A Generic Framework for Component-Based Development of Virtual Machines

Mirko Raner
(raner@acm.org)

30th October 2000

Abstract

Virtual machines play an important role in information technology today and they will probably become even more important in the future. The conception and implementation of new or customized virtual machines is one of the big challenges of the upcoming years.
Though it is easily possible to identify different functional units within a virtual machine model (execution engine, loader/linker, security system, thread manager, native device support, etc) most state-of-the-art VMs are implemented in a monolithic way. That is, often individual components cannot be replaced without major changes to the source code of other components.
The thesis will present a general virtual machine model with separated functional components and will propose component interfaces with exact descriptions. These interfaces will be used as the foundation of a generic framework which is intended for the assembly of virtual machines from individual components.
Besides a thorough discussion of the framework it will also try to provide a "proof of concept" by showing how a simple Java VM could be built by means of component-based development.

Trademarks

Acknowledgements

I am thankful to a large number of people who supported me during the writing of this thesis.

Special thanks – in no particular order – go to: Graham Nice of Tao Group Ltd (Reading, UK) for sending me the specification of the Virtual Processor (unfortunately covered by an NDA), Erwin Hösch of MATHEMA Software GmbH (Erlangen, Germany) for moral support and prompt help concerning the Interface Definition Language, Dr Charles Forsyth of Vita Nuova Holdings Ltd (York, UK) for making the garbage collection code of the Dis Virtual Machine available to me, Gordon Wilkie of Cabot Communications Ltd (Bristol, UK) and Laurie Boshell (Australia) for helping me dig out information about the ancient P-Machine and, last but not least, Christoph Zapf of the University of California at San Diego for showing me my way through the Geisel library at UCSD.
Many thanks go also to my parents Franc and Roswitha Raner, my good friends Ansgar Hoffmann, Fränk Rothgang, Ulli and Gerald Weller, Anja Frisch and Jan Hildebrandt who kept inquiring for the progress of my work – you were the pain in the neck which kept me going ;-)

Contents

List of Tables

List of Figures

1 Introduction

Virtual machines (VMs) have gained a lot of importance since the basic concept was first introduced in early systems like IBM's "VM" (see [MD97, Cha89]) or the UCSD P-System [Bow78]. Today, virtual machines are an omnipresent phenomenon and especially for mobile code and heterogeneous distributed systems they play an important role. The most popular and best-documented VM is undoubtedly the Java Virtual Machine (JVM) [LY99, MD97, Ven99, Dal97] which provides the foundation for the Java programming language and numerous technologies which are based on Java.[1] However, besides Java, there are several other VM systems, like, eg, the "Dis" VM in the Inferno OS (see [WP97, VNH00] and [DPP+97]) or the VP (Virtual Processor) in Tao Systems' Elate OS (see [Tao00]; formerly named "TAOS") which are very successful in their particular market segments.

1.1 Problem Statement

A concrete implementation of a virtual machine model is a very complex piece of software, and the design and implementation of a VM is a very labour-intensive undertaking (see, eg, [WAU99, §1] "Building virtual machines is hard"). The majority of current VM implementations have a monolithic architecture [SGGB99, Har99]. The particular compontens of a virtual machine, like the execution engine, the memory management, the thread scheduling and the device support closely interplay with each other and it is not easy to cleanly separate these components into independent modules [Ran99b]. This monolithic structure makes it very hard to exchange single components of a VM. For example, it can be very interesting to replace an interpreter-based execution engine by a recompiler-based one or to change the implementation of a garbage-collecting memory manager. Such experiments can yield very interesting results and can, eg, help to examine the influence of particular components on the overall performance or suitability of a virtual machine.
In addition to that, virtual machines often need to be customized for new fields of application (eg, for use in embedded devices). Today, even application-specific VMs are an important topic [Har99, FPR98]. It would be an enormous development advantage if a new VM could simply be assembled from a small number of appropriate components.
Most current virtual machine implementations do not offer this flexibility because they are implemented in a monolithic way. A change in one of the components often implies changes in other components and large portions of the VM source code have to be modified.

Though a VM undoubtedly can be split into separate components (see, eg, [MD97, Ran99a] for the parts of the Java VM) these components are often not strictly separated in the actual implementations. For instance, it still poses major problems to replace the garbage collection mechanism or the thread model of existing virtual machines. Even in public domain / cleanroom implementations like Kaffe (http://www.kaffe.org) or Blackdown Java

[1]This is also the reason why the Java VM will be frequently cited as an example throughout this thesis

(http://www.blackdown.org) the borderlines between individual components are not defined accurately enough, so that existing modules can be easily exchanged.

The reason for the monolithic implementation approach can probably be found in the manifold mutual dependencies between the particular components of a virtual machine. For example, figure 1 shows the components of the Java Virtual Machine (as listed in [MD97]) and some possible interactions between them. All runtime components interact with the execution engine, which implements the abstract processor. This does not only mean that their implementation code might actually be running on the VM execution engine, but also includes other interactions such as JIT compiler support for garbage collection (eg, [ADM98, SLC99]) and CPU time allocation for background compilation (eg, [Har98, KGL$^+$00]). Besides that, there are various interactions between the individual parts of the runtime system. The memory manager might need to access low-level memory allocation functions via the native interface and it has to co-operate with the class loader and the thread scheduler in order to allocate and release memory for class definitions and to enable multi-threaded garbage collection. The class loader itself has to interact with the security manager[2] in order to perform class verification and to determine the access rights of a loaded class. The security manager reports access violations by throwing security exceptions and therefore needs the support of the exception manager, and so on.
The VM model shown in figure 1 is not a general VM model and is only applicable to the Java VM, however it is a good demonstration of the manifold interactions between individual VM components (graphical presentation adapted from [Ran99b]).

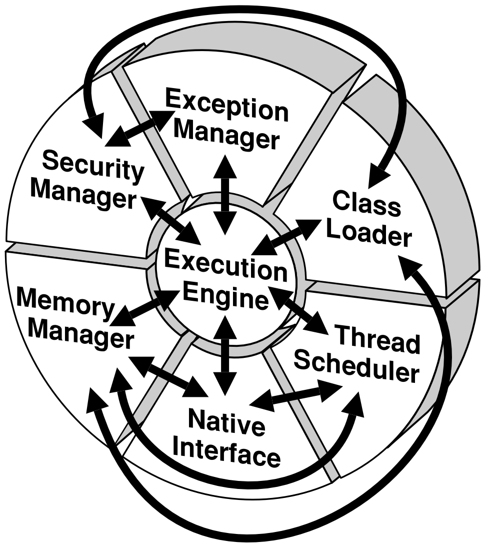

Figure 1: The components of the Java VM and some of the interactions between them.

[2]class verification in the JVM is handled by the bytecode verifier, which is implemented as a component of the class loader; however, from a strict point of view, this functionality actually belongs to the security manager

1.2 Objectives

The objective of this thesis is to specify a general virtual machine model with accurately defined interfaces for its particular components. The result of this work will be a complete framework for the flexible and component-oriented implementation of virtual machines.
The first goal will be the identification of all self-contained parts of a VM. Once this is done, a framework can be specified which can serve as a template for the implementation of future virtual machines (see figure 2 as an example framework for the Java VM). The individual interfaces can be implemented independently of each other. As long as an implementation conforms to the predefined interfaces of the framework it should be no problem to exchange it against another conforming implementation.

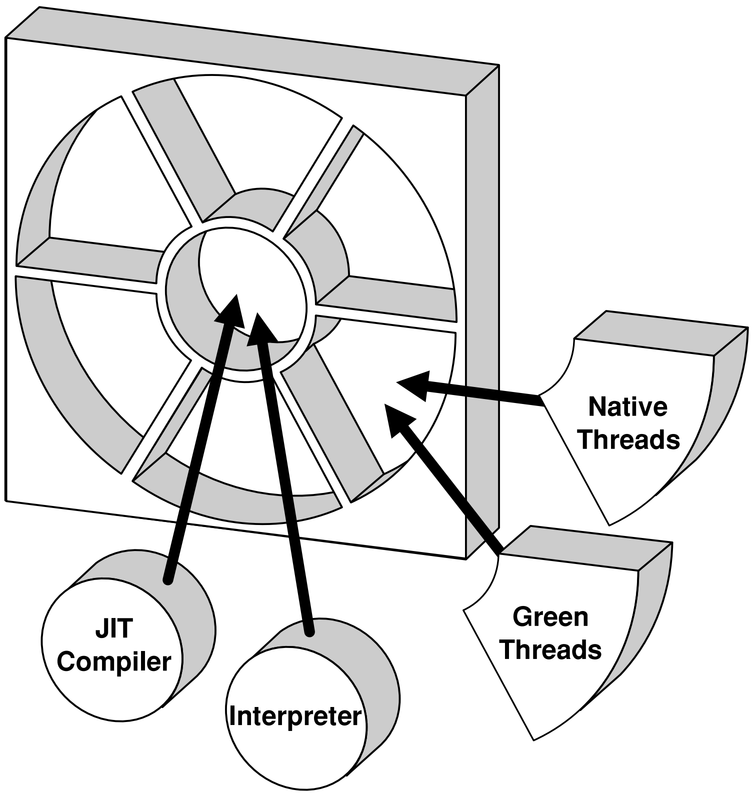

Figure 2: A possible framework for the Java VM and some possible component implementations.

The above described flexibility should be available for all VM components and not only for a small selection. Again the Java VM can be cited as an example: it is possible to replace the implementation of the class loader [LB98] and the security manager, but ,for instance, the garbage collection strategy is hard-coded into the VM. Blackdown's Java VM for Linux and Sun's VM for Solaris support the alternative use of green threads and native threads, but this is not yet a common feature of other VM implementations.

Besides the exchangeability of VM components such a framework makes another important contribution: even if there are no existing reusable components and the VM has to be built from scratch, there are at least cleanly defined interfaces which allow an easy definition of self-contained work packages that can be solved by separate implementation teams. So, another important final goal is a specification which can serve as an implementation guideline and facilitates the team-based implementation of virtual machines.

Above all, the thesis tries to draw a generic picture of a VM which is not restricted to common VM systems (such as the JVM or the Dis VM) but applicable to all types of VM.

1.3 Limitations

The analysis of existing virtual machines concentrated on imperative object-oriented or at least object-based[3] virtual machines with fixed object types (ie, every object has a type associated with it, and this type allows to trace all other objects it references).
Programming languages and virtual machines are conceivable, which do not fulfill this requirement[4] and might therefore not be implementable within the proposed framework.

1.4 Comparison with other Approaches

There were already some research efforts which were heading into a similar direction but only concentrated on particular aspects of the problem. For instance, the Portable Common Runtime (PCR) [WDH89] already defined interfaces for threads, I/O support, memory management and incremental loading and linking. However, the actual objective of the project was the improvement of inter-language interoperability and the approach did not consider reflection and security enforcement.
Other papers concentrated on the partitioning of VM services for distributed virtual machines (eg, [SGGB99]) and did not have a general framework in mind.

A lot of research has also been conducted in the area of meta-models for virtual machines. This includes the Sceptre XVM (Extensible Virtual Machine; see [Har99]) and the VVM (Virtual Virtual Machine; see [FPR97, FPR98]).
A virtual virtual machine or meta-VM is a generic programming environment into which VM definitions can be plugged in. That is, within a single VVM it is possible to execute Java and Smalltalk bytecode at the same time, provided that appropriate definitions for these two virtual machines are loaded into the VVM.
However, these approaches focus on meta-architectures for VMs and not on component frameworks. The presented meta-VMs require a relatively complex hosting environment for the definition of new VMs (eg, by so-called "VMlets" in the VVM, or by a kind of VM description language in the XVM). A virtual virtual machine / VVM provides the abstraction from the underlying operating system and hardware. The VMlet (or VM

[3]The Dis VM and the P-Machine are the only exceptions.

[4]Such candidates are, for instance, LISP- or Scheme-like languages where an expression like `(+ 1 2)` can either be treated as a simple list with the symbols `+`, `1` and `2` or it can be evaluated as a function expression.

definition) specifies a language-specific or application-specific front-end, like for instance, a Smalltalk environment or some kind of processor emulator.

The approach presented in this thesis and the approaches for meta-architectures have very much in common. Some of them are also component-based (eg, [Har99]) and have of course the same need for accurately defined interfaces.
This thesis takes a more conventional and simple approach than most of the meta-architectures. Depending on the implementation language, the individual VM components can be simply linked to a complete application by a standard linker. Meta-architectures with their VMlet interfaces or VM description languages may in some points offer more flexibilty, but the efforts required in order to get to that point must not be underestimated. A meta-VM requires a complete new environment for development and deployment and is naturally even more labour-intensive than a regular "base level" VM.

Due to the large intersections between both approaches it is very probable that some of the presented results are also interesting for meta-architecture approachs. In the VVM and XVM the VMlets/VM definitions are merely responsible for defining the basic operations and primitives of the bytecode execution engine. Therefore, the interfaces between particular runtime components are not as important as the general interface between the execution engine and the remaining runtime services.

1.5 Outline of the Thesis

Section 2 discusses the properties of virtual machines which are relevant for the definition of a framework and gives a brief overview of the particular virtual machines that have been take into consideration. This section uses a prototypic view of VM components.
Section 3 evaluates which basic services the functional units of a virtual machine provide and how they can be separated into components.
In section 4 a final framework with concrete interface descriptions for the components is proposed. The functionality of the particular components is examined in greater details.
Section 5 shows how a Java Virtual Machine could be implemented inside the existing framework. Final conclusions and an outlook to possible future work are presented in section 6.

2 Examination Criteria and Considered Virtual Machines

In order to construct a generic VM model it is inevitable to have a look at a number of existing virtual machines and extract their common characteristics. Virtual machines have a quite long history (see section 1) and therefore one might expect to find plenty of material on all kinds of VM systems. Unfortunately, this is not really the case. Many VMs date back into the 60s, 70s and early 80s. All these systems once were very well-documented, but most of these documents are not anymore accessible today. Almost all of the books about early VMs are long out of print (such as [GR83]) and are very hard to find today. Among these almost obliterated virtual machines are for example the famous UCSD P-Machine [Bow78] and the EUMEL-∅ Machine [Eng88] (which is closely associated with the EUMEL OS and the Elan programming language, which were both developed by the GMD at the end of the 70s).
But it is also hard to obtain information about certain modern virtual machines. Many companies which use VM technology treat their VM model as a company secret. Therefore, the VM specifications are either subject to non-disclosure agreements (eg, Tao Systems' VP) or are completely unavailable to the public, such as Acucorp's ACUCOBOL VM (http://www.acucorp.com) or Cabot Software's MPC (http://www.cabot.co.uk).

For further research the following five virtual machines were selected:

- The Java Virtual Machine – JVM (see section 2.4.1)
- The Dis Virtual Machine (see section 2.4.2)
- The Virtual Processor (VP) (see section 2.4.3)
- The Smalltalk Virtual Machine (see section 2.4.4)
- The P-Machine (see section 2.4.5)

None of these VMs are similar in their structure and their features, so it is a very wide range of different VMs on which all further examinations are based.

There is a number of criteria according to which these virtual machines can be examined. The most important aspects are:

- Computation model (see section 2.1)
- Memory model (see section 2.2)

Besides that, there are also other characteristics that should be taken into account (for instance, typed versus untyped instructions, direct support for object-oriented concepts and multithreading). These other aspects are discussed in section 2.3.

2.1 Computation Models

Basically there are three different models of computation which can be used by a VM: stack-based computation, register-based computation and memory-based computation. The following paragraphs will briefly explain how these computation models work. Each model will be illustrated by a short source code sequence which performs the calculation `v1 = v2 - v3 + 4` (with `v1`, `v2`, `v3` being 32-bit integer variables).

2.1.1 Stack Machines (SM)

A stack machine performs all operations on an operand stack. All operands must be pushed onto the stack before the operation can be executed. The operation removes the operands from the stack and pushes the result back onto the operand stack. The result can be popped from the stack or can be left there for subsequent operations.
This computation model has the advantage that the code is very compact and many instructions can be encoded in a single byte because the operand information is not embedded in the instruction itself. Usually stack machines have additional "local variable" registers which can be only used for load and store operations. Besides that, there usually a very limited number of internal registers for storing the program counter and pointers to the current stack frame and the top of the operand stack (see, eg, [Dal97, §2.1]).
Stack machines do not make any assumptions about the number of available registers of the underlying native microprocessor. They can be implemented equally well – or equally badly – on processors with a small or a large number of registers. Most real-world microprocessors have nothing in common with stack-based processor models. Only very few microprocessors, such as the PSC 1000 by Patriot Scientific (see also [Ran99a]), implement a stack computation model in hardware.

The Java Virtual Machine uses a stack-based computation model. The following listing shows documented JVM code for the example calculation (see section 2.1):

```
iload_2   ; Push local variable 2 onto the stack
iload_3   ; Push local variable 3 onto the stack
isub      ; Subtract variable 3 from variable 2 and push result back
iconst_4  ; Push constant value 4 onto the stack
iadd      ; Add constant value to the previous result
istore_1  ; Store result in local variable 1
```

Table 1: Example code for a stack machine – JVM

2.1.2 Register Machines (RM)

The computation model of register machines very closely resembles that of most real microprocessors. A register machine has a fixed number of registers (which might also be dedicated to certain types of data). All operations have to specify source registers which contain the operands and a destination registers into which the result is to be stored. The register numbers are encoded into the instruction itself. Therefore it is in most cases impossible to encode an instruction in a single byte. The implementation of register machines can be problematic on some host architectures: some processors might have a smaller number of registers than specified in the VM, other processors might have much more registers than needed by the VM. In both cases, an efficient implementation can be very difficult.

The two program examples show the code of the example calculation for the ARM RISC processor [Jag96] (which is not a virtual machine but an exemplary register machine) and in the assembly language of Tao Systems' VP (Virtual Processor) [Tao00]:

```
SUB     R1,R2,R3
ADD     R1,R1,#4
```

Table 2: Example for a register machine (1) – ARM

```
cpy.i   ((i2 sub i3) add 4),i1
```

Table 3: Example for a register machine (2) – VP

Though it looks like VP can solve the problem in a single instruction, it is of course split up by the assembler into separate operations.

2.1.3 Memory Machines (MM)

Memory transfer machines [WP97] – or simply memory machines – work in a similar way as register machines do. The only difference is that they do not use any registers but always directly operate on memory locations. Memory machines combine advantages of stack machines and register machines. Just like a stack machine, a memory machine makes no assumptions about the register count of the host processor. On architectures with many registers, it is possible to use these registers as a cache. Architectures with a small number of registers can at least partially cache the memory contents in registers or can alternatively use regular memory. The structure of a memory machine is still very similar to the structure of a real microprocessor; there is no complicated stack-to-register mapping as is required for stack machines (see [Ran98, §2.4, §4.6]).

The example shows assembly code for the Dis VM, produced by the Limbo compiler. The code shows that the variables `v1`, `v2` and `v3` are stored at offsets 40, 32 and 36 in the current stack frame, the code also uses a temporary variable at location 44:

```
subw    36(fp),32(fp),44(fp)
addw    44(fp),$4,40(fp)
```

Table 4: Example for a memory machine – Dis VM

2.2 Memory Models

Virtual machines can be interfaced with the real machine memory in different ways.
There can be either direct access to the memory or indirect access. Direct access means that the virtual machine operates on real memory addresses. Virtual machines with direct memory access can read and write a particular memory location, which allows for low-level system programming but also bears an inherent security risk. VP and the P-Machine allow direct memory access.
In virtual machines with indirect memory access the memory manager completely shields the virtual machine from the real memory. That is, the VM either does not offer instructions for accessing real memory locations or it does not use real memory addresses. As an example, the Java VM only operates on so-called "references" which refer to the memory associated with an object. However, the reference is not necessarily a real memory address (though, in some implementations it may be one), Sun's classic JVM uses "handles" which then point to the object's memory [LY97].

Both direct and indirect memory models can be supported by a garbage collector [JL96] which automatically reclaims unused memory. Functionality and implementation styles of garbage collectors are discussed in greater detail in section 4.3.

2.3 Other Characteristics of Virtual Machines

There is a number of other features that a virtual machine can provide:

- Support for Object Oriented Concepts
  Some virtual machines only provide a processor model which is similar to that of a real processor, whereas other ones are still object-oriented down at the lowest level. Further on, a VM model will be considered object-oriented if it at least offers direct support for classes[5], inheritance (with overridden methods) and "virtual" method calls, ie, method invocations with dynamic method lookup.

- Type Safety on Instruction Level
  The execution engine of a VM may have substantially more information about the data types being processed than a real-world microprocessor. Increased type safety may be achieved by typed registers and/or typed instructions.

- Support for Multithreading and Synchronization
  Multithreading and synchronization can be supported to a different extent and by different means.

- Basic support for Unicode processing
  Localization or internationalization of software becomes a more and more important aspect of software development. As Unicode is a quite new technology it is only supported by more recent virtual machines such as Dis and the JVM.

[5]This also holds if the particular VM does not use the term "class" for this concept.

Based on these and other examinations it is also possible to make a statement about the "abstraction level" of a virtual machine – which, of course, always remains a relative and not necessarily objective measure.
A very high abstraction level can be observed with the Java and Smalltalk virtual machines: their processor model is object-oriented and has nothing in common with real existing microprocessors, access to any physical resources in only possible through the virtual machine.

Another important point is the self-containedness of the virtual machine. The question here is, what belongs to the VM and what belongs to its surrounding environment – and how cleanly are these things separated. It turned out that this actually an area where most virtual machines are lacking (see 6).

2.4 Overview of the Particular Virtual Machines

Sections 2.4.1 to 2.4.5 give a brief introduction to the structure of the examined virtual machines and also discuss any extraordinary features that the VMs offer.

2.4.1 The Java Virtual Machine (JVM)

Due to its enormous popularity the Java Virtual Machine (JVM) is undoubtedly the best-documented of all virtual machines [LY99, MD97, Ven99, Dal97]. The JVM is a stack machine with additional registers (called "local variables"). The JVM provides an object-oriented processor model which also includes exception handling. These features are implemented at a low level in the machine model, ie, there are special machine instructions for OO features and exception handling. The VM supports multi-threading and thread synchronization with monitor locks and condition variables. An extensive security system with a bytecode verifier controls access to the underlying hardware and OS. There is also a reflection API and the possibility of dynamic class loading at runtime. However, dynamic class loading and reflection are not implemented as low-level operations but are provided by native libraries. That is, there are no JVM instructions which allow to control these features. A similar observation can be made for multi-threading. The complete threading API is provided by native methods and is not a real part of the VM specification. Only instructions for thread synchronization (`monitorenter/monitorexit`) are a real part of the instruction set.

A number of shortcomings can also be observed in the JVM. For instance, there is no way of defining pre-initialized arrays of constants. The constant pool in the class files can only store strings and primitive data types but cannot serve as a general data segment. There is only limited VM-level support for handling arrays; arrays can only be efficiently copied by using native (non-VM) methods.

2.4.2 The Dis Virtual Machine

The Dis[6] VM [VNH00, WP97] is closely associated with the Inferno OS [DPP$^+$97], which evolved from Bell Laboratories former "Plan-9" OS. The Dis VM uses an entirely memory-based computation model. Basically there are only two registers, the module pointer `mp` which points to global memory and the frame pointer `fp` which points to the current stack frame. All operands are either immediate values or memory locations which are addressed through `mp` or `fp`.
The Dis VM is not object-oriented but it offers support for abstract data types (ADTs). However, it has instructions for dynamic loading of new modules (`load`) and for creation and finalization of threads (`spawn/mspawn`, `exit`).
The rich instruction set of the Dis VM offers several groups of specialized instructions: list processing (`cons`..., `head`..., `tail`, `lenl`), string processing (`addc`, `indc`, `insc`, `lenc`, `slicec`) and synchronized inter-thread communication via so-called "channels" (`newc`..., `send/recv`, `alt/nbalt`).

The Dis VM model lacks a single-precision floating point type (which can be a problem when large amounts of float have to be handled and double precision is not really necessary).

2.4.3 The Virtual Processor (VP)

In the early 90s the British software company Tao Systems Ltd introduced their heterogeneous multi-processor operating system "TAOS". In order to achieve processor neutrality TAOS used a portable binary code that was to be executed by the Virtual Processor "VP" [Tao00]. VP code is highly optimized for efficient recompilation to the actual target processor. TAOS was based on the concept of heterogeneous "processor pools": whenever new code had to be executed, a processor was selected from the pool and the VP translator associated with that processor recompiled the code for that processor.
Today, VP is part of Tao Systems' Elate OS which is mainly used for embedded devices.

Unfortunately, the VP specification is not publicly available, and without breaking non-disclosure agreements, not many details can be revealed.
However, it may be said, that the VP uses a register-based computation model (which is no surprise as it is optimized for easy retranslation to real microprocessors) with a small number of type-specific registers for 32-bit and 64-bit integers, single and double precision floats and generic pointers.
Executable code and its associated data are embraced in a so-called "tool" – which is very similar to the object-oriented concept of a class. VP supports virtual method invocation and dynamic tool/class loading (`qcl/qcall` instruction) and support for method inheritance (`parentclass`).

[6]Probably the Dis VM is named after the god of the Underworld in the Greek and Roman mythology (in Greek mythology "Dis" is better known as "Hades"). The "City of Dis" is the city of the damned and is encircled by the river Styx (which happens to be the name of Inferno's resource access and communications protocol); the final entrance to hell is surrounded by a zone called Limbo (which is Inferno's proprietary programming language).

Memory management, multithreading and synchronization are not specified on processor level and are handled through calls to library tools (eg, `lib/malloc`).

With the "intent Java Technology Edition" (JTE) Tao Systems also provides a Java Virtual Machine on top of VP – which is remarkable because in this scenario two (!) virtual machines are between the Java application and the hardware; it is a VM in a VM!

2.4.4 The Smalltalk Virtual Machine

When Smalltalk-80 [GR83] was introduced in the early 80s it was far ahead of its time. Not only that is was completely object-oriented and had exact garbage collection, it was also translated into machine-independent bytecode which is executed by a runtime interpreter or dynamically compiled.

The specification of the Smalltalk VM can be found in chapters 28 and 29 of [GR83]. The VM uses a stack computation model and performs almost all calculations by invocation of a method. Therefore the instruction set can be divided into only four basic groups:

- pushing/popping/storing object references
- performing jumps and branches
- invoking a method
- returning a value from a method

Every Smalltalk object has a number of slots that can contain data (ie, instance variables) or method pointers. In Smalltalk jargon a method or an instance variable is accessed by "sending a message" to the object.
There are some "special" messages (such as `new`, `size` or `==`) and a number of "primitive" methods (such as `+`, `-` and `*` for `Integer` objects, or `wait` for `Semaphore` objects). Amongst all considered virtual machines the Smalltalk VM probably is the "most" object-oriented one: it has the purest object model and makes use of objects very consistently.

2.4.5 The P-Machine

Probably the first commercially successful virtual machine was the P-Machine, which served as a platform-neutral runtime system for UCSD Pascal [Bow78]. UCSD Pascal was developed at the University of California at San Diego and was based on Niklaus Wirth's P2 and P4 Pascal compilers. The UCSD Pascal compiler generated machine-independent intermediate code – the so-called P-Code. The P-Code could either be interpreted by the P-Machine or recompiled into native code. Together, the P-Machine, the UCSD Pascal compiler and a set of OS libraries were known as the UCSD P-System.

The P-Machine was released in several different versions (I.1, I.2, I.3, II, III.1, III.2, IV, V, etc) which once were well documented but are no longer available or reconstructable today.[7] The P-Machine which is presented here is based on specifications taken from [WM92] and [NAJ+81].
It must be noted, that the machines described in [WM92] and [NAJ+81] are not identical, and probably neither of them is equivalent to the "original" P-Machine (though [NAJ+81] is probably closer).

Though still a pretty modern piece of software in its time, the P-Machine was by far not as advanced as the Smalltalk VM. The P-Machine was neither object-oriented nor did it support garbage collection, multithreading, exceptions or dynamic code loading (except for the initial loading of a program into the interpreter).
Like Smalltalk and Java, the P-Machine uses a stack computation model. Its instruction set has the standard instruction groups for arithmetic, logic, comparisons, load/store, array index calculation, branches/jumps and a `new` instructions for allocating memory. The P-Machine described in [NAJ+81] also has instructions for set operations (generate singleton set – `SGS`, set intersection – `INT`, set union – `UNI`, set difference – `DIF`, test set membership – `INN`). That specification also contains a number of primitive procedures for mathematical functions and I/O (similar to the Smalltalk primitive methods).

2.5 Tabular Summary

Table 5 summarizes the general features of the examined virtual machines. A "♦" indicates, that the functionality is fully supported, "◊" means that a functionality is not completely implemented in the VM or provided by some VM-external library or service. If a VM does not at all support a particular feature this is marked with "–".

[7] On his web pages for "The UCSD P-System Museum" (http://www.threedee.com/jcm/psystem/) John Foust writes: "I asked Prof. Wirth if he still had the source code to any of these early Pascal compilers. At first he said "no". Later he found an old CDC 6400 7-track mag tape with the source, but we haven't found a way to read it yet..."

Virtual machine	JVM	Dis VM	VP	P-Machine	Smalltalk VM
Primary language	Java	Limbo	C	Pascal	Smalltalk
Computation model	SM	MM	RM	SM	SM
Memory model	indirect	indirect	direct	direct	indirect
Garbage collection	♦	♦	–	–	♦
Bytecode verifier	♦	◊[1]	–	–	–
OO support	♦	–	♦	–	♦
Multithreading support	♦	♦	♦	–	♦
Dynamic code loading	♦	♦	♦	–	♦
Reflection	◊	–	–	–	♦
Exceptions	♦	–	–	–	–
Basic Unicode support	♦[2]	◊	–	–	–

[1]The Dis VM uses cryptographic signatures in order to verify the validity of unknown code.

[2]Right now, Java has much more than just "basic" Unicode support; earlier versions, however, needed additional external support [Ran97] in order to be fully Unicode-compliant.

Table 5: Features of the considered virtual machines.

Besides the standard instruction groups for arithmetic, logic, branching, and so on, most VM instruction sets also offer specialized instructions for certain data types which are characteristical for a particular VM. Table 6 shows these additional instruction groups. It must be noted that the particular cells will only marked "♦" if there are actual VM instructions available for the functionality; for instance, Java does not have support for reflection on instruction-level and is therefore marked "–".

"Inter-thread communication" means a safe way of transmitting data from one thread to another (Java's `PipedInput/PipedOutputStreams` are a good example).

VM	JVM	Dis	VP	P-Machine	Smalltalk
Memory allocation	♦	♦	–	♦	♦
Array processing	♦	♦	–	♦	♦
List processing	–	♦	–	–	–
Set processing	–	–	–	♦[1]	–
String processing	–	♦	◊	–	♦
Exception handling	◊	–	–	–	–
Thread control	–	◊	–	–	♦
Synchronization	♦	◊[2]	–	–	♦
Inter-thread communication	◊	♦[2]	–	–	–
Virtual method invocation	♦	–	♦	–	♦
Object access	♦	–	–	–	♦
Typechecking	♦	♦	–	–	♦
Dynamic code loading	◊	♦	♦	–	♦
Reflection	–	–	–	–	♦

[1]Set instructions are only listed in [NAJ+81].
[2]In the Dis VM "channels" are used for synchronized inter-thread communications.

Table 6: Special instruction groups.

3 Separation of Virtual Machines into Independent Components

In order to create a generic framework for the implementation of virtual machines, the virtual machine model will now be separated into – relatively – independent components. VM components will always have to use services provided by other component, so there can never be an absolute independence.

Splitting a virtual machine into separate components is to some degree a very arbitrary process. There are many different ways of how this can be done, and for all these ways there may be conclusive reasons.
Therefore the components and interfaces listed in section 4 can only be proposals, and do not claim to be the one and only way of defining VM components. Sections 3.1, 3.2 and 3.3 present reasons which motivate and justify the chosen design.

3.1 Design Philosophy for the Framework

The main design principle of the framework is to provide an open and flexible specification which lets enough room for different types of virtual machines and does not impair the development of VMs due to over-specification.
In particular, this means:

- No assumptions are made about the concrete structure of a virtual machine, ie, the suitability of an interface should not depend on a certain type of implementation (eg, register machine with garbage collection).
- Only those interfaces appear in the specification which need to be accessed by other components. "Internal" data structures are not listed.
- Some interfaces are intentionally left empty. That is, the developer has to provide suitable implementations for them but must define all attributes and methods himself. However, the interfaces are necessary for passing around these abstract data items within the framework.
- The interfaces also do not stipulate complete class hierarchies. Definition of subclasses is in the responsibilty of the implementor – and is very likely to be required for an operational VM. For instance, the security interface defines only a base class `Permission`, but no subclasses (as can be found in the Java security library).
- Whereever possible, functionality is specified in a way which makes use of other services as little as possible. For instance, security violations are not signaled with exceptions but by returning booleans, as a particular VM might not support exceptions.

Another important design principle for the framework is to minimize calls to other components and to ensure that a component can be completely implemented by only using (a) calls to other components in the framework and (b) calls to the underlying operating system.
All VM components are interfaced with the OS on one side and with the other VM components on the other side (see figure 3). The framework described in section 4 only defines the interfaces between the particular VM components and not the interface to the OS.[8]

In addition to all that, functionally related services should go into the same component (eg, the Java bytecode verifier and the security manager would go into the same component, though they are located in different modules of the Java runtime system).

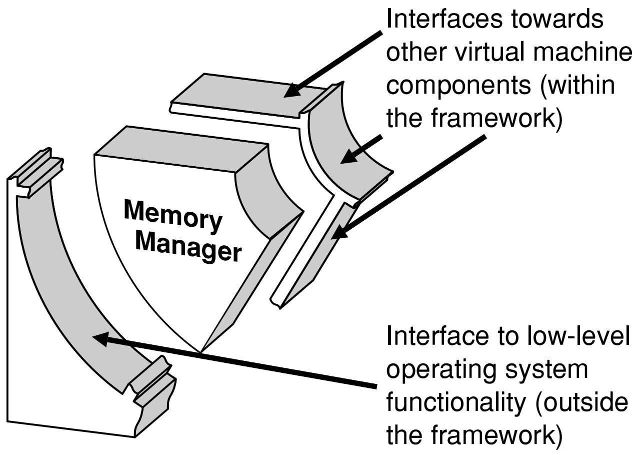
Interfaces towards other virtual machine components (within the framework)
Memory Manager
Interface to low-level operating system functionality (outside the framework)

Figure 3: Interfaces towards VM and OS (example: JVM memory manager).

3.2 Analysis of Services in Existing Virtual Machines

Intuitively it is possible to identify certain related services in a virtual machine, but in order to obtain a complete view of things a couple of VMs have been examined for possible component candiates (see section 2). Because of its rich documentation (see section 2.4.1) the Java Virtual Machine will play an important role here.

Unfortunately, [LY99] does not explicitly define the particular compontents of the JVM. However, the specification makes implicitly clear what functional units will be needed for a JVM. In [MD97, chapter 3] the components of the JVM are summarized as follows:

- Execution engine
- Memory manager

[8]A good example for an OS interface is the Java Host Programming Interface (HPI; mentioned, for instance, in [LY97]). It is part of the Java porting layer and only accessible to Java source licensees; however by issueing the following UNIX command in the JDK base directory it is possible to get a fair impression of how the HPI looks like:
`nm jre/lib/i386/native_threads/libhpi.so | grep 'T sys'`

- Error and exception manager
- Native method support
- Threads interface
- Class loader
- Security manager

For a more detailed picture it is necessary to examine which services each of these components actually provides:

- The execution engine forms the "heart" of the JVM; by executing the instructions from the bytecode stream it provides the actual processor emulation.
- The memory manager is responsible for memory allocation and automatic garbage collection.
- The error and exception manager performs exception propagation and catches exceptions; it also manages the data structures which are necessary to do this.
- The native method support allows the implementation of functions that cannot be performed inside the VM itself; it interfaces the JVM with the C and C++ programming languages.
- The threads interface enables multi-threading and takes care of the thread scheduling and the management of thread-related data structures. To a certain extent it also provides synchronization mechanisms and inter-thread communication.
- The class loader provides dynamic class loading and links new code with already loaded code. It is also responsible for resolving symbolic data from the class files into runtime data structures and performs Java's bytecode verification.
- The security manager checks permissions for critical functions. From JDK 1.2 onwards it also maintains `ProtectionDomains` and does stack frame checking (`Access Controller`)

Some services are actually misplaced a little bit:
Bytecode verification is a security issue and is therefore a function of the security manager. The propagation and handling of exceptions is actually part of the code execution and should therefore lie in the responsibility of the execution engine (in addition to that, exceptions are not necessarily an essential VM concept).

For distributed JVMs the scenario might look a little bit different. In [SGGB99] the authors suggest additional separate services:

- Verification

- Remote monitoring
- Compilation

Separating out these services is very specific to distributed virtual machines (eg, because an embedded device might not have the resources to perform bytecode verification and might therefore pass this task to some other machine in the network). Except for remote monitoring the listed functionality can be placed in the security manager (verification) and execution engine (compilation).

At least as a language feature the Java VM also supports reflection. This is not part of the actual VM specification (as it is provided by native methods in the class library), but this important functionality must definitely be considered. The Smalltalk VM also supports reflection through its purely object-oriented model.
Thus, a reflective system should be added as a further component of virtual machines.

The other virtual machines – Dis VM, VP, and P-Machine – do not state any specific grouping into functional units in their specifications. However, none of them offers additional services which are not covered by the above list.

3.3 Proposed General Virtual Machine Components

The services which were identified in section 3.2 can be regrouped into functionally secluded modules shown in table 7. The table also lists additional responsibilities, such as the definition of stack frame and object format. Section 4 describes the components in more detail.

Component	Responsibilites
Execution Engine	• Execute code / emulate processor • Perform exception propagation, catching and management • Provide low-level language safety (eg, type and array check) • Define stack frame layout
Memory Manager	• Allocate and deallocate memory • Perform garbage collection (if supported) • Finalize objects • Define object memory layout
Link Loader	• Enable dynamic class loading • Link new code with already loaded code • Resolve symbolic references to runtime data structures • Provide binding and lookup of symbolic references • Manage namespace(s)
Thread System	• Enable multithreaded program execution • Allow creation and management of threads • Manage thread scheduling • Provide mechanisms for thread synchronization • Provide mechanisms for inter-thread communication
Security Manager	• Check access permission for resources • Perform stack traversals for security context checks • Verify incoming code
Native Support	• Allow access of functionality outside the VM • Define calling and naming conventions for native code
Reflective System	• Enable reflection and reification

Table 7: Proposed virtual machine components and their responsibilities.

4 Proposed Component Interfaces

Based on the analysis in section 3.2 and the proposals from section 3.3 there are seven components, each of which defines one or more interfaces. Each component has one “main” interface and possibly additional helper interfaces. The “component interface” is the entirety of all interfaces: the main interface plus any additional interfaces. The seven component interfaces are named processor interface, memory interface, linkloader interface, thread interface, native interface, security interface and reflective interface. It is not always the case that the name of the overall component interface corresponds with the name of its main interface (eg, the main interface of the processor component is called `ExecutionEngine` and not `Processor`).
The framework itself requires additional interfaces in order to allow communication between the VM components.
Figure 4 shows a simplified overview of the framework with some possible component implementations. The figure uses the OVAL notation (see [Ran00] or (http://oval.ac); the diagram is simplified in several aspects: actually there is more than one interface per component, and of course these interfaces have more or less than five methods).

The interfaces are specified in the Interface Definition Language (IDL) which is defined in [OMG99, chapter 3]. For most common programming languages an IDL mapping is available, thus the framework can be implemented in almost every major implementation language.

Every section on a particular VM component (4.2 – 4.8) begins with a more detailed analysis of the required functionality and ends with the IDL source code for the interface specification. For sake of brevity, the necessary forward declarations and `#include`s have been omitted from the IDL code. The IDL code contains documenting comments, but the key functions are also separately described in the accompanying full text.
Except for `string`s (which are considered immutable) all non-primitive IDL parameters are passed in `inout` mode, that is, a function may modify an object which it gets passed as an argument (if the object allows such a modification).
Though IDL supports Unicode strings with its `wstring` type, regular 8-bit strings are used in the IDL files. In most implementation languages (eg, C) Unicode strings still require additional libraries or are likely to impose an overhead on string operations. However, the 8-bit entities which are stored in IDL `string`s are considered to be in UTF-8 encoding, which guarantees that no Unicode information is lost.

Almost any of the listed functions may be left unimplemented for a concrete component implementation. If certain functionality is not applicable to a certain VM, the corresponding functions should just silently return. A component must always be prepared to get a null pointer / null reference from another component when certain information is not available.

Design inspirations were found in the following sources:

- Processor: [LY99, Lia99]
- Memory: [LY99, VNH00, JL96, AAB+00, WG98, ADM98, Age98, WDH89]
- Linkloader: [LB98, VNH00, HO91, Fre96, DSS90]
- Threads: [LY99, VNH00, AAB+00, LB96]
- Native: [Lia99, GR83, NAJ+81]

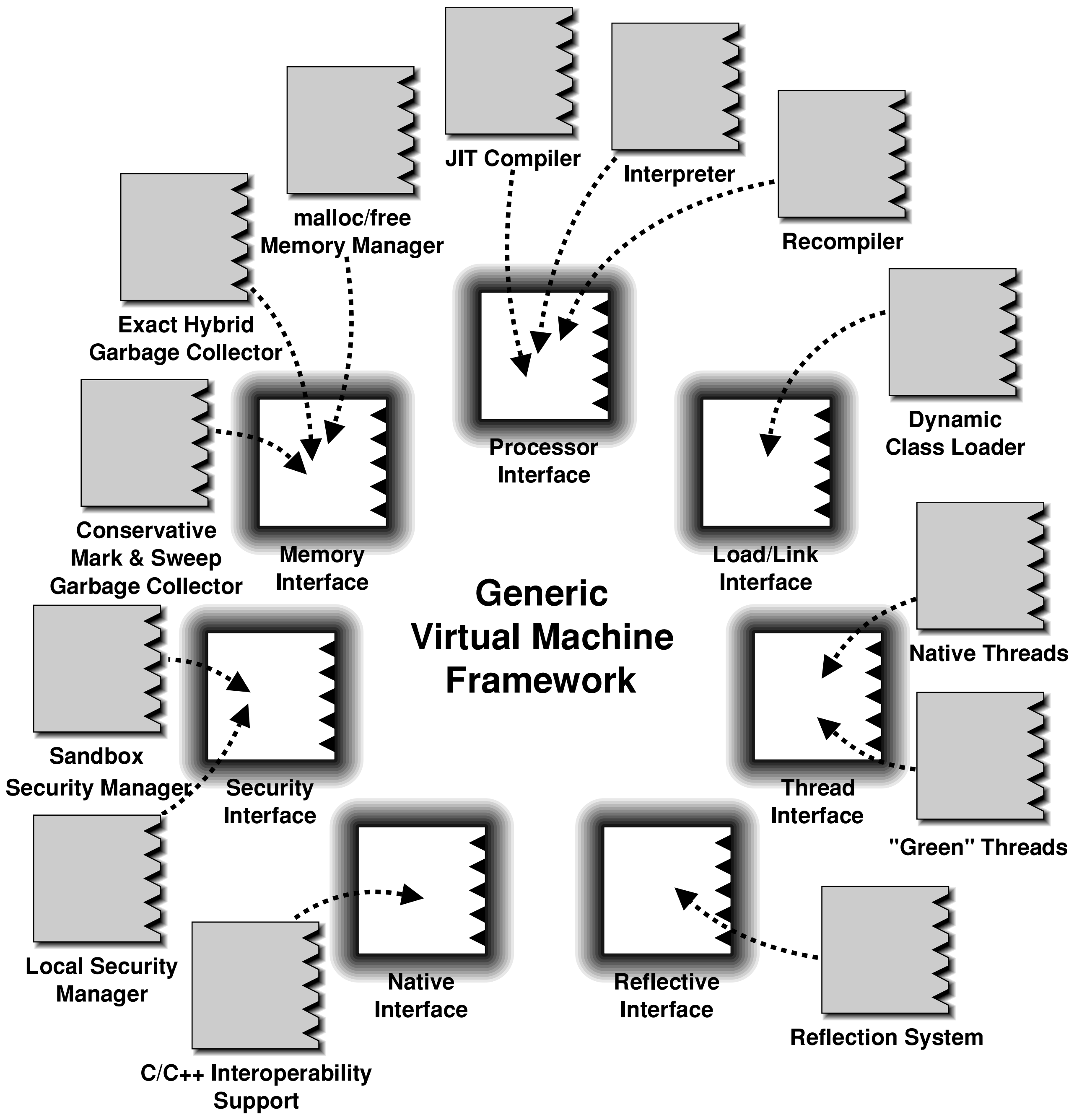

Figure 4: A Generic Virtual Machine Framework with possible implementations.

- Security: [LY99, Ven99]
- Reflection: [GR83, AGH00, LY99, KG96, GK98, Gol97]

4.1 Framework Management

Components inside the framework must know about each other and the framework must provide functions which allow to locate a particular component. The framework is the entity which holds all components together and is also the main means of communication between the particular components.

4.1.1 Required Functionality

The framework defines the common communications interface (`gvmf::Framework`) and the base interface for all VM components (`gvmf::VirtualMachineComponent`). It also provides a general interface for error handling (`gvmf::Error`) and the `StringArray` type (which is necessary, because IDL does not allow `sequence`s as return types; see [OMG99, §3.12]).

For general communication within the framework at least the following functions are required:

- Starting the framework (with optional arguments)
- Shutting down the framework
- Locating a particular component within the framework
- Replacing a particular component

The latter function gives the possibility to change a component during runtime. In order to implement this functionality in a consistent way, the components need additional functions. Besides initializing a VM component (`initialize`) there must also a way to shut down a single component. As other components might depend on that component the component may also refuse to shut down. It can do this by simply returning `FALSE` when its `exitRequested` function is called. A component is also notified when the framework wants to replace another component in the framework (`componentChangeRequested`) and when such a replacement is actually carried out (`componentChanged`). Just like refusing its own termination a component can also object to the termination of another component on which it might depend (by returning `FALSE` as response to the change request).[9]
The interface `VirtualMachineComponent` also defines functions for obtaining a general information string (eg, with a version identification) and for querying the error status of a component.

[9]The dynamic replacement of VM components has not been tested yet and it might turn out that there are more interface functions necessary to fully enable this functionality.

4.1.2 IDL Specification

```
// framework.idl
// Mirko Raner, 17.08.2000
//
module gvmf
{
    typedef sequence <string> StringArray;

    // Main interface of the virtual machine framework
    interface Framework
    {
        // Starts the framework
        boolean start(inout StringArray arguments);

        // Exits the framework
        boolean shutdown();

        // Returns a reference to the processor main interface
        gvmf::processor_::ExecutionEngine getExecutionEngine();

        // Returns a reference to the memory main interface
        gvmf::memory_::MemoryManager getMemoryManager();

        // Returns a reference to the linkloader interface
        gvmf::linkloader_::LinkLoader getLinkLoader();

        // Returns a reference to the thread main interface
        gvmf::thread_::ThreadSystem getThreadSystem();

        // Returns a reference to the security main interface
        gvmf::security_::SecurityManager getSecurityManager();

        // Returns a reference to the native main interface
        gvmf::native_::NativeSupport getNativeSupport();

        // Returns a reference to the reflection main interface
        gvmf::reflective_::ReflectiveSystem getReflectiveSystem();

        // Replaces the execution engine component
        boolean setExecutionEngine
            (inout gvmf::processor_::ExecutionEngine engine);

        // Replaces the memory manager component
        boolean setMemoryManager(inout gvmf::memory_::MemoryManager memory);

        // Replaces the linkloader component
    boolean setLinkLoader(inout gvmf::linkloader_::LinkLoader loader);

    // Replaces the thread system component
    boolean setThreadSystem(inout gvmf::thread_::ThreadSystem threads);

    // Replaces the security system component
    boolean setSecurityManager
        (inout gvmf::security_::SecurityManager security);

    // Replaces the native support component
    boolean setNativeSupport(inout gvmf::native_::NativeSupport ns);

    // Replaces the reflection component
    boolean setReflectiveSystem
        (inout gvmf::reflective_::ReflectiveSystem reflection);
};

// Error status interface
interface Error
{
    // Return the message of the last error
    string getErrorMessage();

    // Clear the error (ie, reset the error status)
    void clear();
};

// VM component base interface
interface VirtualMachineComponent
{
    // Reference to the framework; this is the only pre-defined attribute
    attribute Framework frameworkHandle;

    // Initialize component
    boolean initialize(inout Framework handle, in string arguments);

    // Prepare to exit; return FALSE if component cannot exit (yet)
    boolean exitRequested();

    // Get notified of a component change request
    boolean componentChangeRequested(inout VirtualMachineComponent comp);

    // Get notified of a component change
    void componentChanged(inout VirtualMachineComponent comp);

    // Get general information string (eg, with version identification)
    string getInformation();
```

```
        // Get current error status of component
        Error getErrorStatus();
    };
};
```

4.2 Execution Engine

Probably the most important part of a virtual machine is the execution engine, though it requires a lot of support from other VM components in order to be of any use.
The execution engine can be implemented in many different ways, which makes it hard to define a common interface for it. The processor interface is therefore only very loosely specified.

4.2.1 Processor Implementation in Existing Virtual Machines

There are at least three different ways of implementing the execution engine of a virtual machine (see, eg, [Ran99a]): pure interpreters, interpreters with JIT compiler support and pure recompilers (either JIT or "ahead of time"). For the Java Virtual Machine all three types of implementation can be found. The Dis VM strictly uses either the interpreter or recompiles a module ahead of time – depending on the module's `runtime_flag`. Completely recompiler-based is the Virtual Processor: for each target microprocessor there is a "VP translator" which compiles the VP code to native code.
A look into the original Smalltalk-80 documentation [GR83], tells the reader that Smalltalk has an interpreter and a compiler, but things are not always what they look like: Smalltalk uses either a source code (!) interpreter or a bytecode interpreter which executes compiled source code; the Smalltalk compiler compiles source code to bytecode, not bytecode to native code. Later Smalltalk systems also added the possibility of native compilation.
Though most implementations of the original P-Machine were interpreter-based, also a couple of recompiler-based implementations existed (eg, [CY78]).

4.2.2 Program Execution Control

The framework does not assume that the execution engine is implemented in a particular manner, it may even be a hybrid implementation like an interpreter plus a JIT compiler.

A very flexible design for invoking code, which also allows for interoperability between interpreters and compilers, is the concept of "invokers" which is briefly mentioned in [Yel96] and also appears in some places in [LY99]. For executing a method not only the method's bytecode (and maybe some other runtime data structures) is required, but there is also the need for a so-called "invoker". The invoker contains code and data which is required to transfer control from the caller to the callee – which can be quite complicated, for instance if the caller is recompiled and the caller is interpreted (in old Java VMs, for instance, there

are additional problems like mixed-style JNI/NMI native methods, etc).
The invoker contains information about how a method is executed: for example, by regular or "virtual" method invocation. But there are much more different invokers conceivable. The Virtual Processor, for instance, has different method invocation styles which control the dynamic loading of the required "tool" (for information on VP see section 2.4.3).

For the virtual machine framework the invoker concept has been included as `ExecutionMetaData` interface in the `gvmf::processor_` component. Control can be transferred in two different ways: there is a "bootstrap" call (`main`) which is only called once when the execution engine is started (it is similar to the `main` method in C, C++ or Java).
The generic `execute` method takes a `ResolvedItem`, an `ExecutionMetaData` object and a `MemoryArray` (for arguments, including an optional "this" reference) as parameters and returns a `Memory` object.
`ResolvedItem` is a generic interface for runtime data structures (such as method blocks) and the `Memory` interface encapsulates a piece of memory which stores an object. These structures are described in section 4.4 and 4.3.

4.2.3 Stack Frame Functions

Another important responsibility of the processor interface is the definition of the stack frame layout. The actual layout of a stack frame is hidden but the interface must allow other components to perform certain operations on a stack frame: backtracking to the caller stack frame (`getPredecessor`), getting a memory reference for a particular stack frame (`getMemory`) together with the type information for this frame (`getType`) and obtaining the security context of a frame.
Memory reference and type information are required by the memory manager (section 4.3), the security context might be inquired by the security manager component. As already pointed out, the latter three functions might also not be implemented at all (and silently do nothing).

4.2.4 IDL Specification

```
// processor.idl
// Mirko Raner, 13.10.2000
//
module gvmf
{
    module processor_
    {
        // Invocation meta data ("invoker") interface
        interface ExecutionMetaData {};

        // Interface for a particular stack frame
        interface StackFrame
        {
            // Return the predecessing stack frame (the caller frame) or
            // NULL if this is the first frame in a thread
            StackFrame getPredecessor();

            // Return the type information of the stack frame
            gvmf::memory_::Type getType();

            // Return a reference to the memory where the frame is
            // currently stored
                   gvmf::memory_::Memory getMemory();

            // Return a reference to the frame's security context (if any)
            gvmf::security_::SecurityContext getSecurityContext();
        };

        // Main execution engine interface
        interface ExecutionEngine: gvmf::VirtualMachineComponent
        {
            // Call the "main" method
            long main(inout gvmf::StringArray arguments);

            // Transfer control to other code
            gvmf::memory_::Memory execute
                (inout gvmf::linkloader_::ResolvedItem code,
                 inout ExecutionMetaData md,
                 inout gvmf::memory_::MemoryArray arguments);
        };
    };
};
```

4.3 Memory Manager

The memory management of the considered virtual machines basically falls into two different categories: automatic memory management (ie, garbage collection) or the classic malloc/free memory management. While the classic approach can be boiled down to the three basic functions `malloc`, `realloc` and `free`, automatic memory management requires a number of additional helper functions, depending on what type of garbage collection algorithm is used.
A VM framework must provide a generic superset for all types of memory management strategies, ie, every strategy should be implementable within the framework.

While memory management with the classic malloc/free style requires a only a very simple API, garbage collection algorithms require different levels of support. Conservative garbage collection [JL96, BW88] (as opposed to exact garbage collection [SLC99, ADM98, WG98]) requires the smallest amount of support. A conservative garbage collector only guarantees

that it will not remove objects which are still referenced. However, it does not guarantee that all objects which are not referenced any longer will eventually be garbage collected. That is, there might still be minor memory leaks in such a system.
The classic JVM (as used until JDK 1.1) had a conservative garbage collector, and under certain circumstances failed to reclaim unused memory.
As a conservative garbage collector does not necessarily have type information and information about which memory cells contain pointers and which contain data, it must assume that any value that is a valid address of an object on the heap is indeed an object reference.

If exact garbage collection is required, additional information must be passed to the memory manager. It is no longer sufficient to pass the size of the requested piece of memory when new memory is requested. Every allocated piece of memory must also have a type assigned, which allows the garbage collector to determine where pointers to other data are located. The Dis VM and later Java VMs (such as the EVM [WG98], which is incorporated into the Java 2 SDK for Solaris) support exact garbage collection. Both VMs use pointer maps (or reference maps) that indicate which words within an object reference other objects.

Both, conservative and exact garbage collection can be implemented in different ways. The most common approach is so-called "mark & sweep", but there are also systems that use reference counting or a hybrid solution of both approaches (see [JL96] for detailed information on the algorithms).
Reference counting has its problems (for instance, reclamation of cyclic data structures) but in combination with other mechanisms it absolutely makes sense. The Dis VM, for instance, uses reference counting as its main GC algorithm and occasionally cleans up cyclic garbage by an additional mark & sweep [DPP$^+$97, HW98].
In reference-counted systems not only the memory allocation as such, but also all pointer manipulation operations require interaction with the memory manager. As the memory manager maintains the reference count (RC) for all allocated objects, it must also provide functions for adjusting the counters.

Section 4.3.1 gives an overview of the memory management functions of the considered virtual machines.

4.3.1 Memory Management Operations in Existing Virtual Machines

Table 8 shows the memory operations which are supported by the considered VMs:

VM	Instruction	Effect
JVM	new	allocate memory for an object
	newarray	allocate memory for a primitive array
	anewarray	allocate memory for an object array
	multianewarray	allocate memory for a multidimensional array
Dis VM	new	allocate memory for an object
	newz	allocate memory for an object and set all cells to 0
	mnewz	allocate memory for an object from another module
	newa	allocate memory for an array
	newaz	allocate memory for an array and set all cells to 0
	movm	copy memory (memcpy)
	movmp	copy object; corresponds to `Object a,b; a=b;`
	movp	copy pointer; corresponds to `Object *a,*b; a=b;`
	frame	allocate new stack frame for local call
	mframe	allocate new stack frame for inter-module call
VP	malloc	allocate generic memory (standard C library call)
	realloc	reallocate generic memory (standard C library call)
	free	free generic memory (standard C library call)
P-Machine	new	allocate raw memory
Smalltalk VM	new	allocate memory for an object or array

Table 8: Memory-related virtual machine instructions.

The Dis VM uses both reference counting and mark & sweep and therefore needs a large number of operations dedicated to memory management. Memory can only be addressed indirectly through the frame pointer FP (local variables) or the module pointer MP (global/ static variables). The Dis VM does not shield the applications from the implementation of the memory management, whereas, for instance the JVM memory operations remain completely neutral versus the implementation of garbage collection. If the JVM was to be implemented with additional reference counting support, this functionality would go into JVM instruction like `iload` (increase RC) or `pop` (decrease RC). However, the JVM uses pure mark & sweep and therefore these instructions do not have any overhead for RC-adjustment. The disadvantage is that the mark & sweep garbage collector has to do a lot more work and will take much longer than the mark & sweep in the Dis VM.

The VP specification actually does not contain any memory management. VP completely relies on native library functions such as `lib/malloc` and `lib/free`.

Both the P-Machine and the Smalltalk VM only contain a single memory allocation opcode `new` and implicitly assume that all unused memory is automatically garbage-collected (by a garbage collector which needs no additional support).
While the Smalltalk VM indeed has a garbage collector, this feature is actually missing in most implementations of the early P-Machine. The P-Machine assumes that the programmer does not recklessly allocate new chunks of memory and that once all data structures are

allocated no more new memory is needed. Of course, this assumption fails for long-running applications such as server processes.

4.3.2 Required Memory Management Functions

In order to support VMs which use a direct memory model, there is a need for a function that directly allocates and frees a piece of memory (called `allocate` and `free` in the IDL code). VMs which directly use standard C library functions also require to shrink or enlarge an allocated block (`reallocate`).
For reasons of efficiency there should be also a function for fast copying of memory; there is a low-level function which does not adjust any GC-related information (`memcpy`) and a high-level function (`overwrite`) which, eg, adjusts reference counts.
In contrast to the standard C library functions, which only take the size of the requested memory, the interface functions also require the type information for the memory. The IDL code specifies the type information as an abstract data type (`Type`), this data type can, for instance, be implemented as a descriptor string (like used in the JVM) or as a binary descriptor with a pointer map (used by the Dis VM). VMs that do not make use of garbage collection can simply pass a generic type information (eg, "`void*`"). Arrays can be handled in two different ways: either the `amount` argument of the `allocate` call defines the size (which requires contiguously allocated arrays), or the size is encoded into the type information and the `amount` argument is always 1 (or even ignored). In any case the implementation must be consistent.
All the general allocation and management can be accessed through the interface `Memory-Manager`.
A piece of memory which is used to store an object or some other data is represented by the interface `Memory`.

Virtual machines that use reference counting require additional support. The Dis VM, for instance, encapsulates the updating of reference counters in the instructions `movp` and `movmp`. The functionality of `movmp` is covered by the interface function `overwrite`. The `movp` instruction copies a pointer and adjusts the reference counts of the involved objects. It does not make sense to include such a function in the interface of the memory manager, because object pointers are not necessarily located in memory but could as well be stored in a processor register (for the Dis VM it makes sense because it is implemented as a memory machine; see sections 2.1 and 2.4.2). Instead of a compound pointer manipulation function the interface contains explicit reference counting functions (`incRC` and `decRC`).
VMs with a direct malloc/free memory model or merely conservative garbage collection must a least be able to determine the size of the memory block (`getSize`); when support for exact garbage collection is required the type information (as required by the garbage collector) needs to be associated with a block of memory (`getType`). The `Type` interface allows a traversal of all pointers within a piece of memory (`getNumberOfPointers`, `getPointerValue`).
As for memory allocation, malloc/free-style memory managers can always return a generic dummy type information.

Mark & sweep garbage collection is supported by the `mark` and `isMarked` functions in the `Memory` interface.
Finally, the `MemoryManager` interface provides a functions which actually starts the garbage collector (`runGC`). If required, this function can also be started in a separate thread.

A very important point in garbage-collected memory systems is the finalization of certain objects. Objects which contain references to external resources, such as file handles or graphics contexts, need to be finalized in a way which releases the referenced resource. For this purpose, the `MemoryManager` interface provides the call `registerFinalizer`, which allows to register a finalization method (or more general: finalization code) for a certain object. In fact, finalization is not very often used directly, but still is an essential feature; the importance of finalization is aptly described in [WDH89, §3.3.2]: "*in the two million lines of Cedar code in use at PARC, only twelve calls register for finalization [...] However, doing without finalization is not possible: the twelve kinds of finalizable objects in Cedar include stream and network I/O objects, so nearly all applications indirectly use finalization.*"

All memory operations must be handled via the interface functions. This includes also the allocation of new stack frames. For example, the Dis VM uses the same type descriptors for stack frames as for objects. Other type descriptor systems can be easily extended for stack frames.

Section 5.3 shows how a simple mark & sweep collector could be implemented with the framework.

4.3.3 IDL Specification

```
// memory.idl
// Mirko Raner, 04.08.2000
module gvmf
{
    module memory_
    {
        typedef sequence<Memory> MemoryArray;

        // The interface 'Memory' encapsulates a contiguous block of memory
        // which stores a single object or array or an instance of a data
        // structure.
        interface Memory
        {
            // Increase the reference count
            void incRC();

            // Decrease the reference count and reclaim automatically when 0
            void decRC();

            // Return the (net) size of the memory piece
    long getSize();

    // Get the type information of the memory piece
    Type getType();

    // Mark the piece of memory (mark & sweep)
    void mark();

    // Unmark the piece of memory
    void unmark();

    // Determine whether memory is already marked
    boolean isMarked();
};

// The interface 'Type' encapsulates the type information of a
// piece of memory and allows to find pointers in an object.
interface Type
{
    // Determine how many pointers a pointer contains
    long getNumberOfPointers();

    // Extract a pointer from a piece of memory
    Memory getPointerValue(inout Memory block, in long index);
};

// Main interface
interface MemoryManager: gvmf::VirtualMachineComponent
{
    // Allocate new memory
    Memory allocate(inout Type typeinfo, in long amount);

    // Shrink or enlarge an allocated block
    Memory reallocate(inout Memory block, inout Type typeinfo,
        in long amount);

    // Move a piece of memory and adjust the reference counts
    void overwrite(inout Memory source, inout Memory destination);

    // Move a piece of memory without adjusting reference counts
    void memcpy(inout Memory source, inout Memory destination);

    // Return a block of unused memory to the OS
    void free(inout Memory block);

    // Register a finalizer functions for an object
    void registerFinalizer(inout Memory block,
                inout gvmf::linkloader_::ResolvedItem finalizer);

            // Start the garbage-collector
            void runGC();
        };

    };
};
```

4.4 Link-Loader

Executable application code is represented and handled differently in the considered virtual machines. Only the early P-Machine does not support loading and linking at runtime, all other considered VMs allow dynamic loading and linking.
Still there is considerable difference in the abstraction level on which application code is presented in the particular VMs. The Java and Smalltalk VMs use a very high-level bytecode, which is essentially a more machine-readable transformation of the source code (indeed, the bytecodes of Java and Smalltalk can be considered object-oriented machine code). For this reason, these VMs have to work with a lot of symbolic references which need to be resolved at runtime.
The Dis VM and the Virtual Processor VP are more like variations of real-world microprocessors and are roughly positioned on the same abstraction level as, for instance, SPARC or Intel x86. The Dis VM uses "link items" which associate a symbolic name with a function entry address and a pointer map for the stack frame layout.

4.4.1 Loading and Linking in Existing Virtual Machines

Except for the P-Machine, all discussed virtual machines support dynamic loading and linking.
The Dis VM allows dynamic loading of modules with the `load` instruction. `load` returns a "linkage descriptor" which is required to allocate data structures (`mnewz`) and call functions (`mframe`, `mcall`, `mspawn`) which are defined by that module.
In the JVM dynamic class loading is provided by the standard libraries (`java/lang/ClassLoader` and its subclasses) supported by appropriate functions in the runtime system; there is no direct JVM instruction which triggers loading of a new class.
Dynamic loading in the Virtual Processor is handled by the same instruction which is responsible for method invocation; this instruction has the capability of dynamically loading the "tool" which provides a certain method; infrequently used tools can even be automatically unloaded after use.

Virtual machines can pursue different strategies for loading and linking. Most JVM implementations, for instance, use "eager" loading and linking. This means, when a newly loaded class refers to other not yet loaded classes, the loading of the referred classes is

triggered as well. Therefore, all classes in memory are always completely linked and resolved[10].
When memory is in short supply a virtual machine might choose a "lazy" loading and linking scheme, where unresolved field and method references are not resolved before they are accessed. Indeed, modern VMs like the Jalapeño VM already make use of genuine lazy loading.
An implementation framework for virtual machines cannot decide in advance, which loading scheme will be used and therefore must provide support for both.

4.4.2 Namespaces

While in virtual machines which more resemble real microprocessors (such as the P-Machine) the concept of a namespace is barely present – because these VMs operate on numeric data instead of symbolic references –, higher level VMs clearly support the concept of namespaces.

A virtual machine can have one flat global namespace or can support nested hierarchical namespaces. Basically, hierarchical namespaces can be converted into flat namespaces by introducing something like "fully qualified names". For instance, in the Java VM a class establishes a namespace, and a method signature is always unequivocal within a particular class. However, if fully qualified names (like, eg, `java/lang/Object/hashCode()I`) are used, the particular class namespaces can be united into one global namespace.[11] Similarly, the Dis VM treats modules as namespaces; fully qualified names consist of a module name and an item name within that module (eg, `Bufio->open`) [Ker00]. The Inferno OS, which is running on top of the Dis VM, provides additional hierarchical namespaces [Sha99].
In many cases, hierarchical namespaces can be managed more efficiently because the namespaces tend to be smaller. It is easier to find a certain method within the namespace of a class, than to find it in one global namespace.[12]

The proposed virtual machine framework supports lookup and binding of symbols in a flat global namespace as well as in namespaces which are defined by other resolved entities.

4.4.3 Required Loading and Linking Functions

Two important data structures are defined by the loading/linking component: `Descriptor` and `ResolvedItem`. Both of them need need to be given a concrete implementation by the VM implementor, the framework does not stipulate how these data structures must look

[10]However, this can also have quite undesirable effects: with the UNIX command line

```
java -verbose:class classname 2>&1 1>/dev/null | wc -l
```

it is possible to determine, how many classes are loaded during the execution of a Java program. Alarmingly, this yields a class count of 168 for a simple "Hello World" program under JDK 1.2.2-L for Linux!

[11]Actually, this is not the complete truth as class names need only be unique within the namespace of a class loader and different class loaders can define different classes sharing the same name (see, eg, [LB98]).

[12]Even if hashing is used, a large shared namespace has a higher probability for hash collisions than a small one.

like.
`Descriptor` is a symbolic reference to some kind of runtime entity, eg, Java method signatures or VP tool names are such symbolic references. A simple implementation of the `Descriptor` interface could indeed be a string variable.
`ResolvedItem` is a runtime data structure in its final resolved form. Java method blocks / `method_info` structures or Dis `Linkage_items` are good examples. All runtime data structures – no matter whether they contain information about classes, methods, procedures, fields or constants – are treated as `ResolvedItems`. A concrete implementation of a VM has to provide its own fields and accessor methods for operating on these items.

In its simplest form, a resolved item can be found by looking up its descriptor with the `lookup` function. As the framework also supports hierarchical namespaces, it is also possible to lookup a descriptor in the namespace which is defined by some other resolved runtime item (`lookupIn`). Of course, there are also `bind` and `bindIn` functions, which allow to associate a runtime data structure with a certain descriptor. A descriptor is considered unresolved when it is known to the linking system, but no `ResolvedItem` is associated with it; `linkModule` with the `lazy` flag set can define unresolved descriptors, or they can be intentionally created by using `bind` with a null reference/null pointer.

Within the VM framework the units which are loaded into the system are called "modules". A module can be a VP tool, a Java class or a Dis module, for instance. Modules are simply referred to by their module name.
The function `linkModule` loads and links a new module into the system, with `initModule` additional initialization can be performed (eg, Java static blocks). `unlinkModule` performs a "hard unlink" [HO91], that is it removes a module completely from memory, possibly leaving unresolved or dangling references behind. This should only be done, when it is really safe to do so, which can be determined with the function `isInUse`. When it is not possible to find out to which module a certain descriptor belongs, this can be found out with the `getDefiningModule` function.
The remaining functions of the interface allow inquiries about loaded modules, and resolved or unresolved descriptors.

4.4.4 IDL Specification

```
// linkloader.idl
// Mirko Raner, 24.08.2000
//
module gvmf
{
    module linkloader_
    {
        // Interface for resolved runtime data structures
        interface ResolvedItem {};

        // Interface for symbolic item names
        interface Descriptor {};
```

```
// Necessary for IDL compiler, see CORBA standard, section 3.12
typedef sequence<Descriptor> DescriptorArray;

// Main interface
interface LinkLoader: gvmf::VirtualMachineComponent
{
    // Create a descriptor from a string
    Descriptor createDescriptor(in string desc);

    // Lookup a descriptor in the global namespace
    ResolvedItem lookup(inout Descriptor desc);

    // Lookup a descriptor in the namespace of an already resolved item
    ResolvedItem lookupIn(inout Descriptor desc,
        inout ResolvedItem item);

    // Associate a descriptor with a runtime data structure
    void bind(inout Descriptor desc, inout ResolvedItem item);

    // Associate a scoped descriptor with a runtime data structure
    void bindIn(inout ResolvedItem parent, inout Descriptor desc,
        inout ResolvedItem item);

    // Find the defining module for a certain descriptor
    string getDefiningModule(inout Descriptor desc);

    // Link a new module into the system (lazy or eager)
    void linkModule(in string modulename, in boolean lazy);

    // Initialize an already loaded module
    void initModule(in string modulename);

    // Unlink a module (hard unlink!)
    void unlinkModule(in string modulename);

    // Find out whether a module is still in use or can be unlinked
    boolean isInUse(in string modulename);

    // Get a list of all global resolved descriptors
    DescriptorArray getResolvedDescriptors();

    // Get a list of all global unresolved descriptors
    DescriptorArray getUnresolvedDescriptors();

    // Get a list of all scoped resolved descriptors
    DescriptorArray getResolvedDescriptorsIn(inout ResolvedItem item);
```

```
            // Get a list of all scoped unresolved descriptors
            DescriptorArray getUnresolvedDescriptorsIn
                (inout ResolvedItem item);

            // Get the names of all loaded modules
            gvmf::StringArray getLoadedModules();
        };
    };
};
```

4.5 Thread System

Support for concurrent program execution and synchronized access to shared data structures is an essential demand to every modern programming language. In virtual machines this demand can be addressed in different ways. The classic P-Machine is the only VM that does not consider the multi-threaded execution of programs. The Virtual Processor (VP) and the VP-based Elate OS claim to be multi-threaded, but in the instruction set no MT-related instructions appear.[13]

The JVM [LY99], Dis VM [VNH00] and Smalltalk VM [GR83] approach the problem of multi-threading from different sides and to a different extent.

4.5.1 Thread-related Functions in Virtual Machines

On instruction set level the Java Virtual Machine only supports synchronization (via monitors). The JVM instruction set provides the instructions `monitorenter` and `monitorexit`. In the class file format there is also an `ACC_SYNCHRONIZED` flag which synchronizes an entire method on its `this` object. Besides that, multi-threading facilities are provided by the standard classes `java/lang/Thread` (starting, interrupting and joining a thread; stopping and suspend/resume are deprecated and will no longer be supported in the future) and `java/lang/Object` (classic condition variable (CV): wait and notify).

The Smalltalk VM only supports multi-processing (ie, processes, not threads) and provides special primitives for suspending and resuming a process and for semaphore operations [GR83, §29].

In the Dis VM [VNH00] multi-threading and synchronization is handled somewhat differently. There are instructions for starting (`spawn`, `mspawn`) and terminating (`exit`) a thread but synchronization is exclusively addressed by so-called "channels" (see [Ker00] for a programming example).

[13]For non-customers of Tao Systems there is only limited information on VP and Elate available. Tao Systems provides the VP Reference Manual to interested parties on a non-disclosure basis, however this manual only provides an instruction reference from an assembly programmer's point of view and does not reveal too much information about the inner workings and implementation of VP.

A channel is a means of communication between threads, similar to a named pipe. The Dis VM provides instructions for creating new channels which can transmit a certain type of data (eg, `newcf` or `newcw`). Additional functions such as `alt` (alternate between commmunications and `nbalt` (non-blocking `alt`) allow to realize something like Java's `wait()` / `notify` / `notifyAll`. The classic synchronization mechanisms (monitor, semaphore, CV, etc) can be implemented with channels, though not always efficiently (there is a considerable overhead involved, when a simple monitor lock is implemented with a heavyweight data structure such as a channel).

Table 9 shows an overview of thread-related machine instructions.

JVM	monitorenter	Get monitor lock
	monitorexit	Release monitor lock
Dis VM	spawn	Spawn new thread
	mspawn	Spawn new thread in another module
	exit	Exit thread
	newc...	Create new channel
	send	Send to a channel
	recv	Receive from a channel
	alt	Select a channel from a list of ready channels
	nbalt	Non-blocking `alt`
Smalltalk VM	Process suspend	Suspend a process
	Process Resume	Resume a process
	Semaphore wait	Enter a semaphore
	Semaphore signal	Leave a semaphore

Table 9: Thread-related machine instructions in selected virtual machines.

4.5.2 Synchronization in Virtual Machines and Operating Systems

It is not easy to define an all-embracing universal interface for the various existing synchronization mechanisms [LB96]. The JVM, Dis VM and Smalltalk VM support synchronization in different ways, whilst the P-Machine does not have support for multithreading at all. The the documentation of the synchronization mechanisms for the Virtual Processor is not publicly available.

Besides the synchronization mechanisms used in the JVM, Dis VM and Smalltalk VM there are other common mechanisms which appear in the POSIX and Solaris threads API [LB96, §5]. These mechanisms are also taken into account here, because future VMs might require these mechanisms. Table 10 shows these synchronization mechanisms and the systems (OS/VM) which use them.

Mutex	JVM, POSIX, Solaris
Semaphore	Smalltalk VM, POSIX, Solaris
Condition Variable	JVM, POSIX, Solaris
Reader/Writer Lock	Solaris
Channels	Dis VM

Table 10: Synchronization mechanisms and their use in VMs and operating systems.

Most of these synchronization mechanisms can be simulated by the programmer with other mechanisms (see [LB96, §5] or [Wie97]), but with a loss of efficiency. For sake of compactness, it is not a good idea to provide special functions for all mechanisms in the framework. If a subset of mechanisms is selected, certain VMs could be straightforwardly implemented. As a solution to this dilemma, the framework provides a generic `Synchronizer` interface that serves as an abstract synchronization mechanism. For example, when a component wants to use a monitor it first creates an appropriate synchronizer, by calling `createSynchronizer(MONITOR)`, which it then enters with `performOperation(ENTER)`. `MONITOR` and `ENTER` are string constants.
There is still the problem, what happens if a VM component requests a synchronizer which is not available. However, the solution is better than having 5 or 6 additional interfaces plus a large number of functions – which as well might not be implemented in every case.

4.5.3 Required Thread Interface Functions

The framework provides functions for creating, spawning, exitting, suspending, resuming and joining a thread.
In addition to that it is possible, to inquire whether the thread is currently suspended at a GC point (garbage collection point). Garbage collection points [Age98, WG98, SLC99] are those points in a thread where complete information about the stack content is available; usually only certain instructions, like method calls and backward branches are GC points. The thread interface also allows to suspend all threads (except for the calling thread) at their next GC point. Thus, an exact garbage collection can safely be performed. The caller stack of a thread is usually also the root for any mark & sweep garbage collection. Therefore every thread must be able to return its current top stack frame (however, this function should only be called for suspended threads).

The problem of additional thread attributes, such as daemon status, scheduling parameters and attached/detached state are handled by generic `ThreadAttributes`. This method is also used in the popular POSIX API [LB96].

A complete thread interface description is listed in section 4.5.4.

4.5.4 IDL Specification

```
// thread.idl
// Mirko Raner, 24.08.2000
//
module gvmf
{
    module thread_
    {
        // Generic interface for synchronization mechanisms
        interface Synchronizer
        {
            // Perform a simple operation (eg, enter, exit, signal, wait)
            void performOperation(in string operation);

            // Perform an operation which needs an additional argument
            // (eg, write data to a channel)
            void performOperationWith(in string operation,
                inout gvmf::memory_::Memory syncarg);

            // Destroy the synchronizer
            void destroy();
        };

        // POSIX-style attribute interface (string-based)
        interface ThreadAttributes
        {
            // Set attribute 'attr' to value 'value'
            void setAttribute(in string attr, in string value);

            // Inquire the value of an attribute
            string getAttribute(in string attr);
               };

        // Thread interface
        interface Thread
        {
            // Start the thread
            void spawn();

            // Finish the thread
            void exit();

            // Suspend the thread
            void suspend();

            // Resume a suspended thread
            void resume();

            // Join another thread
            void join();

            // Sleep
            void sleep(in long milliseconds);

            // Determines whether the thread is suspended at a point where
            // all GC-related type information is available
            boolean isAtGCPoint();

            // Get the top stack frame
            gvmf::processor_::StackFrame getStackFrame();

            // Get the attributes of the thread
            ThreadAttributes getAttributes();
        };

        typedef sequence<Thread> ThreadArray;

        // Main interface
        interface ThreadSystem: gvmf::VirtualMachineComponent
        {
            // Create a new thread with the given attributes
            Thread createThread(in ThreadAttributes attributes);

            // Create a synchronizer of the requested type
            Synchronizer createSynchronizer(in string type);

            // Suspend all threads except for caller thread
            void suspendAll();

            // Suspend all threads except for caller thread at next GC point
            void suspendAllAtNextGCP();

            // Get an array of all threads (no matter in which state)
            ThreadArray getThreads();

            // Get a reference to the currently executing thread
            Thread getCurrentThread();
        };
    };
};
```

4.6 Native Support

There is always a boundary line where the virtual machine ends and the underlying operating system begins. Basically, every VM component has such a boundary line to the OS:

the memory manager accesses OS functions such as `malloc` or `mmap`, the thread system might eventually call `thr_create` or `thr_exit`, and so on.
But certain functions cannot be simply handled by calls to other VM components because they either access OS-specific functionality or simply do not fit or belong into one of the VM components. Examples for these situations are I/O functions or graphical user interfaces which are needed to implement standardized system libraries.

4.6.1 Native Support in Existing Virtual Machines

Two different approaches can be taken to address the problem of accessing functionality outside the VM which cannot be accessed by any of the other components:

- A "native interface" which allows interaction with code written in a different language (usually C, though)
- Primitives that provide the required functionality and are hard-wired into the VM

Clearly the first approach is the more flexible one, but different philosophies lie beneath the various VMs:[14]

- Java Virtual Machine [Lia99, Gor98]:
  The JVM provides a native interface to C and C++. Java NMI (Native Method Interface) was the original interface that was used in JDK 1.0.2. It had a number of serious problems and third parties used their own proprietary native interfaces instead (eg, Microsoft's RNI or Netscape's JRI). JNI was a joint effort between Sun an other companies, with the objective to define a platform-independent native interface. Still there are proprietary alternatives to JNI, such as the Cygnus Native Interface CNI [RHI00].
- Virtual Processor [Tao00]:
  VP is intended as a virtual machine for heterogeneous multiprocessing. It is possible to create pre-translated "tools" which can be called by other tools, independent of their current target platform.
- Smalltalk Virtual Machine [GR83]:
  The Smalltalk VM uses primitive methods for the following functional areas: integer and float arithmetics, object and string access, process and semaphore control, I/O functions, array handling, mouse polling (!) and a number of other functions.
- P-Machine [NAJ$^+$81]:
  Like the Smalltalk VM, the P-Machine also uses primitives, called "standard procedures". These standard procedures include: arithmetic functions (`SIN`, `COS`, `ATN`, `SQT`, etc), object allocation and stack manipulation (`NEW`, `SAV`) and file I/O (eg, `ELN`, `GET`, `PUT` or `RDC`).

[14]Unfortunately, a description of the native interface of the Dis VM was not available at the time of this writing.

4.6.2 Required Native Interface Functions

In fact, most of the low-level OS functionality can already be accessed through other components of the virtual machine:

- Loading and managing of code is handled by the linkloader.
- Memory and references/pointers are accessed through the memory manager.
- Threading and synchronization is provided by the thread system.
- Reflection is handled by the reflective system.

A native interface must allow external methods to call methods inside the VM and vice versa. A closer look at a native interface like the JNI reveals that most methods are actually calls into the VM. Transfer of control from the VM to the native environment is simply done by the `native` modifier and a C program which adheres to the JNI calling conventions.

For native programming, the framework assumes that the native functions are written in the same programming language as the framework and that native methods can access all framework methods via the usual mechanisms. The calling conventions are also predefined by the programming language and the tools which were used to generate the framework skeleton from the IDL descriptions.
Special care must be taken for memory which is used on both sides of the native interface, so that no memory block becomes a "memory leak" or is accidentally reclaimed.

Just like the "Call Standard Procedure" (CSP) instruction of the P-Machine, the framework provides only a single call which transfers control to the outside of the VM framework.

4.6.3 IDL Specification

```
// native.idl
// Mirko Raner, 16.09.2000
//
module gvmf
{
    module native_
    {
        // Native interface
        interface NativeSupport
        {
            gvmf::memory_::Memory callNative(in string name,
                inout gvmf::memory_::MemoryArray arguments);
        };
    };
};
```

4.7 Security Manager

Two important services are provided by the security manager [MD97, chapter 1]:

- Controlling access to system resources
- Preventing corrupted or malicious code from getting loaded into the virtual machine

Especially in modern distributed systems additional security-related tasks are also conceivable. For instance, there is a need for secure, interception-proof communications between particular machines within a distributed system. However, these are more high-level services and do not directly affect the integrity and consistent operation of the virtual machine itself.
Also not taken into account are low-level programming safety like null reference checks, array boundary checks and type safety because these features are actually in the responsibility of the execution engine.

4.7.1 Security in Existing Virtual Machines

The demand for a sophisticated security system in VMs has arisen quite recently and was not least driven by distributed, Internet-based and multi-user-enabled applications. This is also the reason, why earlier VMs – namely the P-Machine, Smalltalk and the Virtual Processor – do not include a proper security system. Again, the Java VM is the main source for design inspiration.

An important contribution to security improvement has been made by the new security features in the Java 2 SDK. From version 1.2 onwards there is not only global permission checking (like performed by the `check...` methods of class `java.lang.SecurityManager`) but also caller-specific permission checking which traces back the complete call path which lead to the current point of execution [Ven99, chapter 3]. The `AccessController` examines the security context (or `ProtectionDomain` in Java jargon) of all preceding stack frames and disallows access if any of the calling methods should not have the necessary permissions.

In the Dis VM security is implemented as follows [VNH00, Sha99]:

- All resources are accessed as files through the Styx protocol, access is controlled by file permissions similar to those in the UNIX file system.
- Bytecode integrity is ensured by cryptographic signatures that are included in the module files.

Cryptographic code verification does not really guarantee that only valid code makes its way into the VM. It is technically no problem to affix a valid signature to a malevolent module file. The user has to decide whether or not to accept the signing authority, whereas a real runtime verifier always guarantees a certain minimum amount of security, even without signatures or certificates. If the user unknowingly grants bad code access to his system he has to expect the worst.

4.7.2 Required Security Interface Functions

In order to keep the number of interface functions small (and also to avoid design catastrophes like the JDK 1.0/1.1 security manager) the security component defines a generic `Permission` interface which encapsulates permissions to access a certain system resource. This interface is used by all `check`... functions. Some of these functions take another optional string argument. For instance, access to a certain file can be checked with a general file permission object and the filename as additional string parameter.
The security manager uses the `getSecurityContext` functions of the processor interface in order to recursively check permissions. Thus, a functionality similar to the Java 1.2 `AccessController` can be implemented. The `isPrivileged` function corresponds to a `doPrivileged` in Java: the stack search is cancelled when a privileged context is encountered [Ven99].

4.7.3 IDL Specification

```
\\ security.idl
\\ Mirko Raner, 11.09.2000
\\
module gvmf
{
    module security_interface
    {
        // General perrmission interface
        interface Permission
        {
            // Determine whether a certain permission already implies another
            // permission (eg, "fileaccess *" implies "fileaccess myfile".
            boolean implies(in Permission permission);
        };

        // Security context interface
        interface SecurityContext
        {
            // Determine whether a privileged security context was encountered
            boolean isPrivileged();
        };

        // Main interface
        interface SecurityManager: VirtualMachineComponent
        {
            // Get the current security context from the current stack frame
            SecurityContext getCurrentSecurityContext();

            // Create a new permission
            Permission createPermission(in string);

            // Check whether the caller has a certain permission
            boolean checkPermission(in Permission permission);

            // Check whether the caller has a certain permission
            boolean checkPermissionWithArguments(in Permission permission,
                                                 in string arguments);

            // Check whether the caller has a certain permission
            // (with stack check)
            boolean checkPermissionRecursively(in Permission permission);

            // Check whether the caller has a certain permission
            // (with stack check)
            boolean checkPermissionRecursivelyWithArguments(in Permission perm,
                                                            in string args);

            // Check the integrity of incoming bytecode
            boolean checkBytecode(inout gvmf::memory_::Memory bytecode);
        };
    };
};
```

4.8 Reflective System

One may argue that the reflective system does not fit in between the other components (execution engine, thread system, memory manager, etc) because it operates on a meta-level and should therefore be separated from the other components. But from an implementor's point of view it is a regular VM component that requires an implementation like all other components and therefore it should be placed in its own exchangeable component.

4.8.1 Reflection in Existing Virtual Machines

Reflection in a virtual machine means that the VM can obtain information about itself and can change its state according to certain conditions (see [KG96, Gol97] for definitions). Actually reflection is a quite natural thing which can be implemented in a very straightforward way, if it is considered for the design from the beginning on (eg, Smalltalk-80 has such a reflective system [GR83] where a meta-class is treated like a regular class and it is a straightforward operation to obtain a meta-class from a class). However, many virtual machines have not really been planned as reflective systems and the need for reflection (or more powerful reflection) was found after the original design was finished. This leads to the need for "reification" [Gol97, chapter I]: internal workings and data structures of the virtual machine must be made visible and accessible to the programmer (in a reflective design these structures should already have been visible).

The framework which is proposed here, will only provide simple reflective features such as offered by the `java/lang/reflect` package of JDK 1.1. That is, an object can return its class object, and this class object is able to produce other objects which describe its methods, fields, etc. Therefore, the key problem is to produce an object from a runtime data structure and vice versa – and this is exactly what the only two functions of the reflection interface do.

This approach can definitely not compete with the advanced features of JDK 1.2/1.3 or MetaXa [GK98] but provides a foundation for simple meta-programming. Possibly, future versions of the framework might include extended reflection support.

4.8.2 IDL Specification

```
// reflective.idl
// Mirko Raner, 22.09.2000
//
module gvmf
{
    module reflective_
    {
        // Main reflection interface
        interface ReflectiveSystem
        {
            // Create an object from a runtime data structure
                  gvmf::memory_::Memory getMetaItem
                (inout gvmf::linkloader_::ResolvedItem item);

            // Create a runtime data structure from an object
            gvmf::linkloader_::ResolvedItem getBaseItem
                (inout gvmf::memory_::Memory memory);
        };
    };
};
```

5 Proof-of-Concept Implementation

The objective of the proof-of-concept implementation is the assembly of a simple Java Virtual Machine from individual components. In order to concentrate on the component interfaces and the integration of the components, the implementation re-uses already predefined sources which will then be wrapped in order to fulfill the interfaces of the framework (see figure 5).

Standard C was chosen as the implementation language.

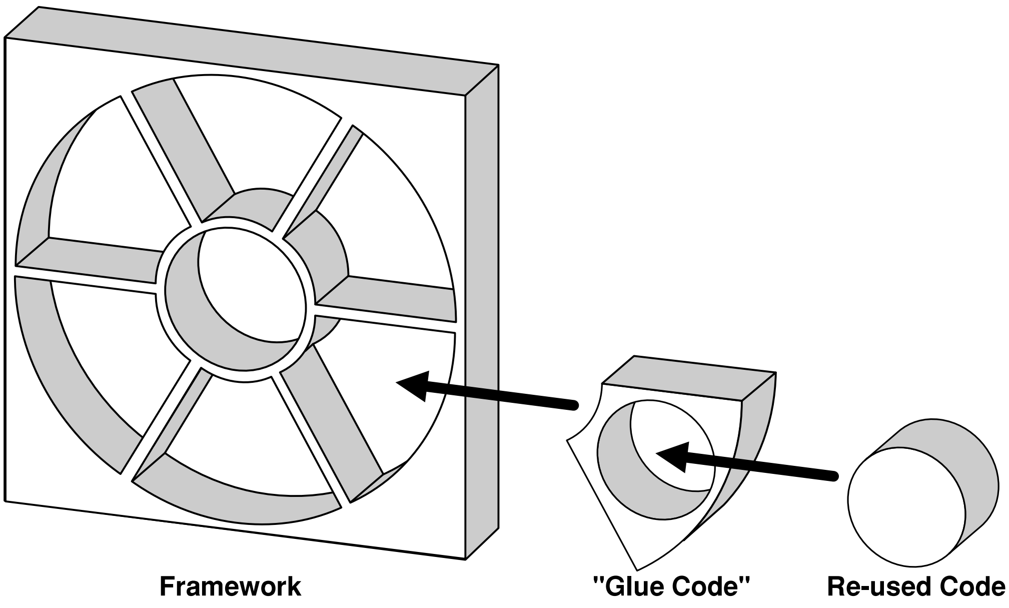

Figure 5: Wrapping existing code for use with the framework.

5.1 Implementation Process

In general, all methods of the framework should be implemented in a reasonable way and should not generate any errors upon invocation. Empty implementations which simply do nothing are absolutely permissible and will even be quite common, because certain methods are only required by certain implementation styles. Thus, both situations may arise: (1) a component calls a method of another component which is not implemented, and (2) a component provides implementations for methods that are never called by other components.

The first step of the implementation is the generation of the framework skeleton (see 6). This can be done by an IDL-to-C compiler such as `orbit-idl` or the ILU `c-stubber` [ILU00]. Unfortunately, there are not many other IDL compilers which are capable of producing C header files, and both compilers had certain problems.

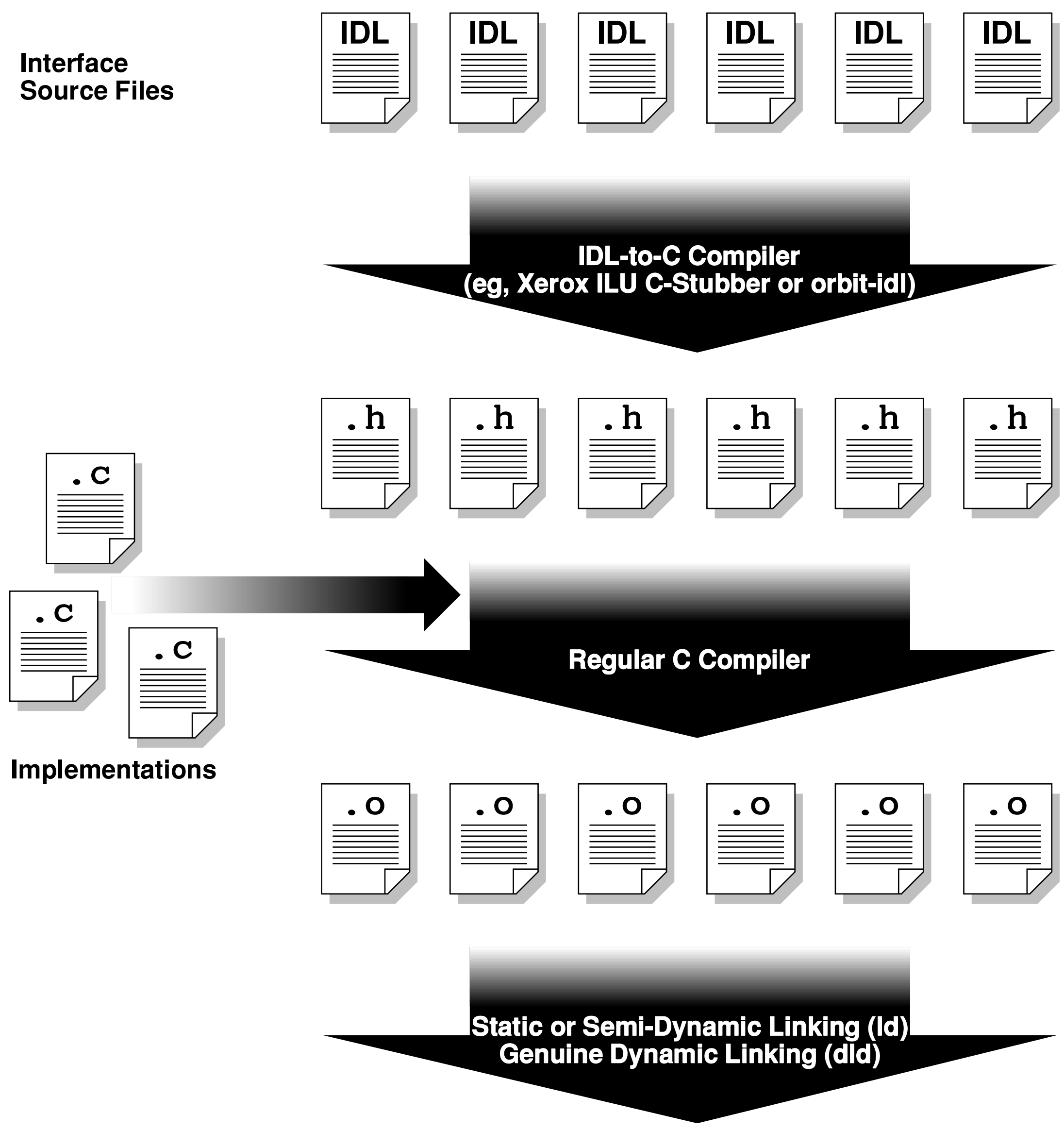
Interface Source Files
IDL
IDL-to-C Compiler (eg, Xerox ILU C-Stubber or orbit-idl)
.h
.c
Implementations
Regular C Compiler
.o
Static or Semi-Dynamic Linking (ld)
Genuine Dynamic Linking (dld)

Figure 6: Generation of the framework skeleton.

After all IDL files had been merged into one big IDL file, `c-stubber` was able to produce usable code.

5.2 Re-usable Existing Material

There is a some material available that might be used for the component-based implementation of the JVM:

- Cygnus Solutions has developed a Java recompiler (GCJ) based on GCC [Bot97]. This recompiler can serve as the implementation of the execution engine and also predefines other tasks such as reflection and exception handling.

- Dynamic class loading can be performed with GNU dld [Fre96, HO91], which also was used for implementing dynamic class loading in GNU Smalltalk.
- There are several different native interfaces available for the Java VM. The Native Method Interface (NMI) is no longer used today. Ideas from Netscape's Java Runtime Interface (JRI) and Microsoft's Raw Native Interface (RNI) led to the current standard native interface for Java – JNI (Java Native Interface). In addition to that Cygnus Solutions provides an optimized interface (CNI; Cygnus Native Interface) for GCJ.
- The well-known Boehm-Demers-Weiser garbage collector [BW88] is publicly available and relatively well encapsulated. However, it is a conservative collector which assumes an "uncooperative" environment. That is, the Boehm-Demers-Weiser collector has only very little interaction with other components, and is therefore not very interesting.
- Many operating systems today come with a POSIX thread library. The multithreading interface of the framework can be mapped to the standard Pthreads interface with moderate effort.

5.3 Implementation Example: Garbage Collector

The following listing shows a part of a garbage collector implementation which was tested in the framework. The header file `gvmf.h` was generated with the ILU `c-stubber` and some `#defines` were used in order to remove some of the CORBA-specific types from the output. But even if CORBA types are replaced, ILU produces some overhead, eg, by the `ILU_C_ENVIRONMENT`. The code only shows the "mark" phase of the garbage-collection, other code has been omitted.

```
// Remove CORBAness from source files; the VM framework does not use CORBA
#define CORBA_
#define server_

// Replace unknown types
#define boolean int

// Include generated header file
#include "gvmf.h"


boolean gvmf_memory__MemoryManager_initialize
(
    gvmf_memory__MemoryManager _this,
    gvmf_Framework* framework,
    ilu_CString arguments,
    ILU_C_ENVIRONMENT *_status
```

```
)
{
    gvmf_memory__MemoryManager__set_frameworkHandle
        (_this, *framework, _status);
}

void gvmf_memory__MemoryManager_runGC
    (gvmf_memory__MemoryManager _this, ILU_C_ENVIRONMENT *_status)
{
    void mark(gvmf_memory__Memory memory, gvmf_memory__Type type,
        ILU_C_ENVIRONMENT *_status);

    gvmf_Framework framework;
    gvmf_thread__ThreadSystem threadSystem;
    gvmf_thread__ThreadArray* threads;
    gvmf_processor__StackFrame stackFrame;
    int numberOfThreads;
    int index;

    framework = gvmf_memory__MemoryManager__get_frameworkHandle
        (_this, _status);

    threadSystem = gvmf_Framework_getThreadSystem (framework, _status);

    gvmf_thread__ThreadSystem_suspendAllAtNextGCP (threadSystem, _status);
    threads = gvmf_thread__ThreadSystem_getThreads (threadSystem, _status);
    numberOfThreads = gvmf_thread__ThreadArray_Length (threads);

    for (index = 0; index < numberOfThreads; index++)
    {
        gvmf_memory__Memory memory;
        gvmf_memory__Type type;
        stackFrame = gvmf_thread__Thread_getStackFrame
            (*gvmf_thread__ThreadArray_Nth(threads, index), _status);
        memory = gvmf_processor__StackFrame_getMemory (stackFrame, _status);
        type = gvmf_processor__StackFrame_getType (stackFrame, _status);
        mark (memory, type, _status);
    }

    // ...
}

void mark(gvmf_memory__Memory memory, gvmf_memory__Type type,
    ILU_C_ENVIRONMENT *_status)
{
    int numberOfPointers, index;
    gvmf_memory__Memory pointer;
    gvmf_memory__Type pointerType;

    if (gvmf_memory__Memory_isMarked (memory, _status))
    {
        return;
    }

    gvmf_memory__Memory_mark (memory, _status);
    numberOfPointers = gvmf_memory__Type_getNumberOfPointers (type, _status);
    for (index = 0; index < numberOfPointers; index++)
    {
        pointer = gvmf_memory__Type_getPointerValue
            (type, &memory, index, _status);
        pointerType = gvmf_memory__Memory_getType (pointer, _status);
        mark (pointer, pointerType, _status);
    }
}
```

6 Outlook and Conclusion

There is still a long way to go until modular virtual machines will be state-of-the-art. This thesis made an attempt to establish a framework which splits the task of building a virtual machine into smaller parts which can be implemented independently. How adequate this framework really is must still be examined in the future. In the given timeframe, only a brief "sanity check" could be conducted, more detailed tests will have to follow.
The design has just passed its first iteration and there are probably things which need to be fixed. Certain features, such as support for weak pointers, are not yet take into consideration for the current version of the framework.

Besides the definition of the framework also a lot of interesting insights about virtual machines and their implementation were obtained.

One of the astonishing insights was the absolute market dominance of the Java virtual machine. It is by far (!) the best documented virtual machine ever. While it is a real challenge to find documentation about the Virtual Processor, the Dis VM or the Smalltalk VM, there is an incredible plethora of data about the JVM and its numerous implementations.
The framework was intended to be independent of a particular virtual machine, but often it was very hard to maintain this general view, because the JVM was simply the only VM that really implemented a certain feature and provided the necessary technical documentation of this feature.

Not only the specifications for the individual components of which a VM consists are often insufficient, even the VM as a whole is often not clearly specified.

Especially the demarcation of the virtual machine against its surrounding environment is often neglected. The question is: where does the VM end and where does the surrounding environment begin? There is no easy answer to this, and in many cases it is not even possible to separate the two. The Smalltalk VM is part of an almost completely self-contained Smalltalk environment and it is very hard to separate the VM from that environment. In a similar way, the Virtual Processor is wedded with the Elate OS and the Dis VM is almost inseparably associated with Inferno. For the Java VM things do not look much different: part of the standard class library is referenced by the VM specification, which has the result that at least certain parts of the class library became an essential component of the JVM.

Another problem area was the actual description of the framework. IDL as description language was chosen in want for a better solution. IDL is so closely associated with the CORBA client/server, that it hardly can be considered a technology-independent interface description language. Also, there are only very few IDL compilers which can generate C code; most compilers only support Java and C++, which are the main CORBA implementation languages. All these IDL compilers generate CORBA-biased code, which can be interesting, if VM components should be distributed across several machines. However, for an efficient implementation on a single machine their output is not a suitable basis.
XMI might be a format which is appropriate for the purpose, but right now there are no

tools available which could generate suitable C skeletons from an XMI file. In addition to that, XMI files are very bulky and do not allow a compact representation of a framework. A possible solution might be a proprietary XML-based description format, as used by other component systems, such as Enterprise Java Beans.

The proof-of-concept implementation could also not be completely finished. Building a VM is a multi-man-month task which is usually performed by large teams of software developers. Even if pre-existing components are re-used it cannot be expected that the implementation is complete. In addition to that it turned out that most of the prospective re-usable units turned out to be not as re-usable and self-contained as expected.

Due to the large structural differences between particular virtual machines it is very hard to propose a framework that is completely appropriate for all different VMs. Real interoperability is only possible to a limited degree due to the different VM structure.

A good approach for future research is probably to concentrate indeed on the Java VM: it has the most complete specification and documentation, and there is hardly any feature that the JVM does not support. When a stable and sane framework for the JVM can be specified, it might be easier to modify that framework so that it is also applicable for non-Java VMs.

A Glossary of Acronyms

ADT	Abstract Data Type
API	Application Programming Interface
ARM	Advanced RISC Machine
CNI	Cygnus Native Interface
CORBA	Common Object Request Broker Architecture
CV	Condition Variable
DLD	Dynamic Loader
DVM	Distributed Virtual Machine
EUMEL	Extendable Multi-User Microprocessor Elan System
EVM	Exact Virtual Machine
FP	Frame Pointer (Dis VM)
GCC	GNU C Compiler
GCJ	GNU Compiler for Java
GMD	Gesellschaft für Mathematik und Datenverarbeitung
GNU	GNU's Not Unix
GVMF	Generic Virtual Machine Framework
HPI	$\rightarrow$ JHPI
IDL	Interface Definition Language
I/O	Input / Output
JDK	Java Development Kit
JHPI	Java Host Programming Interface
JIT	Just In Time
JNI	Java Native Interface
JRI	Java Runtime Interface
JTE	Java Technology Edition
MM	Memory Machine
MP	Module Pointer (Dis VM)
MPC	Media P-Code
MT	Multi-threading (or *multi-threaded*)
OMG	Object Management Group
OO	Object Orientation (or *object oriented*)
OS	Operating System
OVAL	Object Visualization and Annotation Language
PCR	Portable Common Runtime
POSIX	Portable Operating System Interface
RC	Reference Counting
RM	Register Machine
RNI	Raw Native Interface
SDK	Software Development Kit

SM	Stack Machine
SP	Stack Pointer
TAOS	Tao Operating System
UCSD	University of California at San Diego
UTF	Unicode Transformation Format
VCGC	Very Concurrent Garbage Collector
VM	Virtual Machine
VVM	Virtual Virtual Machine
XMI	Extensible Metadata Interchange
XML	Extensible Markup Language
XVM	Extensible Virtual Machine

References

[AAB+00] Bowen ALPERN / C. R. ATTANASIO / John J. BARTON / Michael G. BURKE / Perry CHENG / Jong-Deok CHOI / Anthony COCCHI / Stephen J. FINK / David GROVE / Michael HIND / Susan FLYNN-HUMMEL / Derek LIEBER / Vassily LITVINOV / Mark F. MERGEN / Ton NGO / James R. RUSSELL / Vivek SARKAR / Mauricio J. SERRANO / Janice C. SHEPHERD / Stephen E. SMITH / V. C. SREEDHAR / Harini SRINIVASAN / John WHALEY. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211 – 238, 2000.

[ADM98] Ole AGESEN / David DETLEFS / J. Eliot B. MOSS. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 269 – 279, Montreal, Canada, 1998.

[Age98] Ole AGESEN. GC Points in a Threaded Environment. Technical Report SMLI TR-98-70, Sun Microsystems Laboratories, 1998.

[AGH00] Ken ARNOLD / James GOSLING / David HOLMES. *The Java Programming Language.* Addison-Wesley, Reading (MA), 3rd edition, 2000.

[Bar81] David William BARRON, editor. *Pascal – The Language and its Implementation.* John Wiley & Sons, Chichester, UK, 1981.

[Bot97] Per BOTHNER. A GCC-based Java Implementation. In *Proceedings of the IEEE Compcon '97 Conference*, San Jose (CA), 1997.

[Bow78] Kenneth BOWLES. UCSD Pascal: A (Nearly) Machine Independent Software System. *BYTE Magazine*, May 1978.

[BW88] Hans-J. BOEHM / Mark WEISER. Garbage Collection in an Uncooperative Environment. *Software Practice & Experience*, pages 807 – 820, September 1988.

[Cha89] Paul CHASE. *VM/CMS – A User's Guide.* John Wiley & Sons, New York (NY), 1989.

[CY78] Kin-Man CHUNG / Herbert YUEN. A "Tiny" Pascal Compiler – Part 3: P-Code to 8080 Conversion. *BYTE Magzine*, November 1978.

[Dal97] Matthias Kalle DALHEIMER. *Java Virtual Machine.* O'Reilly Deutschland, Köln, Germany, 1997.

[DPP+97] Sean DORWARD / Rob PIKE / David PRESOTTO / Dennis RITCHIE / Howard TRICKEY / Phil WINTERBOTTOM. Inferno. In *Proceedings of the IEEE Compcon '97 Conference*, pages 241 – 244, San Jose (CA), 1997.

[DSS90] Sean DORWARD / Ravi SETHI / Jonathan SHOPIRO. Adding New Code to a Running C++ Program. In *Proceedings of the 2nd USENIX C++ Conference*, San Francisco (CA), 1990.

[Eng88] Hermann ENGESSER, editor. *Duden Informatik*. BI-Wissenschaftsverlag, Mannheim, Germany, 1988.

[FPR97] Bertil FOLLIOT / Ian PIUMARTA / Fabio RICCARDI. Virtual Virtual Machines. Presented at the 4th Cabernet Radicals Workshop. Réthymnon (Crete), Greece, 1997.

[FPR98] Bertil FOLLIOT / Ian PIUMARTA / Fabio RICCARDI. A Dynamically Configurable, Multi-Language Execution Platform. Presented at the 8th ACM SIGOPS European Workshop. Sintra, Portugal, 1998.

[Fre96] FREE SOFTWARE FOUNDATION, Cambridge (MA). *GNU Dld Manual*, 1996. Manual edition 1.3 for GNU Dld 3.3.

[GK98] Michael GOLM / Jürgen KLEINÖDER. metaXa and the Future of Reflection. In *Proceedings of the OOPSLA Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, 1998.

[Gol97] Michael GOLM. Design and Implementation of a Meta Architecture for Java. Diploma thesis #DA-I4-02-97, University of Erlangen/Nuremberg, IMMD IV, 1997.

[Gor98] Rob GORDON. *Essential JNI*. Prentice Hall, Upper Saddle River (NJ), 1998.

[GR83] Adele GOLDBERG / D. ROBSON. *Smalltalk-80 – The Language and its Implementation*. Addison-Wesley, Reading (MA), 1983.

[Har98] Timothy HARRIS. Controlling Run-Time Compilation. In *Proceedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications (PLRTIA'98)*, pages 75 – 84, Madrid, Spain, 1998.

[Har99] Timothy HARRIS. An Extensible Virtual Machine Architecture. Presented at the OOPSLA '99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design. Denver (CO), 1999.

[HO91] Wilson HO / Ronald OLSSON. An Approach to Genuine Dynamic Linking. *Software Practice & Experience*, 21(4):375 – 390, April 1991.

[HW98] Lorenz HUELSBERGEN / Phil WINTERBOTTOM. Very Concurrent Mark-&-Sweep Garbage Collection. In *Proceedings of the 1998 International Symposium on Memory Management (ISMM'98)*, pages 166 – 175, Vancouver, Canada, 1998.

[ILU00] XEROX PARC. Inter-Language Unification. URL: `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`, 2000.

[Jag96] Dave JAGGAR. *Advanced RISC Machines Architectural Reference Manual.* Prentice Hall, London, UK, 1996.

[JL96] Richard JONES / Rafael LINS. *Garbage Collection – Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, Chichester, UK, 1996.

[Ker00] Brian W. KERNIGHAN. A Descent into Limbo. URL: `http://www.vitanuova.com/inferno/papers/descent.html`, 2000.

[KG96] Jürgen KLEINÖDER / Michael GOLM. MetaJava: An Efficient Run-Time Meta Architecture for Java^{TM}. In *Proceedings of the International Workshop on Object Orientation in Operating Systems (IWOOOS'96)*, Seattle (WA), 1996.

[KGL$^+$00] Chandra KRINTZ / David GROVE / Derek LIEBER / Vivek SARKAR / Brad CALDER. Reducing the Overhead of Dynamic Compilation. Technical Report TR CSE-2000-0648, University of California at San Diego, 2000.

[LB96] Bil LEWIS / Daniel J. BERG. *Threads Primer – A Guide to Multithreaded Programming.* SunSoft Press / Prentice Hall, Upper Saddle River (NJ), 1996.

[LB98] Sheng LIANG / Gilad BRACHA. Dynamic Class Loading in the Java Virtual Machine. Presented at the OOPSLA '98 Workshop on Virtual Machine Design. Vancouver, Canada, 1998.

[Lia99] Sheng LIANG. *The Java Native Interface: Programmer's Guide and Reference.* Addison-Wesley, Reading (MA), 1999.

[LY97] Tim LINDHOLM / Frank YELLIN. Java Runtime Internals. Presented at the JavaOne'97 Conference, Track 1, Session 27. San Francisco (CA), 1997.

[LY99] Tim LINDHOLM / Frank YELLIN. *The Java Virtual Machine Specification.* Addison-Wesley, Reading (MA), 2nd edition, 1999.

[MD97] Jon MEYER / Troy DOWNING. *Java Virtual Machine.* O'Reilly, Sebastopol (CA), 1997.

[NAJ$^+$81] K. V. NORI / U. AMMANN / K. JENSEN / H. H. NAGELI / Ch. JACOBI. *Pascal-P Implementation Notes*, pages 125 – 170. In Barron [Bar81], 1981.

[OMG99] OBJECT MANAGEMENT GROUP, Needham (MA). *The Common Object Request Broker: Architecture and Specification, Revision 2.3*, June 1999.

[Ran97] Mirko RANER. New Developments for the Implementation of Unicode Word Processors in Java. In *Proceedings of the 11th International Unicode Conference (IUC 11)*, San Jose (CA), 1997.

[Ran98] Mirko RANER. Design and Implementation of a Java Bytecode Recompiler for ARM RISC Processors. Practical thesis #SA-I4-98-10, University of Erlangen/Nuremberg, IMMD IV, 1998.

[Ran99a] Mirko RANER. Implementation der Java Virtual Machine (Teil 1). *Java Magazin*, pages 22 – 29, issue #5.99, 1999.

[Ran99b] Mirko RANER. Implementation der Java Virtual Machine (Teil 2). *Java Magazin*, pages 34 – 40, issue #6.99, 1999.

[Ran00] Mirko RANER. Teaching Object Orientation with the Object Visualization and Annotation Language (OVAL). In *Proceedings of the 5th ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2000)*, pages 45 – 48, Helsinki, Finland, 2000.

[RHI00] RED HAT, INC. The Cygnus Native Interface for C++/Java Integration. URL: `http://sources.redhat.com/java/papers/cni/t1.html`, 2000.

[SGGB99] Emin Gün SIRER / Robert GRIMM / Arthur GREGORY / Brian BERSHAD. Design and Implementation of a Distributed Virtual Machine for Networked Computers. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 202 – 216, Kiawah Island (SC), 1999.

[Sha99] Ravi SHARMA. Distributed Application Development with Inferno. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation (DAC'99)*, New Orleans (LA), 1999.

[SLC99] James STICHNOTH / Guei-Yuan LUEH / Michał CIERNIAK. Support for Garbage Collection at Every Instruction in a Java^{TM} Compiler. In *Proceedings of the 1999 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 118 – 127, Atlanta (GA), 1999.

[Tao00] TAO SYSTEMS LTD, Reading, UK. *VP Reference Manual, Version 1.1*[15], 2000.

[Ven99] Bill VENNERS. *Inside the Java Virtual Machine*. McGraw-Hill, New York (NY), 2nd edition, 1999.

[VNH00] VITA NUOVA HOLDINGS LTD. Inferno User's Guide. URL: `http://www.vitanuova.com/inferno`, 2000.

[WAU99] Mario WOLCZKO / Ole AGESEN / David UNGAR. Towards a Universal Implementation Substrate for Object-Oriented Languages. Presented at the OOPSLA '99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design. Denver (CO), 1999. Sun Microsystems Laboratories document #96-0506.

[WDH89] Mark WEISER / Alan DEMERS / Carl HAUSER. The Portable Common Runtime Approach to Interoperability. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 114 – 122, Litchfield Park (AZ), 1989.

[15]available under NDA from Tao Systems Ltd.

[WG98] Derek WHITE / Alex GARTHWAITE. The GC Interface in the EVM. Technical Report SMLI TR-98-67, Sun Microsystems Laboratories, 1998.

[Wie97] Michael WIEDEKING. Nebenläufige Programmierung mit Java (Thread-Programmierung). In *Tagungsband der 1. Deutschen Java Entwicklerkonferenz (DJEK'97)*, pages 29 – 41, 1997.

[WM92] Reinhard WILHELM / Dieter MAURER. *Übersetzerbau.* Springer Verlag, Heidelberg, Germany, 1992.

[WP97] Phil WINTERBOTTOM / Rob PIKE. The Design of the Inferno Virtual Machine. In *HotChips IX Conference, session 4*, Palo Alto (CA), 1997.

[Yel96] Frank YELLIN. The JIT Compiler API. URL: `http://java.sun.com`, 1996.