An embeddable virtual machine for state space generation

Michael Weber

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.

Keywords 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.

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. 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 tool developers, these tools can be used unchanged to interface with other front-ends, for example, to check C code for embedded systems. Section 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 Sects. 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 $\pi$-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) versus 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 Fig. 1 consists of a few global objects and the local state of its processes.