Debugging Abstract State Machine Specifications: An Extension of CoreASM

Marcel Dausend, Michael Stegmaier and Alexander Raschke

Institute of Software Engineering and Compiler Construction,
University of Ulm, Germany
{marcel.dausend, michael-1.stegmaier, alexander.raschke}@uni-ulm.de

Abstract. We introduce a debugger component as an extension of CoreASM to simplify validation of (complex) ASM specifications. As a basis, we map well-known debugging concepts of imperative programs to the ASM context. The architecture of our debugger is described and some background information to the implementation is given. We conclude by summarizing the current functionalities of our debugger and outlining further development prospects.

Keywords: Abstract State Machines, CoreASM, Debugging

1 Introduction

Creating and editing different kinds of specifications are key tasks that have to be done throughout the system development process. An accepted executable formalism for the specification of hard- and software systems are Abstract State Machines (ASMs) [1]. ASMs have been used to describe, verify and validate complex formal languages, especially their semantics, e. g. Java and its virtual machine [6] or comprehensive parts of the Unified Modeling Language [5].

A major problem of complex specifications is their maintainability and comprehensibility. In case of ASMs, several methodologies and tools have been developed to support defining, editing, validating, and verifying ASM specifications. These tools differ in their support of ASM concepts and focus on specific application issues [1]. One of these tools is CoreASM. Amongst others, it provides a flexible plugin architecture and an interpreter for ASMs [2].

Debugging is a common method to "identify and remove errors from (computer hardware or software)" [1] and to comprehend specifications. Our approach is to extend CoreASM with a debugging component so that multi-agent ASM specifications can be revised more easily.

In Sect. 2, we clarify our notion and capabilities of debugging ASMs and give a brief overview of existing tools and their concepts for debugging of ASMs. We then explain how debugging concepts for imperative programs can be mapped to concepts for debugging ASM specifications. In Sect. 3, we briefly show existing debugging features of CoreASM before we describe our extension of CoreASM. In Sect. 4, we summarize the current status of our extension and outline our next steps and future work.

[1] definition of *debug* from http://oxforddictionaries.com/definition/debug

2 Debugging Abstract State Machines

According to [4], we consider debugging as an interactive process, where a running instance of a program can be stepwise observed and the program execution can be controlled by the user. This observation provides opportunities to comprehend and deeply understand the program and finally revise it, if necessary.

Debugging of ASMs has been addressed formerly by the tools ASMGofer and XASM [1]. Both tools enable you to control an ASM execution by starting, pausing, resuming, and stopping. They offer break conditions to automatically pause an ASM execution. If the execution is paused, both tools allow to investigate the status of ASM functions. CoreASM itself does not support debugging as defined in the previous paragraph, which has been indicated as an open issue [2].

2.1 From debugging of imperative programs towards debugging of ASM specifications

In order to enable fine grained control to debug an ASM execution, an execution **step** has to be defined. Whereas a step in an imperative program means setting the program counter from the current instruction to the following instruction, a step in terms of ASMs means evaluating the machine's program (or Agent programs) and applying the resulting update set to the current state of the ASM. Thus, we use this definition as a debugging step. We do not yet take into account microsteps, which are hidden inside a turbo ASM step (cf. [1], p.174).

A **breakpoint** is a clearly defined point in a program, where the execution stops if this point of the program is reached. We consider different kinds of breakpoints: (line) breakpoints, watchpoints, and method breakpoints.

In an imperative program, a (line) breakpoint is reached if the program counter hits a statement (contained in the line of code) which is marked by a breakpoint. In an ASM, a **(line) breakpoint** is reached, if the marked statement causes an update that is contained within the ASM's update set at the end of the current step. Thereby, it is possible that multiple breakpoints are reached at the same time, which is not possible in an imperative program.

A watchpoint in an imperative program marks a declaration of a variable. The watchpoint is hit if this variable is modified or read in the current step. In an ASM, we define a **watchpoint** as a breakpoint marking either a universe declaration or a function declaration. This includes variable declarations, which are functions of arity zero. The breakpoint is reached if a marked universe or function is changed by any update of the current update set.

Method breakpoints in an imperative program mark the head of a method declaration and are reached if this method is invoked. In ASMs we have macro rules instead of methods, so a **method breakpoint** marks the head of an ASM rule. Instead of stopping the ASM execution when invoking the rule, the breakpoint is reached if at least one statement inside the rule's body causes an update which is contained in the current update set.

Another debugging concept for imperative programs is called "watch expression". A watch expression is a well formed expression of the programming

language. It can be defined as part of the debugging environment so that its current result can be evaluated during the program execution. We define a **watch expression** in ASMs as either a function name or a function name including parameters. The values of all locations of the given function or the value of the given location can be observed at each update step of the ASM.

A **Modification** allows to change a function at a given location when the execution is paused.

3 Architecture and implementation

CoreASM implements different aspects of multi-agent ASM using a flexible plugin based architecture. For example, both, basic ASM and turbo ASM, are implemented as separate plugins.

For the purpose of simple debugging, CoreASM offers the plugin *DebugInfo-Plugin*. It allows adding output statements, which are assigned to user defined channels, to a specification. By configuring a set of channels it can be defined which debug info statements are considered for output during the execution.

Additional information, like the current status of an execution, its selected agents, and the current update, can be displayed on the console, but stepwise execution is not possible.

We enhance debugging functionalities of CoreASM using the Eclipse Debug Project (EDP)[2] to implement the concepts introduced in Sect. 2.1. As a basis for debugging we introduce a stepping mode. This mode forces the interpreter to execute exactly one update step and pause the execution afterwards. The user can toggle between the stepping mode and the running mode.

Since our implementation is based on EDP, it is possible to run ASM specifications in *debug mode* and provide a *debug perspective* with views to manage breakpoints, to inspect and modify variables, and to define and inspect expressions. Furthermore, breakpoints can be set or removed directly within the editor. Entries that have been changed in the current step are highlighted to simplify the inspection of updates. The current number of steps and the currently selected agents are displayed at the top of the variables view by default.

In addition to the EDP views, we provide an update view showing all updates of the current update set for a user defined set of agents. Every entry of the update view provides information about the statement which causes the update, its source file and line number, and its executing agent. Inside the update view, all entries of updates matching a breakpoint are highlighted by a special symbol to ease inspection.

The implementation extends the engine driver to enable control of the CoreASM program executions. The Control API of CoreASM provides the information about the current status of the interpreter [3]. This information is used to update the views of our CoreASM debugger after each step.

An ASM specification running in debug mode considers all types of breakpoints (cf. Sect. 2.1) and automatically pauses if any breakpoint is reached.

[2] `http://www.eclipse.org/eclipse/debug/index.php`

4 Conclusion and future work

This work provides the basis for a systematic investigation of complex ASM specifications and opens opportunities to revise them. Our debugger extends CoreASM mainly by using the EDP (cf. Sect. 3) which is an integral part of the Eclipse environment.

Since "Traditional debugging models [...] do not suite ASMs." [2] we propose an adaptation of imperative debugging concepts for the state machine domain of ASMs (cf. Sect. 2.1). In particular, the user interface provided by EDP was adapted and extended to visualize changes of the state of an ASM execution. A new view provides a list containing all updates of a step and allows direct access to its corresponding ASM statements.

Although the debugger is already helpful, there are plenty of possibilities for further extensions. Some ASM constructs are not yet supported: our current definition of a step neglects sequential steps. Derived functions and local rules cannot be debugged, because they do not cause updates which could be observed via the interpreter. We are working on a way to support these constructs.

Additionally, we plan to introduce a history that enables stepping backwards to compare states of an ASM execution, show their differences, and to trace rule calls of specific agents. Furthermore, a record and replay functionality could be used to eliminate non-determinism (e. g. `choose` or the selection of agents for a specific step) in order to enable debugging of an ASM specification repeatedly under the same conditions.

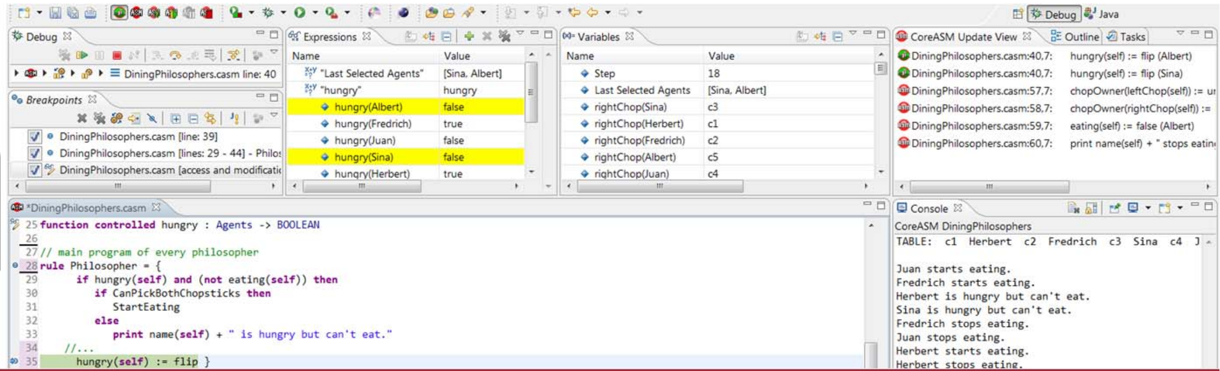
More information about our project and its current status can be found at our website `http://www.uni-ulm.de/en/in/pm/research/projects/coreasm`.

Acknowledgments Thanks to Roozbeh Farahbod for answering numerous questions, trying out the tool, and suggesting some further improvements.

References

1. E. Börger and R. Stärk. *Abstract State Machines – A Method for High-Level System Design and Analysis.* Springer, 2003.
2. R. Farahbod. *CoreASM: An Extensible Modeling Framework & Tool Environment for High-level Design and Analysis of Distributed Systems.* PhD thesis, Simon Fraser University, Burnaby, Canada, 2009.
3. R. Farahbod, V. Gervasi, U. Glässer, and G. Ma. CoreASM Plug-In Architecture. In *Rigorous Methods for Software Construction and Analysis*, pages 147–169, 2009.
4. J. Henkel and A. Diwan. A Tool for Writing and Debugging Algebraic Specifications. In *Proceedings. of the 26th ICSE 2004*, pages 449–458, 2004.
5. J. Kohlmeyer and W. Guttmann. Unifying the Semantics of UML 2 State, Activity and Interaction Diagrams. *Perspectives of Systems Informatics*, 5947:206–217, 2010.
6. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation.* Springer, 2001.

ulm university universität uulm



Debugging Abstract State Machine Specifications

An Extension of CoreASM

Introduction

Although debugging is an integral part of the implementation of software, it is just roughly supported by current Abstract State Machine (ASM) tools.

In order to simplify the validation of (complex) ASM specifications we extend CoreASM [Farahbod2009] by a debugger.

Functionalities

- the control mode "stepping" for the interpreter
- capabilities to debug CoreASM programs based on the Eclipse Debug Project (EDP)
  - line breakpoints
  - watchpoints
  - rule breakpoints (cf. method breakpoints)
  - watch expressions
  - variables view
  - expressions view
  - breakpoint view
- extensions that go beyond EDP
  - updates view
  - agent filter for updates view

Fig. 1 The stepping mode button of the CoreASM control bar

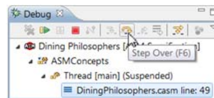


Fig. 2 The standard debug control of Eclipse

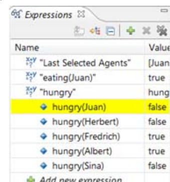


Fig. 5 The variables view presenting the current state of the CoreASM execution

Architecture and Implementation

Architecture and implementation of our debugging component are based on the EDP as a basis for user defined integrated Eclipse debuggers.

The Control API of CoreASM provides information about the current status of the interpreter [Farahbod, Gervasi et al. 2004].

We prepare the conceptual basis for the implementation of the debugger through the transfer of concepts of debugging of imperative programs to concepts of debugging ASM specifications.

Therefore, we characterize the following debugging concepts in terms of ASM: a debugging *step*, a *line breakpoint*, a *watchpoint*, a *method breakpoint*, a *watch expression*, and *modification of data*.

An Example of Using the CoreASM Debugger

As an example, we debug a slightly modified version of the CoreASM sample specification "Dining Philosophers".
A debug execution of CoreASM can be controlled either by using the *extended CoreASM controls* (Fig. 1) or by using the standard eclipse *debug control* (Fig. 2).

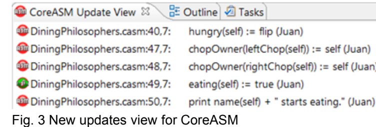


Fig. 3 New updates view for CoreASM

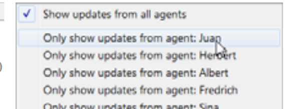


Fig. 4 Agent filter menu of the updates view

The updates of the last execution step are presented within the *update view* (see Fig. 3). Green *highlighted entries indicate* an update which is currently hit by a breakpoint.
A *filter* can be used to focus on a specific agent (see Fig. 4).

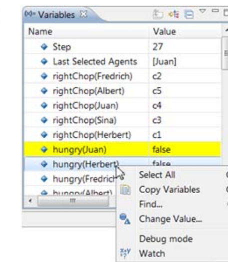


Fig. 6 The expressions view of Eclipse

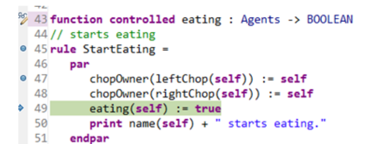


Fig. 7 Editor component with indicators for corresponding breakpoints and the last update of the current update set

The *variables view* (Fig. 5) and the *expressions view* (Fig. 6) can be used to examine the current state of the CoreASM execution.

The main component of CoreASM is the *editor* (Fig. 7).

- A *watchpoint* (see Fig. 7, l. 43) interrupts the interpretation if the marked function at any given location has been changed during the current execution step.
- A *method breakpoint* (rule breakpoint; Fig. 7, l. 45) is hit if any update is caused by any statement within the rules' body.
- A *line breakpoint* (Fig. 7, l. 47) causes the interpreter to pause if a statement of the marked line triggers an update within the current update set.
- The line containing the last update in the current update set is marked by an *indicator* (blue arrow; see Fig. 7, l. 49).

References

[Farahbod 2009] R. Farahbod. *CoreASM: An Extensible Modeling Framework & Tool Environment for High-level Design and Analysis of Distributed Systems*. PhD thesis, Simon Fraser University, Burnaby, Canada, 2009.

[Farahbod, Gervasi et al. 2004] R. Farahbod, V. Gervasi, U. Glässer, and G. Ma. *CoreASM Plug-In Architecture*. In Rigorous Methods for Software Construction and Analysis, pages 147-169, 2009.

Contact

Marcel Dausend marcel.dausend@uni-ulm.de
Alexander Raschke alexander.raschke@uni-ulm.de
Michael Stegmaier michael-1.stegmaier@uni-ulm.de

Project Site
http://www.uni-ulm.de/en/in/pm/research/projects/coreasm/