Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

An Embeddable Virtual Machine for State Space Generation

Michael Weber[⋆]

Formal Methods and Tools
University of Twente, The Netherlands
e-mail: `michaelw@cs.utwente.nl`

The date of receipt and acceptance will be inserted by the editor

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 around the VM implementation we developed, to evaluate the benefits of the proposed approach.

Key words: state space generation, model checking, virtual machine, operational semantics, PROMELA

1 Introduction

Common approaches in state-based model checking employ high-level modeling languages like CSP [16], LOTOS [5], Murϕ [10], DVE [1], or PROMELA [20] to describe actual state spaces. These languages are usually non-trivial: in addition to the concepts found in programming languages (scopes, variables, expressions) they provide features like process abstraction, non-determinism, timers, guarded commands, synchronisation and communication primitives, etc.. Implementing an operational model of high-level languages for use in verification tools is consequently not straightforward.

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

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 sizable 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 reusable 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 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 few 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 based around NIPS, an implementation of the VM described here.

Organisation. In Sect. 2 we describe the virtual machine model and its byte-code semantics. Sect. 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 tool developers, these tools can be used unchanged to interface with other front-ends, for example, to check C code for embedded systems. Sect. 4 presents benchmark results for our VM implementation to show practical usefulness of our approach. We conclude with a summary of related and future work in Sections 5 and 6.

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 in byte-code for conventional VM architectures like the Java Virtual Machine [25]. 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 execution, the machine offers all possible continuations to the scheduler who then decides which path to take.

Concurrency Processes can be created dynamically. They execute with interleaving semantics.

Communication Both, rendezvous and asynchronous channel objects are provided for inter-process communication. In addition, global variables provide unstructured exchange of information.

First-class channels Like in PROMELA and the π-calculus [23], channels are first-class values, i. e., they can be sent over channels like any other value, thus allowing for a dynamic communication structure.

Priority schemes Our byte-code allows us 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 a sequence does not run to completion (see Sect. 2.4). Such non-deterministic effects are naturally not easily replicable in a conventional VM.

External Scheduling Scheduling decisions are delegated to host applications. This allows for implementation of different scheduling policies which is needed for simulation (interactive scheduling) vs. state space exploration with some search strategy (breadth-first, depth-first, heuristics, interactive, 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 [31]. Our starting point was a simple VM model, which we then extended with features needed for PROMELA's semantics. However, in the interest of reusability we tried to keep these additions as generic as possible (see Sect. 3.4).

In the following, we will present a formalisation of the VM which is suitable for implementation. We found this formalism 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.

Definition 1 (Global State). A tuple $\gamma = \langle \Pi, e, G, \Phi \rangle$ describes the global state of our virtual machine:

$$\gamma \in \Gamma = \mathit{Process} \times \mathit{Pid}_\perp \times \mathit{Mem} \times \mathit{Channels}$$

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

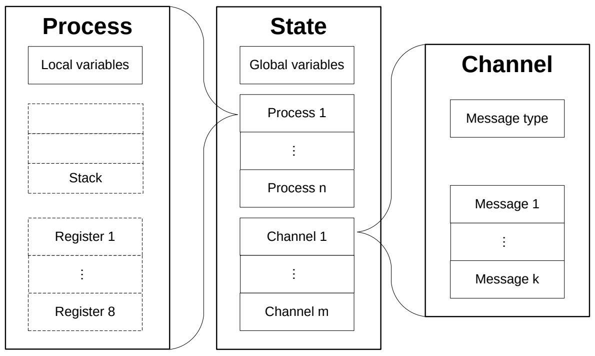

Fig. 1. Overview over the state of the virtual machine. Dotted borders around registers and stack indicate that they are only temporarily part of the machine state, but are not preserved in a state vector.

Definition 2 (Process). A process π is represented as triple $\langle p, M, \Lambda\rangle \in \mathit{Process}$,

$$\mathit{Process} = (\mathit{Pid} \times \mathit{ExecMode} \times \mathit{ProcessState}) \cup \{\mathsf{stop}\}$$

with p denoting a globally unique identifier, its execution mode $M \in \{\underline{\mathrm{N}}, \underline{\mathrm{A}}, \underline{\mathrm{I}}, \underline{\mathrm{T}}\}$ (normal, atomic, invisible, terminated), and Λ' the local state of a process (Def. 3). Furthermore, we allow the special symbol stop to denote a deadlocked process which cannot make any further step.

Execution modes are used to control the observability and scheduling of steps, with $\underline{\mathrm{N}}$ indicating that a process is executing a visible step and willing to yield execution to another process. Mode $\underline{\mathrm{A}}$ and $\underline{\mathrm{I}}$ indicate that a process wishes to continue execution (invisibly in case of $\underline{\mathrm{I}}$). Mode $\underline{\mathrm{T}}$ flags a terminated process.

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 $\gamma \neq \mathsf{stop}$ iff no process in Π is deadlocked: $\forall \pi \in \Pi : \ \pi \neq \mathsf{stop}$.

Definition 3 (Local Process State). An (inactive) local process state $\Lambda_{\mathsf{i}} = \langle L, m\rangle$ is a pair

$$\Lambda_{\mathsf{i}} \in \mathit{ProcessState}_{\mathsf{i}} = \mathit{Mem} \times \mathbb{N}$$

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

When a process becomes active, its state Λ_{i} is augmented with registers R_0 and a stack $D_\varepsilon = \varepsilon$ to its *active local state* $\Lambda_{\mathsf{a}} = \langle L, m, R_0, D_\varepsilon\rangle$:

$$\Lambda_{\mathsf{a}} \in \mathit{ProcessState}_{\mathsf{a}} = \mathit{Mem} \times \mathbb{N} \times \mathit{Registers} \times \mathit{Stack}$$

When Λ_{a} becomes inactive again, its last two components are projected away. Consequently, they can only be used for storing *temporary* values.

We define

$$\mathit{ProcessState} = \mathit{ProcessState}_{\mathsf{i}} \cup \mathit{ProcessState}_{\mathsf{a}}$$

and use Λ to denote a local process state which is either active or inactive.

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 $\sigma[i/v]$. The set *Value* represents the value domain of the VM, and is left unspecified here.

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 do not fit well to a stack model, for instance if values are operated on more than once.

Definition 5 (Data Stack). The evaluation of expressions 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 $D_\varepsilon = \varepsilon$.

2.1.1 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 $c_i \in \mathit{Message} = \mathit{Value}^t$ 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 the outside. Similarly, an asynchronous channel which exceeds its capacity automatically falls back to the same behavior as rendezvous channels: send operations block until they are within their allowed capacity again.

Definition 7 (Rendezvous Communication). We define a predicate sync(γ) 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. This means that it is unnecessary to consider registers and stack as part of a state vector given to a model checking algorithm.

Because the number of local variables is fixed, a (inactive) local process state $\Lambda' \in \mathit{ProcessState}_\mathrm{i}$ then 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 the process level, with only a single process being active at once.

In the spirit of earlier displays of Promela semantics by Holzmann and Natarajan [20], we compose our semantics from several smaller parts by defining four transitions: process activation transitions, internal transitions, intermediate transitions and 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 the transition

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

A process needing exclusive execution privileges *must* be activated, otherwise any process can be activated ($e = \bot$). If its execution mode is deterministic ($M = \underline{\mathrm{D}}$), a process executes its next step deterministically, that is, out of all possible successor states a single one may be chosen and the others discarded. This decision is entirely up to an external scheduler and not modeled within the virtual machine. 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 $\underline{\mathrm{Instr}}$.

2.3.1 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`.

2.3.2 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:

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

Note that, e. g., an insufficient number of values on the stack will simply not result in a transition. More detailed error handling is elided here.

2.3.3 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. For this, we add byte-codes `ELSE` a and its dual `UNLESS` a. They are used in the translation of Promela constructs with similar names, for example (Sect. 3.1.1).

`CALL` a and `RET` can be used to translate function calls. By default the return address is left on the stack, hence it does not survive if a process becomes inactive. It

Table 1. Load and Store byte-codes

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

Table 2. Control-flow byte-codes

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

is in the responsibility of the compiler to produce code which stores it inside the state vector. This allows for some flexibility when dealing with recursive functions. In general, their treatment requires cooperation between the compiler and an analysis tool working with the generated state space.

Promela itself does not allow function calls, so in the translation these byte-codes are used only for sharing common code blocks. However, they have also been used to compile method calls of an object-oriented language [32].

2.3.4 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 [31]. However, we will return to the topic of synchronous communication in Sect. 2.4, when discussing process scheduling.

2.3.5 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:

RUN k, a run a new process starting at address a

$$\langle\{\pi, \pi_1, \ldots, \pi_n\}, e, G, \Phi\rangle \to_{int} \langle\{\pi', \pi_1, \ldots, \pi_n, \pi''\}, e, G, \Phi\rangle$$

with $\pi = \langle p, M, \langle L, m, R, D : v_0 : \cdots : v_{k-1}\rangle\rangle$
and $\pi_i = \langle p_i, M_i, (L_i, m_i)\rangle$ for all $1 \le i \le n$
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

2.3.6 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 a flexible means to control which states become visible to an external scheduler. We expand on the use of *mode* M in Sect. 2.4.

If further execution of a process is not anticipated, 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. Other uses include the translation of receive operations on channels, to deal with the case when a channel is empty.

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', assigning 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:

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

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 (1) a visible transition or (2) a transition leading to deadlock, and no interim rendezvous communication can take place, or (3) 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 the least relation satisfying the following rules.

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

$$\gamma \xrightarrow{p}_{sched} \gamma' \quad \text{if} \quad \gamma = \gamma_1 \xrightarrow{p,\underline{\mathrm{I}}}_{step} \cdots$$
$$\xrightarrow{p,\underline{\mathrm{I}}}_{step} \gamma_{n-1} \xrightarrow{p,M}_{step} \gamma_n = \gamma'$$
$$\text{and } \forall i: \neg\mathrm{sync}(\gamma_i) \text{ and } M \neq \underline{\mathrm{I}} \text{ and } \gamma' \neq \mathrm{stop}$$

2. 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* $\underline{\mathrm{I}}$.

$$\gamma \xrightarrow{p}_{sched} \gamma' \quad \text{if} \quad \gamma = \gamma_1 \xrightarrow{p,\underline{\mathrm{I}}}_{step} \cdots$$
$$\xrightarrow{p,\underline{\mathrm{I}}}_{step} \gamma_{n-1} \xrightarrow{p,-}_{step} \mathrm{stop}$$
$$\text{and } \forall i: \neg\mathrm{sync}(\gamma_i) \text{ and } \gamma' = \gamma_{n-1}$$

3. Lastly, we allow a rendezvous channel to actually contain one message more than its capacity allows, if the immediately following transition resolves the overflow 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.

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

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: $\gamma_{init} = \langle \{\langle 1, \underline{\mathrm{N}}, \langle L_0, \mathit{init} \rangle \rangle\}, \bot, G_0, \emptyset \rangle$

The actual programming interface for state space generation is not described here, as it is largely based on the same principles as Open/CÆSAR [12]. In fact, a preliminary test has shown that we can connect our implementation to the CADP [13] verification suite with little effort, thus leveraging their large set of tools.

Table 3. Operations for Process Deactivation

`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 $\langle L, m, R, D\rangle \rightarrow_{end}$ stop
`NEXZ`	conditional executability shortcut for: `JMPNZ` ℓ; `NEX`; ℓ :

3 Applications

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

Benchmarks of our virtual machine show that it performs well enough to be of practical use (Sect. 4) on its own. In combination with the projects described in the coming sections, we even obtain results which so far have been out of reach.

3.1 Promela

We validated our virtual machine-based approach to state space generation by defining a translation from Promela to byte-code. As a 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 syntax-directed translation scheme is outlined in the following.

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

Separate from this compiler and thus Promela, we developed several common byte-code optimizations for static state space reduction, for example, dead variable reduction and a variant of *statement merging* [26].

3.1.1 Translation to Byte-Code

Covering all details of the translation from Promela to our byte-code would exceed the space limits of this paper. A complete translation procedure has been described by Schürmans [31]. Thus, our aim is to present those constructs which utilize the more exotic instructions of our virtual machine, and demonstrate their rôle in the translation. At the same time, we can formalize some interesting interactions in Promela constructs, e. g., between guards, `goto` and `unless`.

For presentational reasons, we refrain here from repeating a description of the static semantics of Promela, and we omit all details about scoping, as well as variable and label look-up. Furthermore, we restrict ourselves to only the control-flow operators of Promela, as these are the defining elements of any language's "look and feel." For example, it is conceivable to extend Promela with a different expression language or additional data types without changing its appearance much.

Our syntax-directed translation scheme is built up from several functions which take a Promela term as a first parameter: $\mathcal{E}[\![\cdot]\!]$ and $\mathcal{P}[\![\cdot]\!]$ for the translation of *expressions* and *primitives*, respectively (both not further explained here), and $\mathcal{G}[\![\cdot]\!]$ for *guards*, and $\mathcal{S}[\![\cdot]\!]$ for *statements*. In addition, these functions are parametrized with the current environment η (for variable and label look-up), the execution mode $\mu \in \{\underline{\mathrm{N}}, \underline{\mathrm{A}}, \underline{\mathrm{I}}, \underline{\mathrm{T}}\}$ as described in Def. 2, and two blocks of instructions β_1 (for the first statement of a sequence) and β (for any following statements) to be executed before any non-control-flow statement. In particular, the latter class contains language primitives such as variable assignment, send and receive operations, etc..

In Promela, the executability of transitions is determined by *guards*. Thus, if a guard expression *expr* evaluates to *false*, the current execution path is aborted (a semicolon is used to concatenate two byte-code blocks):

$$\mathcal{G}[\![\mathit{expr}]\!]\eta\mu\beta_1\beta := \beta_1; \mathcal{E}[\![\mathit{expr}]\!]\eta; \texttt{NEXZ}$$

If a guard is a sequence of two statements, the first statement is considered in guard position while the second is treated as a normal statement:

$$\mathcal{G}[\![s_1\, ;\, s_2]\!]\eta\mu\beta_1\beta := \mathcal{G}[\![s_1]\!]\eta\mu\beta_1\beta; \texttt{STEP}\ \mu; \mathcal{S}[\![s_2]\!]\eta\mu\beta\beta$$

Note, that β_1 is arranged to be executed before the first statement, while β is used for all other statements. This

[1] http://www.cs.utwente.nl/~michaelw/nips/

special treatment of the actual statement in guard position is later helpful for translating `unless` and `goto`.

A `goto` in guard position is treated specially, as it is always executable, but allows other processes to be scheduled (depending on mode μ) before transferring control. As a control-flow statement, `goto` makes no use of β_1:

$$\mathcal{G}[\![\texttt{goto}\ \ \ell]\!]\eta\mu\beta_1\beta := \texttt{STEP}\ \mu; \mathcal{S}[\![\texttt{goto}\ \ \ell]\!]\eta\mu\beta_1\beta$$

Otherwise, a non-blocking statement is considered as a guard which unconditionally succeeds, unless the statement itself fails:

$$\mathcal{G}[\![\mathit{stmt}]\!]\eta\mu\beta_1\beta := \mathcal{S}[\![\mathit{stmt}]\!]\eta\mu\beta_1\beta$$

Primitive statements, such as send and receive operations can fail (e.g., block on an empty channel). This is handled by $\mathcal{P}[\![\cdot]\!]$. Furthermore, an expression in statement position is considered as guard:

$$\begin{aligned}\mathcal{S}[\![\mathit{prim}]\!]\eta\mu\beta_1\beta &:= \beta_1; \mathcal{P}[\![\mathit{prim}]\!]\eta\\ \mathcal{S}[\![\mathit{expr}]\!]\eta\mu\beta_1\beta &:= \mathcal{G}[\![\mathit{expr}]\!]\eta\mu\beta_1\beta\end{aligned}$$

In both cases, instruction block β_1 is eventually emitted by the translation, in order to deal with exceptional control-flow introduced by PROMELA's `unless` operator:

$$\begin{aligned}\mathcal{S}[\![s_1\ \ \texttt{unless}\ \ s_2]\!]\eta\mu\beta_1\beta\ &:=\ \mathcal{S}[\![s_1]\!]\eta\mu\beta'\beta'\\ &\qquad \texttt{JMP}\ \ell_e\\ &\qquad \ell_2 : \beta_1; \mathcal{G}[\![s_2]\!]\eta\mu\varepsilon\beta\\ &\qquad \ell_e :\end{aligned}$$

with $\beta' := \texttt{UNLESS}\ \ell_2$, and ε the empty block. The effect of this translation is that instruction $\texttt{UNLESS}\ \ell_2$ is emitted in front of every non-control-flow statement of s_1. Thus, the code of s_2 (the "escape sequence"), starting at label ℓ_2, is attempted to execute each time. Execution is resumed after an `UNLESS` instruction only if s_2 fails (via `NEX`) *before* control reaches any `STEP` instruction.

Note, that in case of a nested `unless`, e.g.,

$$\{s_1\ \ \texttt{unless}\ \ s_2\}\ \ \texttt{unless}\ \ s_3$$

the inner escape s_2 is translated such that the outermost escape sequence (starting at label ℓ_3) has priority over the inner:

$$\ell_2 : \texttt{UNLESS}\ \ell_3;\ \mathcal{G}[\![s_2]\!]\eta\mu\varepsilon(\texttt{UNLESS}\ \ell_3)$$

This brings us back to more common control-flow statements:

$$\begin{aligned}\mathcal{S}[\![\texttt{goto}\ \ \ell]\!]\eta\mu\beta_1\beta &:= \texttt{JMP}\ \mathrm{label}(\ell)\eta\\ \mathcal{S}[\![\texttt{break}]\!]\eta\mu\beta_1\beta &:= \texttt{JMP}\ \mathrm{label}(\mathit{break})\eta\end{aligned}$$

A sequence of statements is broken down to its parts, with a `STEP` instruction between each statement:

$$\mathcal{S}[\![s_1\,;\,s_2]\!]\eta\mu\beta_1\beta := \mathcal{S}[\![s_1]\!]\eta\mu\beta_1\beta; \texttt{STEP}\ \mu; \mathcal{S}[\![s_2]\!]\eta\mu\beta\beta$$

Two special sequences change execution mode μ, and thus scheduling and visibility of actions. Deterministic sequences ($\texttt{STEP}\ \underline{\mathrm{I}}$) appear as a single statement which cannot be interrupted, hence the clearing of β:

$$\mathcal{S}[\![\texttt{d\_step}\{\mathit{seq}\}]\!]\eta\mu\beta_1\beta := \mathcal{S}[\![\mathit{seq}]\!]\eta\underline{\mathrm{I}}\beta_1\varepsilon$$

Note, that in SPIN the semantics of `d_step` are not clearly defined ("If non-determinism is present, it is resolved in a fixed and deterministic way, **for instance**, by always selecting the first true guard in every selection and repetition structure." [18, chap. 7]—emphasis ours), and thus we decided to identify resulting states through a predicate function, and defer the decision about their handling to potential host applications of the virtual machine. They are, however, free to follow the above suggestion.

Atomic sequences can be interrupted (e.g., through `unless`), however, one cannot "escape" out of an already deterministic sequence:

$$\mathcal{S}[\![\texttt{atomic}\{\mathit{seq}\}]\!]\eta\mu\beta_1\beta := \mathcal{S}[\![\mathit{seq}]\!]\eta\mu'\beta_1\beta \qquad \text{with } \mu' = \begin{cases}\underline{\mathrm{I}} & \text{if } \mu = \underline{\mathrm{I}}\\ \underline{\mathrm{A}} & \text{otherwise}\end{cases}$$

Finally, non-deterministic choice is handled with the non-deterministic jump instruction `NDET`:[2]

$$\begin{aligned}&\mathcal{S}[\![\texttt{if}\ \ \texttt{::}\mathit{seq}_1\ \ \ldots\ \ \texttt{::}\mathit{seq}_n\ \ \texttt{fi}]\!]\eta\mu\beta_1\beta :=\\ &\qquad\qquad \texttt{NDET}\ \ell_n;\ \ldots;\ \texttt{NDET}\ \ell_2\\ &\qquad \ell_1 : \mathcal{G}[\![\mathit{seq}_1]\!]\eta\mu\beta_1\beta; \texttt{JMP}\ \ell_e\\ &\qquad\qquad \vdots\\ &\qquad \ell_n : \mathcal{G}[\![\mathit{seq}_n]\!]\eta\mu\beta_1\beta; \texttt{JMP}\ \ell_e\\ &\qquad \ell_e :\end{aligned}$$

Similarly, `do`-loops are handled in terms of `if`:

$$\begin{aligned}&\mathcal{S}[\![\texttt{do}\ \ \texttt{::}\mathit{seq}_1\ \ \ldots\texttt{::}\mathit{seq}_n\ \ \texttt{od}]\!]\eta\mu\beta_1\beta :=\\ &\qquad \ell : \mathcal{S}[\![\texttt{if}\ \ \texttt{::}\mathit{seq}_1\ \ \ldots\ \ \texttt{::}\mathit{seq}_n\ \ \texttt{fi}]\!]\eta\mu\beta_1\beta\\ &\qquad\quad \texttt{STEP}\ \mu\\ &\qquad\quad \texttt{JMP}\ \ell\end{aligned}$$

This concludes our description of PROMELA semantics. It also illustrates the amount of details involved in the interaction between different language constructs. In Sect. 3.4, we return to the front-end side by explaining how the translation of a different type of input language, namely C, to byte-code can be tackled.

But first, we show in the coming two sections that the PROMELA front-end enables us to test the viability of new model checking algorithms with real-world test cases.

[2] We elide the treatment of PROMELA's `else` keyword and the slightly different translation if $\mu = \underline{\mathrm{I}}$.

3.2 An External-Memory Model Checker

As main memory is still the most restricting factor in state space generation and model checking of industrial-scale models, a number of papers have been published in recent years which show the viability of *external-memory* model checking methods (e.g., [15,11,3]).

The virtual machine presented here has been used as the state-space generation component in an adaptive external-memory model checking tool [15] 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, the check whether a state has already been visited requires random access to the set of visited states due to commonly used hash-table 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 Sect. 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.

A short summary is given in Sect. 4. Some of the large models, for example Lunar scenario 4(d) [34], have previously been reported as exceeding the capabilities of SPIN with 4 GB RAM, with both partial order reduction and `COLLAPSE` state compression enabled. In contrast, we performed state space generation of the mentioned Lunar scenario with a memory limit of 2.5 GB RAM and without partial order reduction (Sect. 4).

3.3 NIPS and DiVinE

An alternative to the external-memory model checker described in Sect. 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 Brim [8].

The DiVinE library [1] was 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 distributed 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. Heterogeneous 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 hard disk. For example, using DiVinE's distributed algorithms usually gives much faster results. 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 [30, 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 implementation 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 [29].

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),

```
GPRS
  32
SPRS
  _TWBR    "Two-wire Serial interface Bit Rate Register";
  _TWSR    _TWS7 [...] __    _TWPS1 _TWPS0 248;
  _TWAR          _TWA6 [...] _TWA0  _TWGCE 254;
  io _TWDR "Two-wire Serial Interface Data Register"
  i _ADCL  "ADC Data Register Low Byte";
  i _ADCH  "ADC Data Register High Byte";
  [...]
SRAM
  1024
MAPPINGS
  _X 26 27  "X register";
  _Y 28 29  "Y register";
  _Z 30 31  "Z register";
  _SP 93 94 "stack pointer"

ADD Rd, Rr;
  Rd =. Rd + Rr;
  C = Rd(7) & Rr(7) | Rr(7) & !R(7) | !R(7) & Rd(7);
  Z = !R;
  N = R(7);
  V = Rd(7) & Rr(7) & !R(7) | !Rd(7) & !Rr(7) & R(7);
  S = N ^ V;
  H = Rd(3) & Rr(3) | Rr(3) & !R(3) | !R(3) & Rd(3)

BREQ K;
  if (Z == 1) then PC = PC + K + 1
```

Fig. 2. Excerpts from the specification for the ATmega16, defining 32 general purpose registers (GPRs), some special purpose registers (SPRs), 1,024 Bytes RAM, some register-to-memory mappings, the addition instruction and a conditional branch. SPRs which are input or output ports are prefixed with `i` or `o`, and require additional specification. For instructions, the equations describe how the state of the microcontroller is affected in the next state. `R`(n) denotes the nth bit of register R.

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 a single pass. 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 translators from ATmega16/32/324p 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. An example formalization is given in Fig. 2, almost exactly in the syntax of the vendor specification.

It is straightforward to express the effects of ATmega instructions in terms of the byte-code from Sect. 2. However, the critical parts in the formalization of a microcontroller are 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 is quite memory-constrained, typically in the order of 1,024 Bytes. Unsurprisingly, memory allocation is in almost all cases entirely static (no calls to `malloc()`). A limited hardware stack precludes deeply nested 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 separately developed tools described in the previous sections, without extra effort. They are well suited to deal with the potentially large state spaces.

4 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 [17] 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` (which disables partial-order reduction) and `-DBFS` (which enables breadth-first search). The resulting executable was used for benchmarking. BFS is the main strategy in NIPS because of the requirements of distributed algorithms, whereas SPIN's default is depth-first search (DFS). We did not implement optimizations used in SPIN's DFS, which is why we compare BFS only. Note that our VM interprets instructions while `pan.c` is compiled into a native executable.

In our tests, we used models that come with the SPIN distribution. As a separate case study, it would be very interesting to compare the results of SPIN and NIPS with different model-checking backends on the recently published BEEM model database [24], in particular pitting the recently published multi-core versions of SPIN [19] and DIVINE [2] against each other.

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

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.

Model	**States**		**Edges**	**Time**	**Uncompressed**
	Visited	**Stored**		**[h:m:s]**	**Storage [GB]**
GIOP1 [21]	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

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.[3]

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 and the models is given elsewhere [15].

5 Related Work

5.1 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 [20] is incomplete in this sense and now partly outdated, as SPIN has evolved. It was improved on by a more modular and less implementation-specific approach by Weise [33], but there the handling of nested `do` loops in combination with `goto` statements is unsound. Another incomplete attempt is from Bevier [4]. 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 Sect. 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 semantic equivalence except continual testing with publicly available models against SPIN as the reference implementation.

5.2 Virtual Machines

Virtual machines are used extensively in Computer Science. A well-known example is the work of Wirth on the Pascal programming language [35].

Independent to our work, two attempts of virtual machine models for restricted, PROMELA-like languages have been brought to our attention [14,28]. Geldenhuys [14] 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 [9], 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 [28, Sect. 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, the design of Rosien 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. Implementations are not readily available.

Bogor

From the existing model checking frameworks, we found the Bogor framework [27] (part of the BANDERA framework) closest to the work presented here. It is an extensible framework for software model checking, in partic-

[3] An anecdotal example is the `eratosthenes` model (Tab. 5) which, despite differing state counts, still prints out the prime numbers.

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.

		NIPS Virtual Machine				SPIN 4.2.5				NIPS	SPIN
		Unoptimized		with Path Compression		Unoptimized		Data Flow & Stmt. Merge		State size	
Parameter		States	Time	States	Time	States	Time	States	Time	in bytes	
`MAX`		**eratosthenes**									
6		170	0.002	34	0.001	195	0.016	128	0.016	130	124
10		764	0.020	74	0.003	1006	0.018	548	0.018	163	156
14		2744	0.051	190	0.006	3864	0.026	2263	0.026	229	220
18		7766	0.166	342	0.012	12035	0.058	6477	0.058	262	252
22		24092	0.569	626	0.025	41610	0.344	21539	0.344	295	284
26		69920	1.717	**1162**	0.054	129823	2.430	**69618**	0.430	328	316
30		146222	3.824	**1710**	0.088	282914	11.855	**130062**	3.855	361	348
34		**347012**	10.418	**2914**	0.177	713817	171.441	**342028**	26.441	394	380
`N`	`L`	**leader**									
3	6	754	0.009	105	0.002	743	0.018	407	0.018	131	116
4	8	5678	0.082	379	0.008	5626	0.037	2410	0.037	186	180
5	10	46091	0.649	1509	0.035	45937	0.268	15791	0.268	249	220
6	12	382465	**6.180**	**6241**	0.176	382151	**3.120**	**106449**	0.120	320	308
`N`	`L`	**leader2**									
3	6	4571	0.054	667	0.010	4476	0.027	2430	0.027	138	124
4	8	143373	1.321	**10012**	0.161	142260	0.650	**60052**	0.650	193	188
`N`		**peterson_N**									
2		327	0.003	30	0.000	303	0.017	185	0.017	38	40
3		51118	0.268	**853**	0.012	45927	0.085	**25371**	0.085	50	48
		pftp									
		1378184	**10.033**	301603	4.996	1275180	**3.770**	219167	0.770	189	152
		snoopy									
		124434	2.385	68658	**1.442**	91925	0.436	61624	**0.436**	205	188
`N`		**sort**									
5		21245	0.276	572	0.010	14349	0.077	4652	0.077	181	184
6		152628	1.789	**2019**	0.040	95677	0.576	22350	0.576	215	216

ular 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 5,000 lines[4] 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.

Java PathFinder

The Java PathFinder tool (JPF) is a byte-code model checker for Java [7]. In its second generation, JPF interprets Java byte-code via a custom-made Java virtual machine, JVM^{JPF}. Similar approaches with other virtual machines include, e. g., work by Edelkamp et al. [22]. We share their philosophy of model checking executable code directly instead of using a separate model, but chose for our virtual machine a much smaller byte-code language.

In a (re-run of a) case study for the DEOS real-time operating system, developed by Honeywell, the authors chose to (manually) translate C++ source code to Java, which was then compiled to bytecode and checked by JPF. An alternative could have been to compile the DEOS code with a C++ compiler and to translate the resulting machine code to JVM byte-code. We opted for this latter approach within the MCESS project (Sect. 3.4), in which C code is compiled, then disassembled and, instruction by instruction, translated to the byte-code of our NIPS virtual machine. Through the embedding of our VM into a "host" model checking toolkit such as DIVINE, the actual verification takes place.

JPF itself is written in Java, and supports many advanced features which make model checking of Java programs practical, e. g., symmetry reductions, customizable search strategies and symbolic execution. These are all components outside the scope of our VM. Ideally, they could be supplied by third parties, thus keeping our VM small and more easily embeddable into a host application than JPF. The benefit here is in the gained modularity (however, we also anticipate engineering challenges which require further research). In contrast, JPF assumes the rôle of the host application. Thus, new verification algorithms have to be embedded into their virtual machine. This constellation is cumbersome for parallel and distributed algorithms, which often themselves demand a peculiar structuring of code.

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

6 Conclusions

We presented a virtual machine-based approach to state-space generation, in which the virtual machine's instruction set doubles as intermediate model language. Assigning operational semantics in such a way makes them straightforwardly implementable, thus encouraging their reuse. Among the byte-code instructions are all operations commonly needed for the specification of concurrent systems: non-determinism, 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 also 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 available yet.

Also, a command line simulator is available, however, 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 checking, thereby foregoing deviations in results.

Acknowledgements. We thank Michael Rohrbach and Stefan Schürmans for their implementation efforts and valuable discussions. Theo Ruys brought Rosien's work to our attention. We thank the anonymous reviewers for valuable comments and suggestions. Part of the work described here has been carried out at RWTH Aachen University.

References

1. 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.
2. J. Barnat, L. Brim, and P. Rockai. Scalable multi-core LTL model-checking. In Bosnacki and Edelkamp [6], pages 187–203.

3. J. Barnat, L. Brim, P. Simecek, and M. Weber. Revisiting resistance speeds up I/O-efficient LTL model checking. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2008.
4. W. Bevier. Towards an operational semantics of PROMELA in ACL2. In *Proceedings of the 3rd International SPIN Workshop*, April 1997.
5. 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.
6. D. Bosnacki and S. Edelkamp, editors. *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings*, volume 4595 of *Lecture Notes in Computer Science*. Springer, 2007.
7. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - second generation of a Java model checker. In *In Proceedings of the Workshop on Advances in Verification*, 2000.
8. 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.
9. 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.
10. 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.
11. E. A. Emerson and K. S. Namjoshi, editors. *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, volume 3855 of *Lecture Notes in Computer Science*. Springer, 2006.
12. H. Garavel. OPEN/CAESAR: An open software architecture for verification, simulation, and testing. *Lecture Notes in Computer Science*, 1384:68–84, 1998.
13. H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *EASST Newsletter*, 4:13–24, Aug. 2002.
14. J. Geldenhuys. Efficiency issues in the design of a model checker. Msc. thesis, University of Stellenbosch, South Africa, November 1999.
15. 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.
16. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
17. 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 *Lecture Notes in Computer Science*, Toulouse, France, 1999. Springer Verlag.
18. G. J. Holzmann. *The SPIN model checker: primer and reference manual.* Addison-Wesley, Boston, MA 02116, September 2003.
19. G. J. Holzmann and D. Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Trans. Softw. Eng.*, 33(10):659–674, 2007.
20. G. J. Holzmann and V. Natarajan. Outline for an operational-semantics definition of PROMELA. Technical report, Bell Laboratories, July 1996.
21. M. Kamel and S. Leue. Formalization and validation of the general inter-ORB protocol (GIOP) using PROMELA and SPIN. *STTT*, 2(4):394–409, 2000.
22. P. Leven, T. Mehler, and S. Edelkamp. Directed error detection in c++ with the assembly-level model checker StEAM. In S. Graf and L. Mounier, editors, *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 39–56. Springer, 2004.
23. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, June 1999.
24. R. Pelánek. BEEM: Benchmarks for explicit model checkers. In Bosnacki and Edelkamp [6], pages 263–267.
25. 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.
26. G. Quirós. Static byte-code analysis for state space reduction. Master thesis, RWTH Aachen University, March 2006.
27. 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.
28. M. Rosien. Design and implementation of a systematic state explorer. Msc. thesis, University of Twente, The Netherlands, March 2001.
29. 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.
30. B. Schlich, M. Rohrbach, M. Weber, and S. Kowalewski. Model checking software for microcontrollers. Technical Report AIB-2006-11, RWTH Aachen, August 2006.
31. S. Schürmans. Ein Compiler und eine Virtuelle Maschine zur Zustandsraumgenerierung. Diploma thesis, RWTH Aachen University, October 2005.
32. R. Veldema. Personal communication on the Tapir programming language. `http://www2.informatik.uni-erlangen.de/Forschung/Projekte/Tapir/`, 2006.
33. C. Weise. An incremental formal semantics for PROMELA. In *Proceedings of the 3rd International SPIN Workshop*, April 1997.
34. 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.
35. 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.