The ASM Workbench

A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models

Tool Demonstration

Giuseppe Del Castillo*

Heinz Nixdorf Institut, Universität Paderborn, Fürstenallee 11,
33102 Paderborn, Germany (*giusp@uni-paderborn.de*)

Abstract. Gurevich's Abstract State Machines (ASMs) constitute a high-level state-based modelling language, which has been used in a wide range of applications. The ASM Workbench is a comprehensive tool environment supporting the development and computer-aided analysis and validation of ASM models. It is based on a typed version of the ASM language, called ASM-SL, and includes features for type-checking, simulation, debugging, and verification of ASM models.

1 Introduction

The ASM Workbench is a comprehensive tool environment supporting the development and computer-aided analysis and validation of Abstract State Machine models. Abstract State Machines (ASMs), defined by Yuri Gurevich in [4], are an effective approach for specifying and modelling state-based systems, which combines transition systems, used for modelling the dynamic aspects of a system, i.e., its behaviour, with first-order structures (algebras), used to model the static aspects, e.g., data types.

The ASM Workbench is based on a language called ASM-SL (ASM-based Specification Language), which extends the ASM language as defined in [4] by a type system and by constructs to define data types and functions (such extensions are very convenient in order to provide tool support for ASMs). The Workbench itself consists of a kernel, providing basic support for ASM tool development, and a set of tools built upon this kernel, which include a type-checker, a simulator, a debugger-like GUI, and a model-checker interface for formal verification support.

In this abstract, after recalling the basic ideas of Abstract State Machines (Sect. 2), we give an overview of the ASM-SL language (Sect. 3) and of the architecture of the ASM Workbench and of the tools it includes (Sect. 4). A complete account of ASM-SL and of the ASM Workbench, as well as of the underlying concepts and techniques, can be found in [1].

* Currently guest scientist at: Siemens AG, ZT SE 4, Otto-Hahn-Ring 6, 81739 München, Germany — E-Mail: *Giuseppe.DelCastillo@mchp.siemens.de*.

T. Margaria and W. Yi (Eds.): TACAS 2001, LNCS 2031, pp. 578–581, 2001.

2 Gurevich's Abstract State Machines

Abstract State Machines (*ASMs*), introduced by Yuri Gurevich in [4], are a high-level state-based language for modelling discrete dynamic systems, which has been used in a wide range of applications, such as specifications of hardware and software architectures and operational semantics of programming languages (see [5] for a comprehensive overview of applications of ASMs).

The underlying computational model is essentially the well-known model of *transition systems*. Computations (*runs*) are finite or infinite sequences of states $\{s_i\}$, obtained from the *initial state* s_0 by repeatedly executing *transitions* δ_i:

$$s_0 \xrightarrow{\delta_1} s_1 \xrightarrow{\delta_2} s_2 \ \ldots \xrightarrow{\delta_n} s_n \ \ldots$$

In the simple case of deterministic ASMs without any communication with an external environment, there will be exactly one run. Otherwise, the set of possible runs can be represented, as usual, by means of a set $S_0 \subseteq S$ of initial states and a transition relation $R \subseteq S \times S$.

The peculiarity of ASMs is that *states* are first-order structures (algebras) over a given *vocabulary* Υ. In the traditional definition of transition systems, states are identified by the value of a finite number of state variables. In ASMs, instead, states are identified by the interpretation of function names from Υ, which are classified as *static*, *dynamic*, and *external*. Each transition may change the interpretation of dynamic function names in a finite number of places.[1] *External* function names are used to model the environment (their interpretation may change from state to state depending on the environment behaviour, like inputs of finite state machines), while *static* function names never change their interpretation (they typically correspond to operations on some data types).

A language of *transition rules* is defined in [4], which allows to specify ASM transitions. The most essential transition rule is the so-called *update rule*, of the form "$f(t_1, \ldots, t_n) := t$", where f is a n-ary dynamic function name, and t_i, t are terms over the vocabulary Υ. This rule has the effect of changing the interpretation of f such that $\mathbf{f}_{s_{i+1}}(s_i(t_1), \ldots, s_i(t_n)) = s_i(t)$. More complex transition rules can be built by means of additional rule constructors, such as conditionals, parallel composition, non-deterministic choice, etc.

3 The ASM-SL Notation

For the purpose of equipping the ASM method with tool support, it is necessary to extend the basic ASM language of [4]. In particular, the ASM definition does not indicate how to define universes and (static) functions, i.e., the data model underlying the transition system. Clearly, there are several options for specifying data, e.g., axiomatic descriptions in the style of algebraic specification. However, in order to obtain executable specifications, a model-based approach

[1] In this sense, ASMs constitute a generalization of transition systems (which can be considered as a special case of ASMs, where all dynamic function names are 0-ary).

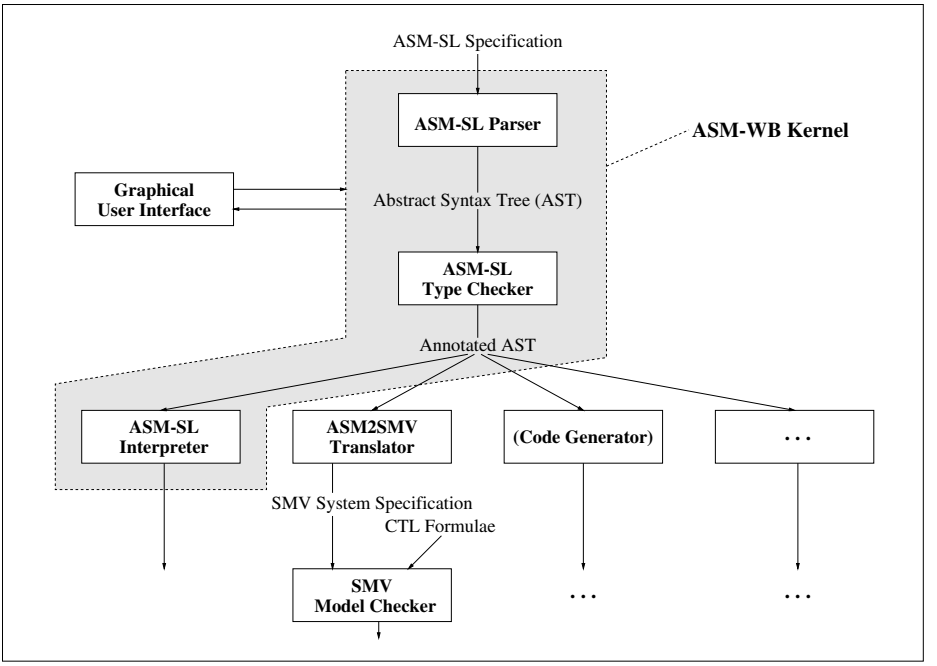

Fig. 1. The ASM Workbench Tool Environment

was adopted, in the style of VDM [6]. A type system was also added (originally, ASMs are untyped). The result is ASM-SL[2], the source language of the ASM Workbench, which extends the ASM language by some constructs borrowed from ML and VDM. The main features of ASM-SL can be summarized as follows:

- Specification of behaviour based on the ASM language of transition rules [4].
- Polymorphic type system based on the type system of Standard ML [8].
- Model-based approach to data specification, including: predefined elementary types (booleans, integers, strings) and type constructors (tuples, lists, finite sets, finite maps), user-definable free types, comprehension notation, pattern matching, recursive and mutually recursive function definitions.

4 The ASM Workbench Tools

The ASM Workbench consists of a *kernel*, a set of modules implemented in the functional language Standard ML [8], which provide basic functionalities (such as parser, type-checker, pretty-printer, and an interpreter-based evaluator), and a few additional components (Graphical User Interface, model-checker interface), built on the top of the kernel. Figure 1 shows the rough architecture of the Workbench, which includes the mentioned tools and could be easily extended by additional components (e.g., code generators), by reusing the kernel

[2] **ASM**-based **S**pecification **L**anguage.

functionalities. For reasons of space, it is not possible to go into details of the tool architecture. Instead, we give an overview of the existing tools.

The *type-checker* for ASM-SL is based on an efficient implementation of the well-known unification-based type inference algorithm [2]. In addition to type-checking, it performs other simple static checks. It can be used as a standalone tool or as a preprocessor for further elaborations.

The *interpreter* allows to simulate ASM runs, while keeping track of the computation history. In this way, computation steps can also be retracted (*backward step* feature). It also possible to simulate ASM models which interact with the environment by means of external functions. This can be done by means of an *oracle process*, which communicates with the interpreter in order to provide it with the values of the external functions, whenever needed.

A *Graphical User Interface* (*GUI*) allows to control the simulation and inspect its results, providing all the typical features of a debugger (browsing through the code, performing single steps forward or backward, setting breakpoints, observing the values of some terms, etc.).[3]

Finally, a *model-checker interface* provides support for formal verification of finite-state ASM models. Although ASM models have, in general, an infinite state space, the ASM-SL language provides a syntactic construct—so-called "finiteness constraints"—by which the finiteness of the model can be enforced by local modifications (restricting the ranges of dynamic and external functions to finite sets). Then, the ASM model is translated, by applying transformation techniques (unfolding and flattening of transition rules), into a model amenable to model-checking. The actual verification is performed by the SMV model-checker [7]. The ASM model is translated into the SMV language and then checked against a set of CTL formulae, to be provided separately (for details, see [3,1]).

References

1. G. Del Castillo. *The ASM Workbench: A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models.* PhD thesis, Universität Paderborn (to appear in 2001).
2. L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. of the 9th ACM Symposium on Principles of Programming Languages*, 1982.
3. G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'2000*, LNCS 1785. Springer, 2000.
4. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.
5. J.K. Huggins. Abstract State Machines home page. EECS Department, University of Michigan. `http://www.eecs.umich.edu/gasm/`.
6. C.B. Jones. *Systematic Software Development using VDM.* Prentice Hall, 1990.
7. K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
8. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[3] A snapshot of the GUI can be found in the appendix.