. The pauseless gc algorithm. In *VEE '05: Proceedings of the 1st ACM/USENIX international confer-*

ence on Virtual execution environments, pages 46–56, New York, NY, USA, 2005. ACM. doi:10.1145/1064979.1064988.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. *ACM SIGPLAN Notices*, 24(10):49–70, 1989. doi:10.1145/74878.74884.

[FCS$^+$08] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with aspects: An empirical study on design stability. In *Proc. ICSE'08*. ACM, 2008. Available from: http://portal.acm.org/citation.cfm?id=1368124&jmp=abstract&coll=&dl=ACM, doi:10.1145/1368088.1368124.

[GBE07] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, 2007. doi:10.1145/1297105.1297033.

[GLvB$^+$03] 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. Available from: http://portal.acm.org/citation.cfm?id=781133, doi:10.1145/781131.781133.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[GSS$^+$06] 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. doi:10.1109/MS.2006.24.

[GTL$^+$10] 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. doi:10.1145/1837854.1736006.

[HAT$^+$09] 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. doi:10.1049/iet-sen.2007.0121.

[HHP$^+$10] 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. doi:10.1145/1822090.1822098.

[JL96] R. Jones and R. Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[KHH$^+$01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. Lindskov Knudsen, editor, *Proc. ECOOP'01*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001. doi:10.1007/3-540-45337-7_18.

[LA04] 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. doi:10.1109/CGO.2004.1281665.

[Lad10] R. Laddad. *AspectJ in Action*. Manning, 2nd edition, 2010.

[LB96] B. Lewis and D. J. Berg. *Threads Primer. A Guide to Multithreaded Programming*. Prentice Hall, 1996.

[LSGF08] Neil Loughran, Pablo Sánchez, Alessandro Garcia, and Lidia Fuentes. *Language Support for Managing Variability in Architectural Models*, volume 4954 of *LNCS*, pages 36–51. Springer, 2008. doi:10.1007/978-3-540-78789-1_3.

[LV95] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995. doi:10.1109/32.464548.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE TSE*, 26(1):70–93, 2000. doi:10.1109/32.825767.

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

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972. doi:10.1145/361598.361623.

[Par92] Robert 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.

[Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. *LNCS*, 1241:419–434, 1997. doi:10.1007/BFb0053389.

[RP06] Armin Rigo and Samuele Pedroni. Pypy's approach to virtual machine construction. In *Proc. OOPSLA'06*, pages 944–953. ACM, 2006. doi:10.1145/1176617.1176753.

[RSSA08] 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. doi:10.1145/1449913.1449917.

[SN05] 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;
    #include <pthread.h>
    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 { ... }
}

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;
    }
}

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

required interfaces from other service modules

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

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

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

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

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

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

About the authors

Michael Haupt is a Principal Member of Technical Staff in the Maxine team at Oracle Labs. Before joining Oracle, he was a post-doctoral researcher and lecturer in the Software Architecture Group at the Hasso-Plattner-Institut (HPI) in Potsdam. Michael holds a doctoral degree from Technische Universität Darmstadt, Germany. He can be contacted by e-mail via `michael.haupt@oracle.com`, his home page is at `http://labs.oracle.com/people/haupt`.

Stefan Marr is a PhD student at the Software Languages Lab of the Vrije Universiteit Brussel. He graduated with a master degree at the Hasso-Plattner-Institut (HPI) before going to Brussels to work on virtual machines for the manycore era. His email address is `stefan.marr@vub.ac.be` and his home page can be found at `http://soft.vub.ac.be/~smarr/`

Robert Hirschfeld is a Professor of Computer Science at the Hasso-Plattner-Institut (HPI) at the University of Potsdam. He received a Ph.D. in Computer Science form the Technical University of Ilmenau, Germany. He can be reached at `hirschfeld@hpi.uni-potsdam.de`. See also `http://www.hpi.uni-potsdam.de/swa/`.

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.

Stefan Marr is supported by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.