ViennaVM: a Virtual Machine for VDM-SL development

Tomohiro Oda[1], Keijiro Araki[2], and Peter Gorm Larsen[3]

[1] Software Research Associates, Inc. (tomohiro@sra.co.jp)
[2] National Institute of Technology, Kumamoto College (araki@kyudai.jp)
[3] Aarhus University, Department of Engineering, (pgl@eng.au.dk)

Abstract. The executable subset of VDM allows code generators to automatically produce program code. A lot of research have been conducted on automated code generators. Virtual machines are common platforms of executing program code. Those virtual machines demand rigorous implementation and in return give portability among different operating systems and CPUs. This paper introduces a virtual machine called ViennaVM which is formally defined in VDM-SL and still under development. The objective of ViennaVM is to serve as a target platform of code generators from VDM specifications.

1 Introduction

Quality of software systems is important in many cases. Model-based development with automated code generation techniques is a promising approach to develop software systems with affordable quality and productivity. Many automated code generators from different VDM dialects [6] have been studied and developed as strong tools to reduce cost of the implementation phase [3,1,7,10]. Those automated code generators emit source code for general programming languages such as C++, C, Java and Smalltalk. However, there are still challenges with applying code generators. The first challenge is the availability of compiler and runtime environments for various target hardware. General programming language systems often provide rich language with build-in functions and libraries. Thus, it is often costly to port the compiler and full set of libraries to brand-new hardware platforms. The second challenge is the portability of the compiler and runtime environment. In some programming languages that provide low-level programming functionality, such as C and C++, does not provide the source level compatibility among different platforms.

This paper proposes and introduces the development of a Virtual Machine (VM) named ViennaVM that is designed as a common target platform for automated code generators from VDM dialects. A VM is an abstracted computer platform targeting efficient execution of code represented in an Intermediate Representation (IR) [9]. IR code is typically designed specific to a particular guest programming language. For example, the Java VM executes Java byte-code of which instruction set efficiently implements the language features of Java, such as primitive operations, boxing/unboxing and method invocations. ViennaVM will have an instruction set suitable to model-based developments with VDM.

One significant advantage of VMs is portability. For example, the VM for Squeak Smalltalk is auto-generated from Squeak Smalltalk itself with the exception of several platform dependent interface to the host operating system. Having a small fraction of hand-written code, the VM of Squeak Smalltalk was ported to various platforms including Windows, Unix and PDAs [2].

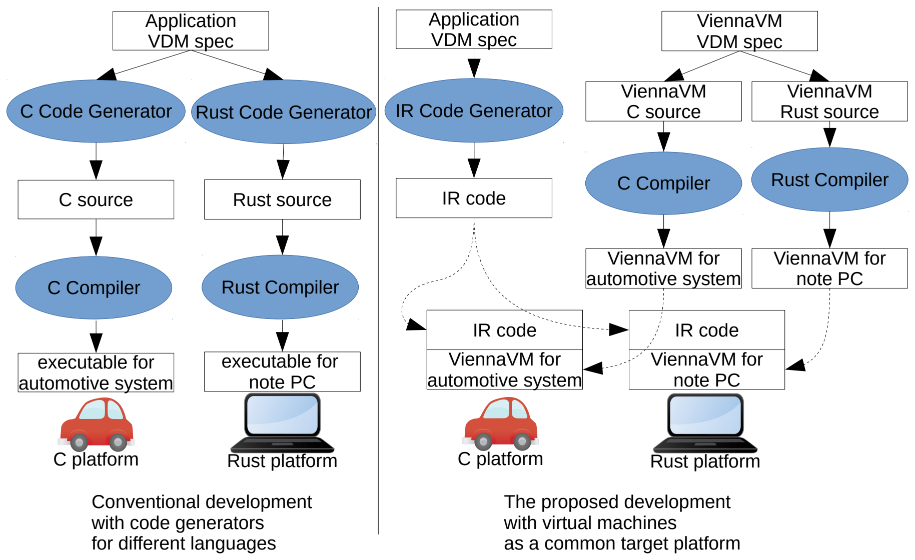

Fig. 1. Code generator-based development and VMs as a common platform

Figure 1 shows how ViennaVM will serve as a common target platform for code generators. Assuming that platform A provides only a C compiler and platform B offers Rust as its standard programming language, two source code, one for C and another for Rust would be generated. Each source would need modification to work with external libraries, such as networks and user interfaces. Using a VM as a common target platform, it is possible to implement one ViennaVM in C and another in Rust to run the same IR code. ViennaVM's IR code is binary compatible among versions of VMs in different programming languages on various target platforms. Each instance of ViennaVM can also be reused in later developments of other applications on the same device.

The rest of this paper starts of with an overview of the objective of ViennaVM in Section 2. Afterwards an overview of the formal specification of ViennaVM using VDM-SL is provided in Section 3. Then the preliminary implementation of a ViennaVM as well as initial benchmarks are provided in Section 4. This is followed by Section 5 about the planned further development. Finally, Section 6 provides a few concluding remarks about the possibilities for this work.

2 Objectives and Non-functional requirements of ViennaVM

The objective of ViennaVM is to provide a common target platform for software development with VDM dialects. To achieve the objective, it is desirable that the VM is *reliable*, *portable*, *productive* and *adaptable* to the host platform. This section explains why these non-functional qualities are required to VMs for smart devices.

In order to make ViennaVM dependable it will itself be developed using VDM-SL. We split the specification process into two phases: an exploratory specification phase that we use ViennaTalk [8] as a development platform, and rigorous specification phase that we use the Overture tool [4] to gain and ensure the quality of the specification. ViennaTalk is a Smalltalk-based IDE with live specification animations for interactive specification authoring in an exploratory manner. ViennaTalk also provides a pretty printer and automated execution of unit tests to enhance agility-related qualities of specification against frequent modifications. ViennaVM will be tested using a unit testing framework on ViennaTalk, called ViennaUnit. ViennaUnit is a simple unit testing framework for VDM-SL that automatically collect test modules whose names end with `Test` and automatically run all test operations whose name start with `test`. After developing a valid specification of ViennaVM, the Overture tool will be used as a development platform. The Overture tool is full-fledged IDE based on the Eclipse platform. The Overture tool's functionality includes combinatory testing and automated generation of proof obligations that enhance rigour qualities of specification as the final product of the specification phase. Combinatory testing will be used to rigorously test a large number of combinations of IR code [5].

ViennaVM needs to be portable. The term *portable* has two sides; one is the portability of ViennaVM, and the second is the portability of IR code. The portability of the both sides is required to software systems distributed among various platforms. The term *productivity* also has two sides; the productivity of ViennaVM and the productivity of the target software on ViennaVM. For the productivity of the VM, the combined use of ViennaTalk and Overture as a tool chain will be a significant factor. For the productivity of the target software, ViennaVM will be ported to Smalltalk so that the target system can be seamlessly developed on ViennaTalk and ViennaVM as a tool chain.

ViennaVM needs to be adaptable to different host platforms. Considering diverse constraints on hardware such as user interfaces and computational resources, ViennaVM needs to be implemented differently according to those constraints, yet satisfying its formal specification. One smart device may be equipped with voice cognition and speech synthesis while another smart device may have small touch screen in a few square centimetres and a physical push button. It is desirable that one program in IR code works on every smart device without having redundant UI code because those devices typically have limited computational resources. Conventional VMs provides low-level interface to UI devices and standard libraries written in the IR code of VMs provide UI frameworks. For example, the Java VM provides the awt and swing frameworks in Java byte-code. Application developers implements UI code for different UI devices and choose either to create different deployment file for each host platform or to provide one deployment file that has all UI code. ViennaVM is planned to provide an abstraction of interactions with the user to make its applications adaptable to various smart devices with different physical user interfaces.

3 Specification of ViennaVM

This section explains the specification of ViennaVM. Although the formal definition is not complete yet, the current snapshot of the formal definition of ViennaVM can execute a simple numeric computation.

3.1 Definition layers

The current snapshot ViennaVM is specified in a moduled form of VDM-SL. Figure 2 shows modules in layers of definition.

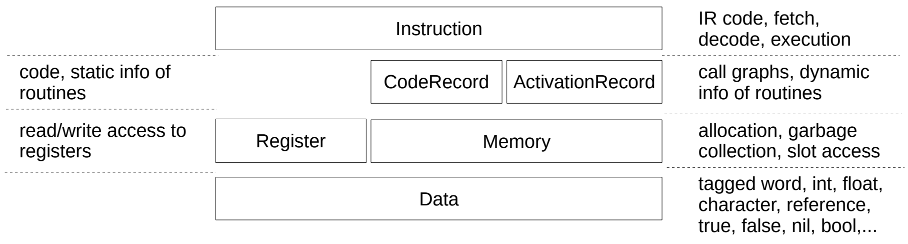
Instruction
IR code, fetch, decode, execution
code, static info of routines
CodeRecord
ActivationRecord
call graphs, dynamic info of routines
read/write access to registers
Register
Memory
allocation, garbage collection, slot access
Data
tagged word, int, float, character, reference, true, false, nil, bool,...

Fig. 2. Layers of the VDM-SL definition of ViennaVM

The bottom layer is the `Data` module that defines the data model of ViennaVM including data type definitions and constant values. Based upon the `Data` module, the `Register` module and the `Memory` module are defined. The `Register` module specifies the internal structure of each register, and the `Memory` module provides a memory model including data layout in a heap object and a garbage collector. ViennaVM is a register machine, which has large number of registers and passes arguments and return values via registers, while Java VM is a stack machine which handles temporary values in a data stack and passes arguments and return values via the data stack. `CodeRecord` and `ActivationRecords` are modules that defines heap objects that represents static and dynamic properties of routines. Then, the `Instruction` module defines IR code instructions and its execution mechanisms such as code fetch and a decoder. The sections below explains each module.

3.2 Data definitions

The `Data` module specifies the data types internally used in ViennaVM and also VDM's basic types. ViennaVM uses 64 bits tagged word as an atomic data entity in IR code. Tagged words are fixed sized data packed with runtime type informations so that the VM can uniformly handle values of different types. A tagged word can either be an integer, a floating point number, a character or a pointer with flags that identify its runtime type. A pointer is not a raw address of a heap object, but it is a reference which

tagged word vector		64 bits unsigned int		b63, ..., b4	b3	b2	b1	b0
		(b0: int flag, b1: non-heap flag, b2: type flag, b3: option flag)						
	int	63bit signed int						1
	pointer to value	56 bit unsigned int		0 0 0 0	0	0	0	0
	float	8 bits dummy	IEEE 754 float32	0 0 0 1	0	0	1	0
	unicode character	8 bits dummy	32 bits unicode	0 0 0 1	1	0	1	0
	nil	56 bits dummy		0 0 1 0	0	0	1	0
	true	56 bits dummy		0 0 1 0	1	0	1	0
	false	56 bits dummy		0 0 1 1	0	0	1	0
	unit type	56 bits dummy		0 0 0 0	0	1	1	0
	bool type	56 bits dummy		0 0 0 0	0	1	1	0
	:	:						
	pointer to type	56 bit unsigned int		0 0 0 0	0	1	0	0
	invalid word			0 0 0 0	1	0	1	0
int		1 bit dummy	63 bit signed int					
float		32bit dummy	IEEE 754 float32					
char		32bit dummy	32bit unsigned int					
pointer		8 bits dummy	56 bits unsigned int					

Fig. 3. Data format of primitive values inside ViennaVM

consists of the heap page index and offset of the heap object. The basic types and values of VDM are also defined in the `Data` module. Figure 3 lists the data definitions.

If the Least Significant Bit (LSB) of a tagged word is 1, the remaining 63 bits represents a signed integer. The second least significant bit of a tagged word is the non-heap flag that indicates whether the tagged word carries a pointer or not. If the second least significant bit is 0, the tagged word has a pointer. The third least significant bit is the type flag that indicates whether the data is a VDM's value or a VDM's type. For example, the **`bool`** type is encoded as `0x06` in the tagged word format, and the **`true`** value is encoded as `0x2a`.

The `Data` module also provides functions that converts values between a tagged word and a primitive value namely integer, float, character or a pointer. Figure 4 is an excerpt from the `Data` module. Because VDM-SL does not have bit manipulation operators, arithmetic operators on integers are used instead.

3.3 Registers

ViennaVM has 2^{16} data registers. Each data register has five statically typed fields, namely `oid` (tagged word), `i` (integer), `f` (floating point number), `c` (character) and `p` (pointer). Figure 5 is an excerpt from the definition of ViennaVM's registers in VDM-SL. The `read_int` operation accepts a 16 bits register ID and look for a cached value in the `i` field of the specified register. If the cache is invalid, it yields a 63 bits signed integer value. Because fields in a register and tagged words in memory slots are strongly typed, IR code has strongly typed semantics. Type safety at IR code level will contribute to reliability of applications.

One major overhead of tagged words is the cost of tagging and untagging. Some VMs provide explicit tagging and untagging instructions to convert values. The explicit

```
functions
    oid2int : OID -> Int
    oid2int(oid) ==
        if oid mod 2 = 1
        then
            (if oid <= 0x8000000000000000
             then oid div 2
             else oid div 2 - 0x8000000000000000)
        else
            invalidIntValue;

    int2oid : Int -> OID
    int2oid(i) ==
        if i <> invalidIntValue
        then
            (if i >= 0
             then i  * 2 + 1
             else (0x8000000000000000 + i) * 2 + smallIntegerTag)
        else
            invalidOidValue;
```

Fig. 4. Data conversion functions defined in VDM-SL

tagging/untagging naturally requires those tagging and untagging instructions in the IR code which makes the IR code larger. Also, erroneous IR code may possibly mix up values of wrong types, e.g. use an integer value as a pointer. Other VMs handles only first class objects in the form of tagged words. This approach brings an overhead of massive tagging and untagging operations. For example, when evaluating `(1 + 2) * 3`, the VM should untag to obtain the values `1` and `2`, compute `1 + 2`, tag the resulting value `3` to computer the `1+2` part. Then, the VM untag it, then untag to obtain the value `3`, compute `3 * 3` and then tag the resulting value `9`. This approach is costly in return of type safety.

The basic idea of ViennaVM's five statically typed fields per register is to cache the tagged and untagged data to avoid unnecessary tagging/untagging operations. When evaluating `(1 + 2) * 3`, ViennaVM stores the tagged value of `1` and `2` into the `oid` field of each register. Then, `ADD` instruction will read the integer values from the registers, which will implicitly untag and cache the integer values into the `i` fields, and then stores the resulting value `3` into the `i` field of another register. Then `MUL` instruction will read the integer value, which need untagging operation and stores the resulting value into the `i` field of a register. Instructions that read those registers can later read the integer values cached in the `i` fields. Wrong conversions, e.g. read a float value from a register that has integer value, can be detected at runtime. This approach using statically typed fields in a register can provide reduced cost of tagging and untagging while keeping type safety of the IR code.

```
types
     Reg ::
        oid : Data`OID
        i : Data`Int
        f : Data`Float
        c : Data`Char
        p : Data`Pointer;
     Register = nat inv r == r < 65536;
operations
    read_int : Register ==> Data`Int
    read_int(r) ==
        let reg : Reg = registers(r), i : [Data`Int] = reg.i
        in
            if i = Data`invalidIntValue
            then
                let i2 : Data`Int = Data`oid2int(reg.oid)
                in
                    (if i2 <> Data`invalidIntValue
                    then registers(r) .i := i2;
                    return i2)
            else return i;

    write_int : Register * Data`Int ==> ()
    write_int(r, i) ==
        (let p = registers(r).p
        in
            if p <> Data`invalidPointerValue
            then Memory`decrement_reference_count(p);
        registers(r)
            := mk_Reg(
                Data`invalidTag,
                i,
                Data`invalidFloatValue,
                Data`invalidCharValue,
                Data`invalidPointerValue));
```

Fig. 5. The definition of ViennaVM's registers in VDM-SL

Another benefit of this approach is that garbage collectors (GCs) can accurately count references from registers. Because the pointers in registers are stored in the `p` field, a GC can detect whether or not the value in a register is a pointer or not. Runtime type information at IR code level enables simple and reliable implementation of GC that does not rely on the correctness of the application code.

3.4 Memory model

The `Memory` module defines the memory model of ViennaVM. An object allocated in a heap space has slots that store tagged words and headers for memory management. Figure 6 shows the format of objects allocated in the heap space. Because all data other than the object headers are tagged word, a GC can retrieve type information from the binary data. Unlike most other VMs, ViennaVM's instructions and immediate values in the IR code page are also tagged words and thus subject to garbage collection.

offset	field name	type	description
0	SIZE_OFFSET	32 bits unsigned int	size of this object aligned by 64 bytes
4	FLAGS_OFFSET	32 bits unsigned int	flags for memory management
8	REFERENCE_COUNT_ OFFSET	64 bits unsigned int	reference count for garbage collection
16	FORWARDER_OFFSET	64 bits unsigned int	link to the updated object
24	SLOTS_SIZE_OFFSET	64 bits unsigned int	number of slots in this object
32	SLOT1	64 bits tagged word	the first element of this object
40	SLOT2	64 bits tagged word	the second element of this object
	:		

Fig. 6. Data format of object in a heap space

The heap allocator gives 64 bytes alignment to the required size of an object. ViennaVM uses reference counters to collect unreferenced objects in the heap space. Although GCs based on reference counts have difficulty in detecting objects with cyclic references, ViennaVM assumes tree structured data from specifications in VDM-SL and therefore there will be no cyclic references. ViennaVM manages reference counters from tagged pointers not only in heap objects but also in registers. At every write access to a register or a memory slot, the contents of the old tagged word and the new tagged word are checked, and if a tagged word has a pointer, the reference counter will be increased and/or decreased.

3.5 Code Record and Activation Record

In ViennaVM, VDM functions and operations are implemented as *routines*. A code record is an object that represents a routine that consist of a series of IR code, type signature, precondition, postcondition, measure function and declaration of registers used in the IR code. An activation record, also known as a stack frame, is an object that represents an execution contexts that holds the caller activation record (the dynamic link), the caller code record, the caller's instruction pointer, measure value, old state, register id to pass a return value and slots to save registers.

Code records and activation records are also stored as objects. Like other heap objects, code records and activation records are subject to garbage collection.

3.6 Instructions

ViennaVM's IR instruction set has basic instructions for memory allocation, data transfer, primitive operators, control structures (error, jump, conditional jump, call, recursive call, return and conditional return). Table 1 lists the basic instructions. Since tagged words in heap objects and registers have type information, data transfer instructions and primitive operator instructions perform runtime type checking by default. ViennaVM needs to be adaptive to different target platforms, so each implementation of ViennaVM may or may not perform such runtime checking to manage the balance between safety and computational costs.

Figure 7 is the definition of the `SUB` instruction in VDM-SL. The `SUB` instruction takes three operands each of which specifies a register ID. This instruction computes the second operand minus the third operand and stores the result into the first operand. The VM first checks the operands are specified. If any operand is omitted, the VM issues an error. The definitions part of the let statement defines data retrieval from the registers. The VM tries to read an integer value from the register specified by the second operand. If failed, it tries to read a float value from the same register. The VM does the same to the third operand. Then, the "in" clause of the let statement defines the computation and data transfer to the register specified by the first operand. If both values are successfully retrieved, the VM computes `num1 - num2` and stores it to the integer field of the destination register if the both arguments are integer values. Otherwise, it stores the resulting value to the float field of the destination register.

4 Example IR code and Preliminary Performance Evaluation

ViennaVM is still under development. We have created a prototypical implementation from the current snapshot of its specification in VDM-SL for a preliminary performance check. Although performance is not specifically pointed as a requirement to ViennaVM in Section 2, we conducted a preliminary performance test to check feasibility of the design of ViennaVM. The C version for now implements a subset of the full instruction set of ViennaVM.

4.1 Performance of Fibonacci

Figure 8 shows the specification of the benchmark function `fib`, the fibonacci sequence function. One call to the `fib` function with an argument larger than 1 will make two recursive calls, and therefore costs exponential time.

The fibonacci series function is implemented by C and also IR code of ViennaVM along with the executable VDM-SL specification. Figure 9 shows the IR code. Informal

Table 1. Basic instruction set of ViennaVM

opcode	operand1	operand2	operand3	description
`ERR`				Triggers error handler
`NOP`				Do nothing
`ALLOC`	r1			Allocate an object with the slot size given by r1
`RESET`				Restart with the latest image file
`DUMP`				Dump an image file
`MOVE`	r1	r2		copy r2 to r1
`MOVEI`	r1			copy the given immediate value to r1
`LOAD`	r1	r2	r3	copy the r3-th slot of the object pointed by r2 to r1
`LOAD1`	r1	r2	r3	copy the (r3+1)-th slot of the object pointed by r2 to r1
:				
`LOAD7`	r1	r2	r3	copy the (r3+7)-th slot of the object pointed by r2 to r1
`STORE`	r1	r2	r3	copy r3 to the r2-th slot of the object pointed by r1
`STORE1`	r1	r2	r3	copy r3 to the (r2+1)-th slot of the object pointed by r1
:				
`STORE7`	r1	r2	r3	copy r3 to the (r2+2)-th slot of the object pointed by r1
`ADD`	r1	r2	r3	set r1 to r2 + r3
`SUB`	r1	r2	r3	set r1 to r2 - r3
`MUL`	r1	r2	r3	set r1 to r2 * r3
`IDIV`	r1	r2	r3	set r1 to r2 div r3
`IMOD`	r1	r2	r3	set r1 to r2 mod r3
: ⋆3				
`EQUAL`	r1	r2	r3	set r1 to r2 = r3
`NOTEQ`	r1	r2	r3	set r1 to r2 <> r3
`LESSTHAN`	r1	r2	r3	set r1 to r2 < r3
`LESSEQ`	r1	r2	r3	set r1 to r2 <= r3
`GREATER`	r1	r2	r3	set r1 to r2 > r3
`GREATEREQ`	r1	r2	r3	set r1 to r2 >= r3
`NOT`	r1	r2		set r1 to not r2
`JUMP`	r1			set ip to r1
`JUMPTRUE`	r1	r2		if r1 is true, set ip to r2
`JUMPFALSE`	r1	r2		if r1 is false, set ip to r2
`CALL`	r1	r2		call the code record pointed by r2, save registers and set the return value to r1
`CALLREC`	r1			call the current, save registers and set the return value to r1
`RET`	r1			return to the caller, restore registers and pass r1 as the return value
`RETTRUE`	r1	r2		if r1 is true, return to the caller, restore registers and pass r2 as the return value
`RETFALSE`	r1	r2		if r1 is false, return to the caller, restore registers and pass r2 as the return value

⋆1 other VDM-SL built-in operators on numbers and booleans

```
operations
  sub : Register`Register * Register`Register
        * Register`Register ==> ()
  sub(dst, src1, src2) ==
    if (dst > 0 and src1 > 0) and src2 > 0
    then
      let
        int1 = Register`read_int(src1),
        num1 : [real] =
          if int1 <> Data`invalidIntValue
          then int1
          else
            (let float1 = Register`read_float(src1)
              in
                (if float1 <> Data`invalidFloatValue
                then Data`float2real(float1)
                else nil)),
        int2 = Register`read_int(src2),
        num2 : [real] =
          if int2 <> Data`invalidIntValue
          then int2
          else
            (let float2 = Register`read_float(src2)
            in
              (if float2 <> Data`invalidFloatValue
              then Data`float2real(float2)
              else nil))
      in
        if num1 <> nil and num2 <> nil
        then
          let num3 : real = num1 - num2
          in
            if
              int1 <> Data`invalidIntValue and
              int2 <> Data`invalidIntValue
            then
              Register`write_int(dst, num3)
            else
              Register`write_float(dst, Data`real2float(num3))
        else
          err("sub instruction error: operands not integer")
    else
      err("sub instruction error: operands not specified");
```

Fig. 7. The formal definition of the SUB instruction

```
functions
   fib : nat -> nat
   fib(x) == if x < 2 then x else fib(x-1) + fib(x-2);
```

Fig. 8. The specification of fibonacci sequence function

explanation of each instruction is shown as a comment led by semicolons. As ViennaVM is a register machine, arguments passed to a routine is stored in r1, r2 and so on in order. In this case, the argument `x` is stored in r1. The first instruction, `MOVEI` set the immediate value to the specified register. In this case, this instruction sets the tagged word of `1` to r2. The `LESSTHAN` instruction is a three-operanded opcode that compares the second and the third operands and stores the result to the first operand. The `RETFALSE` instruction is a conditional return. In this case, if r3 is false, then return r1. Then, two sets of `SUB` and `CALLREC` instructions follows that computes fib(x-1) and fib(x-2). The `ADD` instruction will add the two return values from the `CALLREC` instruction and stores the result to r1. Finally, the `RET` instruction returns the r1.

```
                             ; x is passed to r1
MOVEI          2, int(1)     ; r2 <- 1
LESSTHAN       3, 2, 1       ; r3 <- r2 < r3
RETFALSE       3, 1          ; if r3 is false then return r1
SUB            1, 1, 2       ; r1 <- r1 - r2
CALLREC        3             ; r3 <- fib(r1)
SUB            1, 1, 2       ; r1 <- r1 - r2
CALLREC        1             ; r1 <- fib(r1)
ADD            1, 1, 3       ; r1 <- r1 + r3
RET            1             ; return r1
```

Fig. 9. The IR code for fibonacci series function

Table 2 shows the result of performance test. The IR code shown in Figure 9 was executed on ViennaVM in C. The fibonacci series function in Figure 8 was implemented in C as an ideal performance target. The fibonacci series function in VDM-SL was also animated by VDMJ as a baseline performance. VDMJ is a standalone interpreter implemented in Java. VDMJ is used as a baseline in this benchmark because its implementation is mature and has less overhead than GUI-based IDEs.

The ViennaVM was also implemented in Smalltalk using automated code generator and a hand tuning was done to five methods. The Smalltalk version also executed the IR code to check the feasibility of ViennaVM used in ViennaTalk for debugging. The specification of ViennaVM was also animated by VDMJ as a baseline performance against the Smalltalk version.

Two expressions `fib(5)` and `fib(30)` were executed. The prototypical ViennaVM in C performed better than VDMJ and slower than the C version. Although the ViennaVM in C is a naive IR code interpreter without just-in-time or runtime optimisation, it performed reasonably well. ViennaVM in Smalltalk took almost 50 minutes to compute `fib(30)`. The result mentions that the use of ViennaVM on Smalltalk will be limited to debug by step execution. One possible way to debug a computationally demanding application is to use two versions of ViennaVM, in C and in Smalltalk, and combine them by image files. The ViennaVM in C will execute the IR code and dump an image file at a break point. The ViennaVM in Smalltalk will resume the image file to interactively debug by step execution.

Table 2. Performance comparison by fibonacci numbers

	fib(5) (ms)	fib(30) (ms)
C	0.0	11.0
ViennaVM (C)	1.8	410.2
VDMJ	1.8	3,335.4 ⋆2
ViennaVM (Smalltalk ⋆1)	18.4	2,870,079.0
ViennaVM (VDM-SL)	351.6	NA ⋆3

⋆1 automated code generator + hand tuning
⋆2 measured after 10 warm-up runs
⋆3 computation did not finish in 4 hours

4.2 Discussions about perspectives

VMs are commonly used architecture of runtime systems of various programming languages. While most language VMs are designed for programming languages, ViennaVM is specially dedicated for VDM languages. To take VDM's advantage in productivity of developing embedded software, ViennaVM should support not only in the final product, but the prototypical development of smart devices. The most VMs for high level programming languages can assume that the compiler generates valid IR code. ViennaVM, on the other hand, should provide rich runtime checking mechanism for types, states, precondition, postconditions and metrics. Another interesting feature of ViennaVM is that it is specified in VDM-SL. The development of VMs requires high skills of system programming and long development time to debug. We expect the utilisation of formal specification language would improve reliability, productivity and portability of the VMs.

The preliminary performance evaluation in Section 4 shows that the prototypical C implementation of ViennaVM can perform better than the VDMJ interpreter. Although there are still a big performance gap between ViennaVM and native C code, it can be possibly narrowed down by runtime optimisations and just-in-time compilation.

5 Planned Further Development

5.1 User Interfaces

We are planning to define UI instructions in ViennaVM. Smart devices have various user interfaces. While many VMs provide low level functions and let user programs to implement UI frameworks, we will provide high level instructions to absorb the difference of physical user interfaces. Figure 10 illustrates how ViennaVM will provide portability of its application code among different smart devices. *inform a text message to the user* instruction that will play a voice on ViennaVM for automotive smart speakers and display a small notifier on ViennaVM for smartphones. UI instructions of ViennaVM provide hooks to invoke and accept interactions with the user, and each implementation of ViennaVM will supply UI functions available in the target platform.

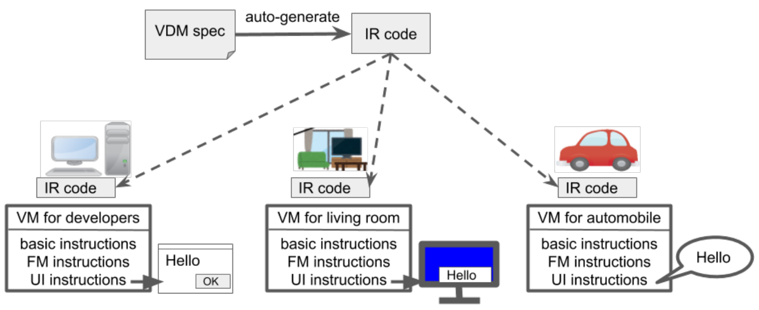

Fig. 10. The same IR code will adapt to different UI devices

5.2 Code Generators and Runtime Support for Formal Methods

We will develop an automated code generator that generates IR code of ViennaVM, and extend ViennaTalk to run an external ViennaVM implemented in C and also an internal ViennaVM implemented in Smalltalk. The internal ViennaVM will be better integrated with ViennaTalk and gives more flexible debugging feature while the external ViennaVM will execute the specification in closer environment to the target platform.

We also plan for extending ViennaVM by instructions that support assertions. A state invariant will be checked at every update to the state variable by setting it as a watch variable. Precondition and postcondition of a function or an operation will be held in a code record, and measure function of a recursive function will be also stored.

6 Conclusions

The development of ViennaVM is still at an early stage. Its objective is to provide a common target platform of automated code generators from VDM. It can also be seen as a case project of VDM in VM developments. We will investigate how VDM can improve the development of language VMs as well as how a dedicated VM may change the process of the model-based development using VDM.

Acknowledgments

The authors would thank Christoph Reichenbach, Kumiyo Nakakoji and Yasuhiro Yamamoto for their inspiring comments on the initial idea of this research. A part of this research was supported by JSPS KAKENHI Grant Number JP 18K18033. We would also like to pay special thanks to the anonymous reviewers who have helped improving the quality of the paper.

References

1. Diswal, S.P., Tran-Jørgensen, P.W., Larsen, P.G.: Automated Generation of C# and .NET Code Contracts from VDM-SL Models. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 32–47. Aarhus University, Department of Engineering, Cyprus (November 2016), ECE-TR-28
2. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future - the story of squeak, a practical smalltalk written in itself. ACM SIGPLAN Notices 32(10), 318–326 (1997)
3. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A Code Generation Platform for VDM. In: Battle, N., Fitzgerald, J. (eds.) Proceedings of the 12th Overture Workshop. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446 (January 2015)
4. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (January 2010), `http://doi.acm.org/10.1145/1668862.1668864`
5. Larsen, P.G., Lausdahl, K., Battle, N.: Combinatorial Testing for VDM. In: Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods. pp. 278–285. SEFM '10, IEEE Computer Society, Washington, DC, USA (September 2010), `http://dx.doi.org/10.1109/SEFM.2010.32`, ISBN 978-0-7695-4153-2
6. Larsen, P.G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Tran-Jørgensen, P.W.V., Oda, T.: VDM-10 Language Manual. Tech. Rep. TR-001, The Overture Initiative, www.overturetool.org (April 2013)
7. Oda, T., Araki, K., Larsen, P.G.: Automated VDM-SL to Smalltalk Code Generators for Exploratory Modeling. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 48–62. Aarhus University, Department of Engineering, Aarhus University, Department of Engineering, Cyprus (November 2016), ECE-TR-28
8. Oda, T., Araki, K., Larsen, P.G.: ViennaTalk and Assertch: Building Lightweight Formal Methods Environments on Pharo 4. In: Proceedings of the International Workshop on Smalltalk Technologies. pp. 4:1–4:7. Prague, Czech Republic (Aug 2016)
9. Smith, J.E., Nair, R.: The architecture of virtual machines. Computer 38(5), 32–38 (May 2005)
10. Tran-Jørgensen, P.W.V., Larsen, P.G., Leavens, G.T.: Automated translation of VDM to JML-annotated Java. International Journal on Software Tools for Technology Transfer pp. 1–25 (2017), `http://dx.doi.org/10.1007/s10009-017-0448-3`