An Embeddable Virtual Machine for State Space Generation

Michael Weber*

SEN2, CWI, Amsterdam, The Netherlands
Michael.Weber@cwi.nl

Abstract. The semantics of modelling languages are not always specified in a precise and formal way, and their rather complex underlying models make it a non-trivial exercise to reuse them in newly developed tools. We report on experiments with a virtual machine-based approach for state space generation. The virtual machine's (VM) byte-code language is straightforwardly implementable, facilitates reuse and makes it an adequate target for translation of higher-level languages like the SPIN model checker's PROMELA, or even C. As added value, it provides efficiently executable operational semantics for modelling languages. Several tools have been built on top of the VM implementation we developed, to evaluate the benefits of the proposed approach.

1 Introduction

Common approaches in state-based model checking employ modeling languages like CSP [11], LOTOS [4], Murϕ [7], DVE [1], or PROMELA[13] to describe actual state spaces. These languages are usually non-trivial: in addition to concepts found in programming languages (scopes, variables, expressions) they provide features like process abstraction, non-determinism, guarded commands, synchronisation and communication primitives, timers, etc. Implementing an operational model of high-level languages for use in verification tools is consequently not straightforward.

That being said, when developing new verification algorithms and tools it is highly desirable to reuse an already existing modeling language like PROMELA, which has been used in a sizeable number of real-world case studies. In our experience, we identified four main benefits. First, we can reuse existing case studies to test new tools and compare to already published results, instead of having to resort to artificial examples. Secondly, tool developers can concentrate on the implementation of algorithms if the part of how model data enters the developed tool is either reuseable or easily reimplemented, and can be incorporated in whatever infrastructure is dictated by the requirements of a new algorithm. From a user perspective, switching to a model checking tool with compatible input language is made easier, as it avoids the penalty of having to reimplement

* This research has been partially funded by the Netherlands Organization for Scientific Research (NWO) under FOCUS/BRICKS grant number 642.000.05N09

the model in another formalism, and showing that the semantics have been preserved in the translation. In addition, existing models can be used to benchmark new tools on realistic data sets. Lastly, by taking the virtual machine as an intermediate layer, we can implement (and reuse!) common analyses like dead variable reduction and statement merging independent of the high-level input language.

Contributions In order to remedy the perceived shortcomings we propose a virtual machine (VM) based approach to state space generation, in which high-level modeling languages are translated to byte-code instructions. Subsequent execution of such byte-code programs with a VM yields state spaces for further use in model checkers, simulators and testing tools. A key point is that the VM is easily embeddable into a host application (for example, a model checker). As such, it should have a formal specification and a straightforwardly implementable execution model, which imposes as little constraints as possible on the tool it is embedded into. In the rest of the paper we present how this can be carried out. We validated the approach with a number of applications.

Organisation In section 2 we describe the virtual machine model and its byte-code semantics. Section 3 summarizes how the virtual machine is used for state space generation in a number of applications: a target for PROMELA compilation, which has been embedded into external-memory and distributed-memory model checkers. As further benefit for tools developers, these tools can be used unchanged to interface with other front-ends, for example, to check C code for embedded systems. We conclude with a summary of related and future work in sections 4 and 5. Appendix A presents benchmark results for our VM implementation to show practical usefulness of our approach.

2 Virtual Machine Specification

The virtual machine (VM) we are using as running example here contains a couple of features not all of which are commonly found at byte-code level in conventional VM architectures like the Java Virtual Machine (JVM) [15]. They are a superset of the features we observed as common in modeling languages. In particular, we have:

Non-determinism If non-deterministic choice is encountered during executing, the machine offers all possible continuations to the scheduler who then decides which path to take.

Concurrency Processes can be created, not only statically but also during execution of the model.

Communication Both, rendezvous and asynchronous channel objects are provided for inter-process communication.

First-class channels Like in PROMELA and π-calculus [14], channels can be sent over channels, thus allowing for a dynamic communication structure.

Priority scheme Our byte-code allows to specify which actions have to be given preference. Together with explicit control over externally visible actions, this allows to encode high-level constructs like PROMELA's `atomic` and `d_step`.

Speculative execution Code sequences like guards are executed speculatively, and changes to the global state are rolled back if the sequence does not run to completion (see Section 2.4).

External Scheduling Scheduling decisions are delegated to host applications. This allows for implementation of different scheduling policies which is needed to cater for simulation (interactive scheduling) vs. state space exploration with some search strategy (breadth-first, depth-first, random, or combinations thereof).

The design of our VM was mainly driven by pragmatic decisions: it was our intention to create a model that is simple, efficient and embeddable as component into host applications, with implementation effort split between the VM and compilers targeting it. For example, many instructions make use of the VM's stack because it is trivial for compilers to generate stack-based code for expression evaluation. On the other hand, a stack-based architecture alone is inconvenient for translation of counting loops, thus registers were added. The RISC-like instruction set is motivated by the need for fast decoding inside the instruction dispatcher, the VM's most often executed routine.

Although our machine is a mixture of register-based and stack-based architecture, we are nevertheless dealing with finite state models in this paper by putting bounds on all resources. Concurrency is modeled by interleaving semantics.

A complete specification of a virtual machine suitable as target for PROMELA is available [20]. Our starting point was a simple VM model, which we then extended with features needed to cater for PROMELA's semantics. However, in the interest of reusability we tried to keep these additions as generic as possible (see Section 3.4).

In the following, we will present a formalisation of the VM which is suitable for implementation. We found this an invaluable help in allowing different groups working independently on compilers, byte-code optimizers and the VM itself. It also serves as a reference in case the VM needs to be reimplemented, or for answering questions regarding the semantics of compiled languages.

We start by specifying global and local state, and invariants which translations must preserve. Afterwards we present the byte-code semantics and how scheduling between alternatives is done.

2.1 Machine State

The machine's global state as depicted in Figure 1 consists of a few global objects and the local state of its processes.

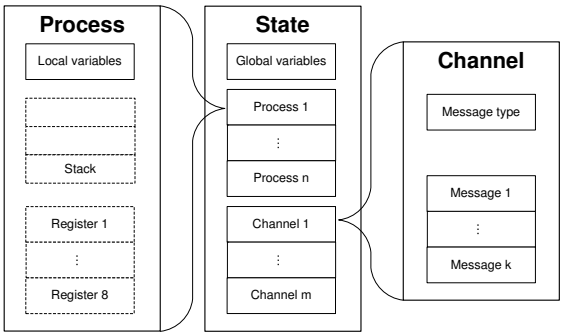

Fig. 1. Overview over the state of the virtual machine. Dotted borders around registers and stack indicate that they are only temporarily part of the machine state, but are not preserved.

Definition 1 (Global State). *The global state* $\Gamma = \langle \Pi, e, G, \Phi \rangle$ *of our virtual machine is a tuple*

$$\Gamma \in \mathit{Processes} \times \mathit{Pid}_\bot \times \mathit{Mem} \times \mathit{Channels}$$

with Π *denoting a finite set of processes,* e *the process identifier of a process with exclusive execution privileges (*$\bot$ *if none),* G *the global variable store, and* Φ *the—again finite—set of existing channels (channels are global objects).*

We will refer to the set of all global states as Γ as well, if the context makes clear what is meant.

Definition 2 (Process). *A process* $\pi = \langle p, M, \Lambda' \rangle$ *is a tuple*

$$\pi \in \mathit{Processes} = (\mathit{Pid} \times \mathit{ExecMode} \times \mathit{ProcessState}') \cup \{\mathrm{stop}\}$$

with p *denoting a globally unique identifier,* $M \in \{\underline{\mathrm{N}}, \underline{\mathrm{A}}, \underline{\mathrm{I}}, \underline{\mathrm{T}}\}$ *its execution mode (normal, atomic, invisible, terminated), and* Λ' *the local state of a process (Definition 3).*

Furthermore, we allow the special symbol stop *to denote a deadlocked process which cannot make any further step.*

A process can be either inactive or active.

While a single process can be deadlocked, there might be others which can still continue, so that there is no *global deadlock* yet.

Often, we do not want a global state $\Gamma = \langle \Pi, e, G, \Phi \rangle$ to contain the deadlocked process stop. To simplify notation, we write $\Gamma \neq \mathrm{stop}$ iff no process in Π is deadlocked: $\forall \pi \in \Pi : \ \pi \neq \mathrm{stop}$.

Definition 3 (Local Process State). *A local process state* $\Lambda' = \langle L, m \rangle$ *is a pair*

$$\Lambda' \in \mathit{ProcessState}' = \mathit{Mem} \times \mathbb{N}$$

and denotes the process-local variable store L and its program counter m.

When a process becomes active, its state Λ' is augmented with registers R_0 and a stack $D_\epsilon = \epsilon$ to its active local state *$\Lambda = \langle L, m, R_0, D_\epsilon \rangle$:*

$$\Lambda \in \mathit{ProcessState} = \mathit{Mem} \times \mathbb{N} \times \mathit{Registers} \times \mathit{Stack}$$

When it becomes inactive again, its last two components are projected away. As a result, they are to be used for storing temporary values only.

Definition 4 (Store). *We identify three stores in our virtual machine model: for global (G) and local variables (L), and for registers (R). As usual, we model stores as mappings $\sigma \in \mathbb{N} \to \mathit{Value}$, that is for a store σ, $\sigma[i]$ denotes the store's value at position i. Replacing a value v at position i in the store is written as $\sigma[i/v]$.*

Initial stores are denoted as σ_0 ($\forall i : \ \sigma_0[i] := 0$). For convenience, we write r_i to reference the ith register $R[i]$.

We added registers to our virtual machine for situations when byte-code effects on the machine's state are not fitting well to a stack model, for instance if values are operated on more than once.

Definition 5 (Data Stack). *Expression evaluation takes place on the data stack component $D \in \mathit{Stack} = \mathit{Value}^*$ of a process state. A stack is represented as finite (possibly empty) word $D = v_n : \cdots : v_1, \quad v_i \in \mathit{Value}, n \in \mathbb{N}$.*

We denote the empty stack as $D_\epsilon = \epsilon$.

Communication Processes can use several ways to communicate values among each other. First, they can use the global store G which can be modified by any process at any time. A more structured way of communication is provided by means of channels. They also offer a model for message-passing synchronization. In our machine, communication channels are typed and bounded, and we distinguish between rendezvous channels and asynchronous channels.

Definition 6. *A* channel *$\phi = \langle c, l, t, C \rangle$ is a tuple*

$$\phi \in \mathit{Channels} = \mathit{ChanId} \times \mathbb{N} \times \mathbb{N} \times \mathit{Message}^*$$

with c denoting a globally unique channel identifier, l the channel capacity, and $C = c_0 : \cdots : c_l$ its current contents (c_l being the last message in the channel). Each message $c_i \in \mathit{Message} = \mathit{Value}^*$ consists of a sequence of values of length t.

Rendezvous channels have zero capacity. A message can temporarily be stored in a channel during rendezvous communication, hence exceeding the capacity of the channel. Such states are internal to the virtual machine and unobservable to its outside. Similarly, asynchronous channels which exceed their capacity automatically fall back to the same behavior as rendezvous channels: send operations on those block until they are within their allowed capacity again.

Definition 7 (Rendezvous Communication). *We define a predicate* $\mathrm{sync}(\Gamma)$ *on a global state* $\Gamma = \langle \Pi, e, G, \Phi \rangle$ *to determine whether rendezvous communication is taking place: at least one channel* $\phi = \langle c, l, t, C \rangle$ *contains more messages than its capacity* l *allows.*

$$\mathrm{sync}(\Gamma) := \begin{cases} \text{true} & \text{if } \exists \phi = \langle c, l, t, C \rangle \in \Phi : \ |C| > l \\ \text{false} & \text{otherwise} \end{cases}$$

2.2 Invariants

Translation to our byte-code language must guarantee the following invariants: as already pointed out in Definition 3, a process becoming active again always resumes execution with register set R_0 and the empty stack D_ϵ. Conversely, at those points in the program when a process may become inactive, the contents of registers and stack are discarded and need not matter for the rest of its execution.

Because the number of local variables is fixed, a local state Λ' occupies constant space only.

2.3 Byte-code Semantics

Having defined the state of our virtual machine, we now proceed by defining the semantics of operations on it. These operations are carried out at process level, with only a single process being active at once.

In the spirit of earlier displays of Promela semantics by Holzmann and Natarajan [13], we compose our semantics from several smaller parts by defining four relations to model process activation, internal, intermediate and finally scheduler transitions.

A transition from state Γ_1 to Γ_2 is a relation $\rightarrow_T \in \Gamma \times \Sigma_T \times \Gamma$, with a finite set of labels Σ_T and set of states Γ. If not important, we will elide labels from our presentation. For brevity, we write $\Lambda_1, G_1, \Phi_1 \rightarrow \Lambda_2, G_2, \Phi_2$ instead of

$$\begin{aligned} &\langle \{ \langle p, M, \Lambda_1 \rangle, \pi_1, \ldots, \pi_n \}, e, G_1, \Phi_1 \rangle \\ &\qquad \rightarrow \langle \{ \langle p, M, \Lambda_2 \rangle, \pi_1, \ldots, \pi_n \}, e, G_2, \Phi_2 \rangle \\ &\qquad\qquad \pi_i = \langle p_i, M_i, \langle L_i, m_i \rangle \rangle \text{ for all } 1 \leq i \leq n \end{aligned}$$

State components remaining unchanged in a transition are left out.

As mentioned before, only one process can be active at any point in time. Thus we define process activation as transition

$$\begin{aligned} &\langle \{ \langle p, M, \langle L, m \rangle \rangle, \pi_1, \ldots, \pi_n \}, e, G, \Phi \rangle \\ &\qquad \xrightarrow{p}_{act} \langle \{ \langle p, M, \langle L, m, R_0, D_\epsilon \rangle \rangle, \pi_1, \ldots, \pi_n \}, e, G, \Phi \rangle \\ &\qquad\qquad \forall i \in \{1, \ldots, n\} : \ \pi_i = \langle p_i, M_i, \langle L_i, m_i \rangle \rangle \\ &\qquad\qquad\qquad \text{and } e \in \{p, \perp\}, M \neq \underline{\mathrm{T}} \end{aligned}$$

`LDC` c	load constant c onto top of data stack $\langle L, m, R, D\rangle \rightarrow_{int} \langle L, m+1, R, D : c\rangle$
`LDV` g	load variable onto top of data stack $\langle L, m, R, D : a\rangle \rightarrow_{int} \langle L, m+1, R, D : L[a]\rangle$ if $g = \underline{\mathrm{L}}$ $\langle L, m, R, D : a\rangle, G \rightarrow_{int} \langle L, m+1, R, D : G[a]\rangle, G$ if $g = \underline{\mathrm{G}}$
`STV` g	store stack top in variable $\langle L, m, R, D : v : a\rangle \rightarrow_{int} \langle L[a/v], m+1, R, D\rangle$ if $g = \underline{\mathrm{L}}$ $\langle L, m, R, D : v : a\rangle, G \rightarrow_{int} \langle L, m+1, R, D\rangle, G[a/v]$ if $g = \underline{\mathrm{G}}$
`POP` r_i	pop top-most value from stack into register $\langle L, m, R, D : v\rangle \rightarrow_{int} \langle L, m+1, R[i/v], D\rangle$
`PUSH` r_i	push value from register onto stack $\langle L, m, R, D\rangle \rightarrow_{int} \langle L, m+1, R, D : r_i\rangle$

Table 1. Load and Store byte-codes

A process needing exclusive execution privileges *must* be activated, otherwise any process can be activated ($e = \bot$). Processes already run to completion ($M = \underline{\mathrm{T}}$) are not activated again.

Next, we define those transitions an active process can possibly take: the *internal-step* relation $\rightarrow_{int} \in \Gamma \times \Gamma$ is the least relation satisfying the rules given below. For reasons of presentation, we divided internal steps into several parts. Note that the byte-code operation to be executed next is determined by indexing program counter m of the currently active process into a global instruction list $\underline{\mathrm{Instr}}$.

Load and Store Our machine supports usual operations to load constants (`LDC`), and manipulate values of local and global variables (`LDV`, `STV`), as defined in Table 1. The differentiation of local and global store access simplifies byte-code analysis for, e.g., statement merging.

To avoid stack juggling operations like `DUP`, `SWAP`, etc., values can be stored into and retrieved from registers with `PUSH` and `POP`.

Arithmetic and Boolean Operations Expression byte-codes like `ADD`, `LT`, `AND`, `NEG`, etc., operate on one or more of the stack's top-most entries. Their semantics are obvious and thus only defined exemplarily:

$$\texttt{OP}_{\otimes}: \quad \langle L, m, R, D : u : v\rangle \rightarrow_{int} \langle L, m+1, R, D : u \otimes v\rangle$$

Control-flow Operations For control flow changes, we define conditional and unconditional jumps in Table 2. In order to allow explicit modeling of non-determinism, we define `NDET` a as having two possible successor states: one continuing with the next instruction and the other continuing at instruction a. In some situations, it is helpful to allow *conditional non-determinism*, where the existence of one alternative is dependent on the presence or absence of another.

`JMPNZ` a	jump if non-zero
	$\langle L, m, R, D : 0\rangle \rightarrow_{int} \langle L, m+1, R, D\rangle$
	$\langle L, m, R, D : v\rangle \rightarrow_{int} \langle L, a, R, D\rangle$, if $v \neq 0$
`NDET` a	non-deterministic jump
	$\langle L, m, R, D\rangle \rightarrow_{int} \langle L, m+1, R, D\rangle$
	$\langle L, m, R, D\rangle \rightarrow_{int} \langle L, a, R, D\rangle$
`ELSE` a	else jump
	$\langle L, m, R, D\rangle \rightarrow_{int} \langle L, m+1, R, D\rangle$
	$\langle L, m, R, D\rangle \rightarrow_{int} \langle L, a, R, D\rangle$ if $\langle L, m+1, R, D\rangle \rightarrow^*_{int} \Lambda' \rightarrow_{end}$ stop
`UNLESS` a	unless jump
	$\langle L, m, R, D\rangle \rightarrow_{int} \langle L, a, R, D\rangle$
	$\langle L, m, R, D\rangle \rightarrow_{int} \langle L, m+1, R, D\rangle$ if $\langle L, a, R, D\rangle \rightarrow^*_{int} \Lambda' \rightarrow_{end}$ stop
`CALL` a	call subroutine
	$\langle L, m, R, D\rangle \rightarrow_{int} \langle L, a, R, D : m+1\rangle$
`RET`	return from subroutine
	$\langle L, m, R, D : a\rangle \rightarrow_{int} \langle L, a, R, D\rangle$

Table 2. Control-flow byte-codes

For this, we add byte-codes `ELSE` a and its dual `UNLESS` a. They are used in the translation of PROMELA, for example.

`CALL` a and `RET` can be used to translate procedure calls. By default the return address is left on the stack, thus does not survive if a process becomes inactive. It is in the responsibility of the compiler to store it inside the state vector. This allows for some flexibility when dealing with recursive functions. In general, their treatment requires some cooperation between the compiler and an analysis tool working with the generated state space.

As PROMELA itself does not allow function calls, these byte-codes are not used in its translation. However, they have been used to compile method calls of an object-oriented language [21].

Operations on Channels For inter-process communication, our virtual machine model contains several operations on channels. These include operations to dynamically create channels, query their properties, and manipulate their contents. Both, synchronous and asynchronous channels are supported.

Because of space constraints, we elide their treatment here and refer to the full specification [20]. However, we will return to the topic of synchronous communication in Section 2.4, when discussing process scheduling.

Spawning New Processes To start a new process, its current parameters are placed onto the data stack. Specifying the size of these parameters and the start address of its code, a new process is instantiated:

`STEP` M'	step complete with mode M' $\langle\{\langle p, M, \langle L, m, R, D\rangle\rangle\} \cup \Pi, e, G, \Phi\rangle$ $\xrightarrow{M'}_{end} \langle\{\langle p, M', \langle L, m+1\rangle\rangle\} \cup \Pi, e', G, \Phi\rangle$ $e' := \begin{cases} p & \text{if } M' \in \{\underline{\mathrm{A}}, \underline{\mathrm{I}}\} \\ \bot & \text{otherwise} \end{cases}$ and $\forall \pi_i \in \Pi :\ \pi_i = \langle p_i, M_i, \langle L_i, m_i\rangle\rangle$
`NEX`	step not executable $\langle L, m, R, D\rangle \not\rightarrow_{end}$ stop

Table 3. Operations for Process Deactivation

`RUN` k, a	run a new process starting at address a
$\langle\{\pi, \pi_1, \ldots, \pi_n\}, e, G, \Phi\rangle \rightarrow_{int} \langle\{\pi', \pi_1, \ldots, \pi_n, \pi''\}, e, G, \Phi\rangle$	
	with $\pi = \langle p, M, \langle L, m, R, D : v_0 : \cdots : v_{k-1}\rangle\rangle$
	and $\pi' = \langle p, M, \langle L, m+1, R, D : p''\rangle\rangle$
	and $\pi'' = \langle p'', \underline{\mathrm{N}}, \langle L_0[0/v_0, \ldots, k-1/v_{k-1}], a\rangle\rangle$
	and $p'' \in \mathit{Pid}$ a unique process identifier

Deactivation of Processes Following a cooperative multitasking approach, eventually a process allows resumption of other processes by deactivating itself with one of the operations in Table 3.

We introduce `STEP` M as flexible means to control which states become visible to an external scheduler. If further execution of a process is not anticipated (e.g. because of unsatisfied guard conditions or reception attempts on empty channels), process execution may be aborted explicitly by `NEX`. This byte-code instruction can be used to translate guards—boolean conditions which can enable or disable a transition. By attaching an action label to a `STEP`, we can cater for action-based setups as well.

2.4 Scheduling

With all the machinery in place, we now proceed with the relation of *scheduler transitions*, $\rightarrow_{sched}$. We define it in terms of *intermediate transitions* $\rightarrow_{step}$, which is the least relation satisfying

$$\Gamma \xrightarrow{p,M}_{step} \Gamma' \quad \text{if} \quad \Gamma \xrightarrow{p}_{act} \Gamma_0 \rightarrow^*_{int} \Gamma_1 \xrightarrow{M}_{end} \Gamma'$$

This means, that in a machine state Γ some process identified as p is activated, then a number of internal transitions happen, until at some point the process deactivates itself in state Γ', giving the whole sequence mode M.

In case the machine gets "stuck" without successor states because some process with exclusive execution privileges becomes deadlocked, this process loses

them, thus enabling execution possibilities for other processes:

$$\langle \Pi, e, G, \Phi \rangle \xrightarrow{p,M}_{step} \Gamma' \quad \text{if} \quad \langle \Pi, e, G, \Phi \rangle \xrightarrow{e,\text{-}}_{step} \text{stop}$$
$$\text{and} \quad \langle \Pi, \bot, G, \Phi \rangle \xrightarrow{p,M}_{step} \Gamma'$$

We can then define the transitions visible to an external scheduler. The approach we took is due to our decision to model rendezvous communication within the interleaving model and thus using an intermediate state which is not revealed to the scheduler. We can distinguish three cases: a process ends a sequence of invisible steps with either a visible transition or a transition leading to deadlock, and no interim rendezvous communication can take place, or, rendezvous communication can take place, with the restriction that the sending and receiving halves of the communication must be consecutive.

Definition 8 (Scheduler Transition). *We define the scheduler transition relation* $\xrightarrow{p}_{sched}$ *as least relation satisfying the following rules.*

- *A scheduler transition consists of a (possibly empty) sequence of invisible steps, followed by a visible step, that is, a step with mode* $\underline{\text{N}}$ *(normal),* $\underline{\text{A}}$ *(atomic) or* $\underline{\text{T}}$ *(terminated). None of the steps is a rendezvous communication.*

$$\Gamma \xrightarrow{p}_{sched} \Gamma' \;\; \textit{if} \;\; \Gamma = \Gamma_1 \xrightarrow{p,\underline{\text{I}}}_{step} \cdots \xrightarrow{p,\underline{\text{I}}}_{step} \Gamma_{n-1} \xrightarrow{p,M}_{step} \Gamma_n = \Gamma'$$
$$\textit{and } \forall i: \; \neg\text{sync}(\Gamma_i) \textit{ and } M \neq \underline{\text{I}} \textit{ and } \Gamma' \neq \text{stop}$$

- *Alternatively, if a sequence of invisible steps leads to a deadlocked process, the last step right before the deadlock becomes visible* irrespectively of its mode $\underline{\text{I}}$.

$$\Gamma \xrightarrow{p}_{sched} \Gamma' \;\; \textit{if} \;\; \Gamma = \Gamma_1 \xrightarrow{p,\underline{\text{I}}}_{step} \cdots \xrightarrow{p,\underline{\text{I}}}_{step} \Gamma_{n-1} \xrightarrow{p,\text{-}}_{step} \text{stop}$$
$$\textit{and } \forall i: \; \neg\text{sync}(\Gamma_i) \textit{ and } \Gamma' = \Gamma_{n-1}$$

- *Lastly, we allow a rendezvous channel to actually contain one message more than its capacity allows, if the immediately following transition resolves this again by having a rendezvous partner receiving this message, so that said rendezvous channel is within its limits again and the resulting state becomes visible to the scheduler again. In this case the sender loses its execution privilege. It can then be picked up by the receiver. Note that we do not allow a process to have rendezvous communication with itself* ($p \neq p'$).

With this mechanism, rendezvous communication can be used to pass around execution privileges between processes like in Promela.

$$\Gamma \xrightarrow{p}_{sched} \Gamma'' \;\; \textit{if} \;\; \Gamma \xrightarrow{p,M}_{step} \Gamma' = \langle \Pi', e', G', \Phi' \rangle \;\; \textit{and} \; \langle \Pi', \bot, G', \Phi' \rangle \xrightarrow{p',M'}_{step} \Gamma'' \;\; \textit{and} \; \mathrm{sync}(\Gamma') \;\; \textit{and} \; \neg\mathrm{sync}(\Gamma'') \;\; \textit{and} \; p \neq p' \;\; \textit{and} \; \Gamma'' \neq \mathrm{stop} \;\; \textit{and} \; M \neq \underline{\mathrm{T}}$$

In all cases, we do not allow a scheduler transition to lead to a global state containing a deadlock process stop.

Our handling of deadlock processes allows us to define a global deadlock state Γ where no process can complete a scheduler transition naturally: there is no Γ' such that $\Gamma \xrightarrow{p}_{sched} \Gamma'$.

Definition 9 (Initial State). *The scheduler starts program execution with the initial state of our machine:* $\Gamma_{init} = \langle \{\langle 1, \underline{\mathrm{N}}, \langle L_0, init \rangle \rangle\}, \bot, G_0, \emptyset \rangle$

The actual interface to run state space generation is not described here, as it is largely based on the same principles as Open/CÆSAR [8]. In fact, a preliminary test has shown that we can connect our implementation to CADP with little effort, thus leveraging their large toolset.

3 Applications

The described virtual machine has been utilized successfully in a number of projects[1], which we briefly detail below. On the same time, these projects served as testbed to check whether the virtual machine based approach we advocate is generic enough to accomodate different modelling languages and verification frameworks.

3.1 Promela

We validated our virtual machine-based approach to state space generation by defining a translation from Promela to byte-code. As positive side-effect, we obtain an operational semantics for Promela which in particular is suitable for classical compiler-based analyses and also for reimplementation. A complete translation procedure is described by Schürmans [20]. Separate from the compiler and thus Promela, we developed several common optimizations for static state space reduction on the byte-code level, for example, dead variable reduction and a variant of statement merging.

[1] `http://www.cwi.nl/~weber/nips/`

Although other modeling languages could have been used just as well, PROMELA was chosen because it is a truly non-trivial example and it has wide acceptance inside and outside academia.

Benchmarks of our virtual machine show that it performs well enough to be of practical use (Section A) on its own. In combination with the projects described in the following, we could even obtain results which so far have been out of reach.

3.2 An External-Memory Model Checker

The virtual machine is used as state-space generation component in an adaptive external-memory model checking tool [10].

As main memory is still the most restricting factor in state space generation and model checking of industrial-scale models, we developed an algorithm which gradually moves parts of the state space to hard disk when memory fills up. Thus, as long as enough memory is available, it behaves mostly like a regular, memory-bound algorithm.

In unmodified state-space exploration algorithms, checking whether a state has already been visited requires random access to the *closed set* due to commonly used hashtable schemes. In a disk-based setting, such access patterns are prohibitively expensive because they incur large latency when reading from hard disk, in comparison to memory accesses. We get around these limitations by reordering queries such that disk access is avoided if at all possible (through caching strategies) and, failing that, queries are carried out at least in large groups rather than one by one. Besides compression, this allows us also to access the state space stored on disk in a linear fashion, which is orders of magnitude faster than random access.

The amount of main memory available still influences the time needed for full state space generation, however it does not impose a hard limit anymore. With this out of the way, we were able to benchmark our algorithm: the unmodified virtual machine, together with the PROMELA compiler mentioned in Section 3.1 allowed us to use models of real case studies as benchmark material, instead of being constrained to artificial models. In addition, we were able to compare our results with prior experiments.

An short summary is given in Section A. Some of the large models, for example LUNAR scenario 4(d) [23], have previously been reported as exceeding the capabilities of SPIN with 4 GB RAM, with both partial order reduction and `COLLAPSE` enabled. In contrast, we performed state space generation of the mentioned LUNARscenario with a memory limit of 2.5 GB RAM and without partial order reduction (Appendix A).

3.3 NIPS and DiVinE

An alternative to the external-memory model checker described in Section 3.2, is the use of *distributed algorithms* in verification to get around memory limitations

of a single computer. Much research has been devoted to this theme in recent years, for a motivation and recent overview we refer to [5].

The DiVinE library [2] has been conceived as a toolkit and testbed for distributed model checking algorithms, with among other things, an emphasis on LTL model checking. While DiVinE features its own modelling language, DVE, we can apply their algorithms unmodified on Promela models, through the use of our virtual machine. In effect, the combination of the two libraries yields a distributed model checker for Promela, with, at the time of writing, five different LTL model checking algorithms to choose from.

Moving from a sequential to a distributed setting requires some consideration. In particular, the design of data structures must support relocation to other computers. For our virtual machine, this means that snapshots of its run-time state can be captured and send to another computer. This is particularly easy in our case, as a snapshot is represented opaquely in an architecture-independent, continuous array of binary data which can be written directly to a network connection, without a potentially costly serialization step. Heterogenous distributed environments are supported as well.

In addition, we can redecide on analysis tools without having to modify or rewrite our models, solely depending on the availability of computing resources and harddisk. For example, using DiVinEs distributed algorithms gives much faster results usually, however, if the used computing cluster is busy, a job may spend days in the batch queue before being processed, thus making our external-memory algorithm a viable alternative.

3.4 Model Checking Embedded Systems Software

In the previous sections, we have mainly highlighted the use of our virtual machine as target for Promela. Despite it being the initial inspiration, we aimed at designing a generic framework which can cope with different modelling formalisms. As our litmus test we have based the mcess (short for *Model Checking Embedded Systems Software*) project on our virtual machine. We proceed with a short summary, a more detailed description is given elsewhere [19, Sect. 5.2ff].

Embedded systems based on microcontrollers are often used in safety-critical environments. In mcess, we address the problem of checking correctness of code written for particular microcontrollers. Regrettably, and despite the sensitivity of the application area, often no formal specifications exist on such projects, so we either have to extract a specification (semi-)manually, or base our analysis *directly* on the implemention under scrutiny. Matters are complicated further by the fact that systems are implemented in a mixture of assembly language and C, most often utilizing specific hardware idiosyncrasies of these severely resource-constrained devices. Previous case studies have shown that existing C model checkers are not directly applicable to such implementations due to the hardware-specific nature [18].

Instead of trying to parse and analyze source code, we chose to compile it with an off-the-shelf C compiler (which is often supplied by the microcontroller vendor), and take the generated binary executable as starting point. We rely on

the generated debugging information to present results back to the user. The approach allows us to process assembly and C code in one go. Also, we successfully sidestepped dealing with the complex syntax and even more daunting semantics of C.

Conceptually, assembly language is much easier to formalize, and its semantics are usually precisely described by the vendor. To take a concrete example, we chose the widely used ATMEL ATmega family of microcontrollers, and implemented a translator from ATmeta16 assembly (or rather disassembly, to be precise) to our virtual machine instruction set. For many of the instructions, the translator itself has been *generated* semi-automatically from the semantics given in the ATMEL specification. The critical parts are the hardware dependencies like interrupts (modeled as processes), I/O ports, timers (replaced by non-determinism and abstractions by the compiler), etc. These require (one-time) manual effort, for example, to obtain a closed system.

A number of factors contribute to the viability of this approach: the type of microcontrollers we are dealing with very space-constrained, typically they have memory in the order of 1024 Bytes. Unsurprisingly, memory allocation is mostly done completely static (no calls to `malloc()`). A limited hardware stack precludes recursive function calls. All these factors make a straight-forward translation much more amenable to yield good results. Deeper analyses can then be layered on top of that.

For state space generation and model checking of such microcontroller programs, we can again utilize the tools described in the previous sections without extra effort. They are well suited to deal with the potentially large state spaces.

4 Related Work

4.1 Promela Semantics

Several formal semantics for Promela have been proposed in the past, but it turns out that none of them covers all aspects of the language. The original publication [13] is incomplete in this sense and now partly outdated, as SPIN evolved. It was improved on by a more modular and less implementation-specific approach by Weise [22], but there the handling of nested `do` loops in combination with `goto` statements is unsound. Another incomplete attempt is from Bevier [3]. The specification is a Lisp program and as such peppered with implementation artefacts.

In contrast, we developed a compiler for Promela targeting the virtual instruction set defined in Section 2.3. Our translation aims at being faithful to SPIN's Promela semantics. It mainly deviates in allowing nested scopes, in order to straighten out the rather confusing static semantics of declarations (variables can be used before being declared). However, we concede that regrettably there are no good means to assure this except continual testing with publicly available models against SPIN as reference implementation.

4.2 Virtual Machines

Virtual machines have been used extensively in Computer Science. A well-known example is the work of Wirth on the Pascal programming language [24].

Independent to our work, two (unpublished, to the best of our knowledge) attempts of virtual machine models for restricted PROMELA-like languages have been brought to our attention [9,17]. Geldenhuys [9] describes a virtual machine as part of the general design of a model checker, while our work is focused on providing a reusable component for state space generation.

ESML [6], the high-level language translated into byte-code is restricted in several ways when compared to PROMELA, and its underlying virtual machine inherits some of these restrictions. For example, it lacks support for asynchronous channels, shared variables and dynamic process creation.

Rosien [17, Section 8] describes some shortcomings of his attempt, for example the lack of arrays, no support for data types beyond integers, unclear semantics for `do` loops or handshake communication inside atomic blocks ("[...] causes undesired results, unexpected atomic deadlocks or otherwise erratic behavior."). Besides that, Rosien's design did not take into account, e.g., distributed settings where successive states may be generated on different computers.

Both papers do not provide a formal model of their VM or of the translation into their byte-code language, making it non-trivial to derive implementations from their work. Neither are implementations readily available.

Bogor From existing model checking frameworks, we found the Bogor framework [16] closest to the work presented here. It is an extensible framework for software model checking, in particular object-oriented software. Its intermediate representation (BIR) is a high-level guarded command language, not unlike PROMELA. While it can be translated further down to a certain extent, constructs like arrays, locks, exceptions, and high-level control constructs remain, complicating an implementation of its operational semantics.

The Bogor framework consists of a large Java code base, which ruled it out when we were looking into possibilities to interface with other languages. In contrast, our VM implementation itself comprises less than 5000 lines[2] of C and has been interfaced efficiently with C, C++ and Java, and connected to model checking frameworks aimed at high performance like DIVINE.

While Bogor and the work presented here share some common goals, our focus is on embedding into other applications, and thus we aim to show the feasability to provide a reusable *library*, rather than a framework (which might hamper its integration with a host application with incompatible structure.)

From the tool point of view, our aim is not to beat the Bogor framework in terms of features, but rather to provide a small but versatile component which can easily be reused, or written from scratch based on a formal specification.

[2] according to SLOCCount, `http://www.dwheeler.com/sloccount/`

5 Conclusions

We presented a virtual machine-based approach to state-space generation, in which the virtual machine's instruction set doubles as intermediate language. Assigning operational semantics in such a way makes them straightforwardly implementable, thus encouraging reuse. Among the byte-code instructions are all operations commonly needed for the specification of concurrent systems: nondeterminism, process creation, communication primitives, and a way to express scheduler constraints (atomic regions). State-space generators derived in such a way can be small and portable, while benchmarks with a concrete implementation showed that we can obtain practically usable results.

However, some critical thoughts are in order. For example, it is possible to relate analysis results like error traces from the VM back to the original input (PROMELA or C), but there is no stable interface yet available.

Also, a command line simulator is available, yet while working towards integration of our VM implementation in IBM's Eclipse IDE, we found the need for deeper introspection of the VM state. Providing a suitable interface without slowing down state space generation requires some more research. This is worthwhile because we are using the same code for simulation and model checker, thereby foregoing deviations in results. For example, to the best of our knowledge SPIN has been plagued from time to time with the interactive simulation and a model checking run yielding different outcomes.

Nevertheless, with our applications we have shown benefits to be expected through synergy effects of developing an embeddable component for use in third-party tools.

Acknowledgement We thank Michael Rohrbach and Stefan Schürmans for their implementation efforts and valuable discussions. Theo Ruys brought Rosien's work to our attention. Part of this research has been carried out at RWTH Aachen.

References

1. J. Barnat, L. Brim, I. Černá, and P. Šimeček. DiVinE – Distributed Verification Environment. Submitted to PDMC'05's short presentations., 2005.
2. J. Barnat, L. Brim, I. Černá, and P. Šimeček. DiVinE the distributed verification environment. In M. Leucker and J. van de Pol, editors, *4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'05)*, Lisbon, Portuga, July 2005.
3. W. Bevier. Towards an operational semantics of PROMELA in ACL2. In *Proceedings of the 3rd International SPIN Workshop*, April 1997.
4. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers North-Holland, 1989.
5. L. Brim. Distributed verification: Exploring the power of raw computing power. In L. Brim, B. Haverkort, M. Leucker, and J. van de Pol, editors, *Formal Methods: Applications and Technology*, volume 4346 of *Lecture Notes in Computer Science*, pages 23–34. Springer, August 2006.

6. P. de Villiers and W. Visser. ESML—a validation language for concurrent systems. In J. Bishop, editor, *7-th Southern African Computer Symposium*, pages 59–64, July 1992.
7. D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *ICCD '92: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 522–525, Washington, DC, USA, 1992. IEEE Computer Society.
8. H. Garavel. OPEN/CAESAR: An open software architecture for verification, simulation, and testing. *Lecture Notes in Computer Science*, 1384:68–??, 1998.
9. J. Geldenhuys. Efficiency issues in the design of a model checker. Msc. thesis, University of Stellenbosch, South Africa, November 1999.
10. M. Hammer and M. Weber. "To Store or Not To Store" reloaded: Reclaiming memory on demand. In L. Brim, B. Haverkort, M. Leucker, and J. van de Pol, editors, *Formal Methods: Applications and Technology*, volume 4346 of *Lecture Notes in Computer Science*, pages 51–66. Springer, August 2006.
11. C. A. R. Hoare. *Communcating Sequential Processes*. Prentice Hall, 1985.
12. G. J. Holzmann. The engineering of a model checker: the gnu i-protocol case study revisited. In *Proc. of the 6th Spin Workshop*, volume 1680 of *LNCS*, Toulouse, France, 1999. Springer Verlag.
13. G. J. Holzmann and V. Natarajan. Outline for an operational-semantics definition of PROMELA. Technical report, Bell Laboratories, July 1996.
14. R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, Oct. 1993. *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer, 1993.
15. Z. Qian. A formal specification of java virtual machine instructions for objects, methods and subrountines. In *Formal Syntax and Semantics of Java*, pages 271–312, 1999.
16. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. *SIGSOFT Softw. Eng. Notes*, 28(5):267–276, 2003.
17. M. Rosien. Design and implementation of a systematic state explorer. Msc. thesis, University of Twente, The Netherlands, March 2001.
18. B. Schlich and S. Kowalewski. Model checking c source code for embedded systems. In *Proceedings of the IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA 2005)*, September 2005.
19. B. Schlich, M. Rohrbach, M. Weber, and S. Kowalewski. Model checking software for microcontrollers. Technical Report AIB-2006-11, RWTH Aachen, August 2006.
20. S. Schürmans. Ein Compiler und eine Virtuelle Maschine zur Zustandsraumgenerierung. Diplomarbeit, RWTH Aachen University, Oktober 2005.
21. R. Veldema. Personal communication on the Tapir programming language. `http://www2.informatik.uni-erlangen.de/Forschung/Projekte/Tapir/`, 2006.
22. C. Weise. An incremental formal semantics for PROMELA. In *Proceedings of the 3rd International SPIN Workshop*, April 1997.
23. O. Wibling, J. Parrow, and A. Pears. Automatized verification of ad hoc routing protocols. In *FORTE*, volume 3235 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2004.
24. N. Wirth. Pascal-s: A subset and its implementation. In D. W. Barron, editor, *Pascal - The Language and its Implementation*, pages 199–259. John Wiley, 1981.

A Benchmarks

Our test setup consists of an AMD Athlon 64 3500+ running Linux. We used SPIN 4.2.5 for comparison. SPIN translates PROMELA models into C source code which subsequently is compiled, and then run for the analysis.

By default, SPIN uses data-flow optimizations and statement merging [12] to reduce size of the *explored state space*, thus requiring less time and memory for the task. The optimizations can be disabled optionally (`spin -o1 -o3`). We benchmarked SPIN without said optimizations against our implementation (columns "Unoptimized" in Table 5), and another time with both optimizations enabled, against our *unmodified* VM, but with *path compression* (a variant of statement merging) enabled in our PROMELA compiler.

We compiled the `pan.c` files generated by SPIN from the PROMELA models, and used gcc (version 3.3.5) with option `-O2` (C optimisations), `-DNOREDUCE` (disabling partial-order reduction) and `-DBFS` (enabling breadth-first search). The resulting executable was used for benchmarking.

In our tests, we used models that come with the SPIN distribution. Our experiments show that NIPS (version 1.2.2) is close enough to SPIN both in state vector size (rightmost columns of Table 5) and state space generation speed for our purposes. The actual state count of models is not directly comparable, due to different ways of counting (for example, SPIN counts both halves of a rendezvous communication separately), and due to differing base levels and optimizations. However, crucial behaviour is not optimized away of course.

The size of state vectors, which contain all information needed to restart the virtual machine from (global and local variables, channels, processes), is typically within a few bytes of what is reported by SPIN.

Table 4 shows the results for some large PROMELA models. The experiments were carried out on a 64-bit AMD Opteron^{TM} 248 Dual Processor machine (only one processor used) with 16 GB RAM and a single 200 GB Serial-ATA hard disk, running Linux 2.6.4. For the first two models an arbitrary limit of 2.5 GB RAM was set, whereas the other models were given 16 GB RAM. A full account of the experiments is given in [10].

Model	States		Edges	Time	Uncompressed
	visited	stored		[h]	storage [GB]
GIOP1	192.9M	162.5M	664.6M	13:34:21	79.2
LUNAR 4(d)	1.3G	248.3M	1.9G	35:37:29	153.0
HUGO: Hot fail	555.6M	205.3M	864.9M	15:18:16	166.9
LUNAR 4(f)	1.6G	334.6M	2.6G	38:36:02	230.0

Table 4. Runs for large PROMELA models. *States visited* are all states, including single-successor states, whereas column *States stored* shows only states with more than one successor. M and G denote factors 10^6 resp. 10^9, GB means Gigabyte.

Parameter		NIPS Virtual Machine: Unoptimized States	Time	States/sec.	NIPS Virtual Machine: with Path Compression States	Time	States/sec.	SPIN: Unoptimized States	Time	States/sec.	SPIN: Data-Flow Opt.& Stmt. Merging States	Time	States/sec.	NIPS State size in bytes	SPIN State size in bytes
MAX		**eratosthenes**													
	6	170	0.002	76853.53	34	0.001	51593.32	195	0.016	12187.50	128	0.016	8000.00	130	124
	10	764	0.020	38055.39	74	0.003	26047.17	1006	0.018	55888.89	548	0.018	30444.44	163	156
	14	2744	0.051	53339.55	190	0.006	33009.03	3864	0.026	148615.38	2263	0.026	87038.46	229	220
	18	7766	0.166	46893.02	342	0.012	28023.60	12035	0.058	207500.00	6477	0.058	111672.41	262	252
	22	24092	0.569	42364.48	626	0.025	24584.69	41610	0.344	120959.30	21539	0.344	62613.37	295	284
	26	69920	1.717	40730.78	**1162**	0.054	21361.08	129823	2.430	53425.10	**69618**	0.430	161902.33	328	316
	30	146222	3.824	38237.95	**1710**	0.088	19514.09	282914	11.855	23864.53	**130062**	3.855	33738.52	361	348
	34	**347012**	10.418	33307.78	**2914**	0.177	16451.75	713817	171.441	4163.63	**342028**	26.441	12935.52	394	380
N	L	**leader**													
3	6	754	0.009	79410.22	105	0.002	59965.73	743	0.018	41277.78	407	0.018	22611.11	131	116
4	8	5678	0.082	69216.89	379	0.008	47889.82	5626	0.037	152054.05	2410	0.037	65135.14	186	180
5	10	46091	0.649	70986.55	1509	0.035	43504.58	45937	0.268	171406.72	15791	0.268	58921.64	249	220
6	12	382465	**6.180**	61891.22	**6241**	0.176	35533.72	382151	**3.120**	122484.29	**106449**	0.120	887075.00	320	308
N	L	**leader2**													
3	6	4571	0.054	85268.71	667	0.010	63907.25	4476	0.027	165777.78	2430	0.027	90000.00	138	124
4	8	143373	1.321	108507.81	**10012**	0.161	62102.25	142260	0.650	218861.54	**60052**	0.650	92387.69	193	188
N		**peterson_N**													
	2	327	0.003	109879.03	30	0.000	72289.16	303	0.017	17823.53	185	0.017	10882.35	38	40
	3	51118	0.268	190879.12	**853**	0.012	71112.96	45927	0.085	540317.65	**25371**	0.085	298482.35	50	48
		pftp													
		1378184	**10.033**	137368.71	301603	4.996	60372.40	1275180	**3.770**	338244.03	219167	0.770	284632.47	189	152
		snoopy													
		124434	2.385	52180.87	68658	**1.442**	47603.20	91925	0.436	210837.16	61624	**0.436**	141339.45	205	188
N		**sort**													
	5	21245	0.276	76930.88	572	0.010	54533.32	14349	0.077	186350.65	4652	0.077	60415.58	181	184
	6	152628	1.789	85331.92	**2019**	0.040	49967.83	95677	0.576	166105.90	22350	0.576	38802.08	215	216

Table 5. State Space Generation: A comparison between NIPS and SPIN. PROMELA models are taken from the SPIN distribution. Times are measured as wall-clock time in seconds on an AMD Athlon 64 3500+ running Linux.