WebASM: An Abstract State Machine Execution Environment for the Web

Simone Zenzaro, Vincenzo Gervasi, and Jacopo Soldani

Dipartimento di Informatica, University of Pisa, Italy

Abstract. We describe WebASM, a web-based environment that embeds the CoreASM execution engine in a web page. WebASM provides several advantages to specification writers: (1) complex behaviour expressed via ASM can be made visible by using the full power of the web-based presentation layer; (2) ASM specifications can be edited and run interactively via any web browser; (3) the full CoreASM environment is made available via zero-install deployment, thus eliminating a major barrier to the adoption of the language.

In this paper, we briefly outline the technicalities of the approach, present an example, and survey possible applications of WebASM.

1 Introduction

Abstract State Machines (ASM) [2] have been demonstrated to be a powerful yet intuitive formalism for describing specifications. A vast number of case studies, including language specifications, microprocessor design, sequential and distributed algorithms, and industrial plant control machines (see [1] for a full survey) have established the practical applicability of ASMs to real-world systems.

In the 30 years history of the ASM method, a number of execution environments have been developed; among the major efforts, we cite [9,8,4,6]. In varying degrees, all these approaches required setting up a moderately complicated programming environment (e.g., using a Gofer interpreter in [9], or a .NET development environment for [8], or using the Eclipse IDE for [4]). Moreover, none of the existing environments are endowed with convenient graphics facilities (although some of them, e.g. AsmL and CoreASM, can make recourse to native calls to platform-specific graphic APIs).

With WebASM, we set to improve on those two aspects by providing a web-based, fully self-contained embodiment of the CoreASM execution environment [4] which can be run in any modern web browser, and that can be controlled via JavaScript so that arbitrarily complex user interfaces and graphical displays can be rendered as a (dynamic) HTML page.

In the following, we first describe the technical approach taken by WebASM; we then present an example, describing the graphical animation of a distributed leader election protocol specified in ASM. A discussion about possible applications of WebASM and some reflections on future work conclude the paper.

Y. Ait Ameur and K.-D. Schewe (Eds.): ABZ 2014, LNCS 8477, pp. 216–221, 2014.
© Springer-Verlag Berlin Heidelberg 2014

2 Technical Outline

One of the premises of the CoreASM project was that the resulting execution engine should be easy to embed in other applications. In WebASM, we made good on that promise by embedding the whole CoreASM engine (including all the plugins packaged in the official distribution) in a Java applet, which is then connected to the hosting web page through JavaScript bindings.

Security policies restrict what applets can do: in particular, by default no access to the local filesystem is possible (and special configurations are undesirable in a zero-install perspective). As a consequence, dynamic addition of user-developed plugins is not allowed in WebASM. Saving and loading specifications, instead, is managed on the JavaScript side by treating the specification as a string and passing it to the engine for interpretation. In a typical application, the specification text could be obtained from a text editor hosted on the same page, thus allowing the user to write and run ASM specifications in the same environment.

The JavaScript bindings exposed by WebASM include methods to create and initialise ASM machines, to load specifications, to perform an ASM step, and to access the whole abstract state of the machine or a single location.

In particular, access to the abstract state is limited by the concrete representation of values. Indeed, in its full generality the ASM model allows for arbitrary sorts, including those whose values do not have a literal notation. The JavaScript bindings for WebASM allow reading any value in the ASM state (technically, any CoreASM `Element` instance) as a string; the converse is not always possible (e.g., an element of *Agent* can be printed as a name, but not re-created from its name alone). However, all basic types which are commonly used (e.g., strings and numbers) are fully mapped between ASM and JavaScript.

The final element in our implementation is a map between locations of the ASM state and attributes of DOM elements in the page, optionally transformed by a custom JavaScript function (to account for syntactic differences between ASM and HTML/CSS notations).

What is left to the user is to design an HTML page with suitable graphics to visualise the salient elements of the ASM state, and define a map, as described above, in order to visualise state evolution during the ASM computation. After each ASM step, locations of the ASM state mentioned in the map are read (through the JavaScript bindings), their values are mapped or transformed, and finally applied to attributes of the various corresponding DOM elements, thus updating the DOM state. After each update cycle, the browser re-renders the modified portions of the web page (namely: the part hosting the graphical depiction of the ASM state), and the engine is then ready to execute the next step.

Designing suitable HTML graphics for the desired representation of the ASM state can be a tricky at times, depending on how sophisticated the depiction is. However, HTML design and JavaScript programming skills are much more readily available than what would be required to produce a custom, full-blown application to the same end. Moreover, it is reasonable to assume that in a

context where the goal is to teach formal modeling skills, programming skills are already available, so we do not expect this part of the approach to be a significant burden.

In extreme cases, our technique can be extended to manipulate elements of arbitrary SVG vector graphics embedded in a web page, instead of DOM elements, again by using simple JavaScript mappings. We had no need of such an extension in our experiences with the tool.

3 Examples

To exemplify the WebASM approach, we use a specification for a classical distributed algorithm, namely the Extrema Finding by Franklin [5]. In this problem, a number of processes (modelled in ASM as separate agents) are arranged in a ring topology with bidirectional communications, with each process holding a value; their task is to identify the process holding the maximal value.

The corresponding ASM specification, which is a straightforward translation of the algorithm provided in [5], is shown in Figure 1.

```
ExtremaFinding =
  if mode(self) = ACTIVE then
    if not isLargest(self) then
      rightMsg(l(self)) := id(self)
      leftMsg(r(self)) := id(self)
    if largerMsgReceived then
      mode(self) := INACTIVE
    if myMsgReceived then
      isLargest(self) := true
      notified(r(self)) := true
  if mode(self) = INACTIVE then
    if notified(self) then
      notified(r(self)) := true
    else
      rightMsg(l(self)) := rightMsg(self)
      leftMsg(r(self)) := leftMsg(self)
  if isLargest(self)
    and notified(self) then
      ExtremaFound
```

Fig. 1. The main rule in ExtremaFinding (signature can be seen in Figure 2)

As for the visualisation, we have chosen to show each process (hence, each ASM agent) as a box displaying three figures: at the top, the process' value (*id*(*self*)); on the bottom left and the right the values received from its left and right neighbour according to the ring topology (*leftMsg*(*self*) and *rightMsg*(*self*)). Moreover, a box's border colour indicates the process state (*mode*(*self*), with *ACTIVE* → green and *INACTIVE* → red), and a box's background colour indicates whether the process has been notified (*notified*(*self*), *false* → white and *true* → grey). Finally, the border style indicates the value of *isLargest*(*self*) (*true* → dashed, *false* → solid).

The user can then experiment running (and modifying) the specification, in a continuous way or step-by-step, while observing its progress through the animation happening on the page. Figure 2 shows the browser-hosted animation environment.

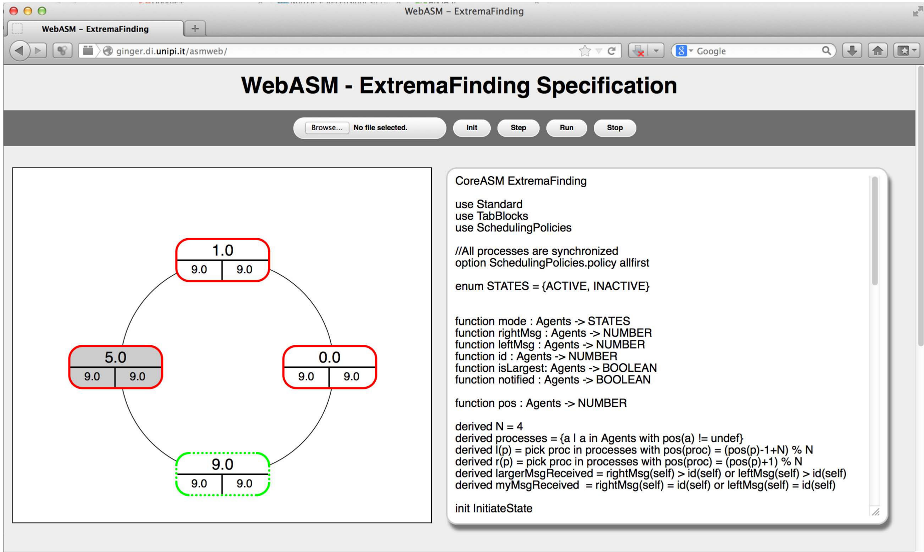

Fig. 2. A screenshot of WebASM animating the Extrema Finding specification

As an additional example, Figure 3 shows three subsequent stages of animation for another classical specification, based on the Distributed Termination Protocol from [3] (the corresponding ASM specification has been published in [7]). Here, each box represents an agent (simulating a different machine), with each machine spontaneously exchanging messages with others. At each step, exactly one machine holds a coloured token (which can be black or white), that is passed around as the protocol progresses. The goal is to determine whether the entire distributed computation has finished, which is detected when a "white" token is returned to the master machine (depicted with a grey background).

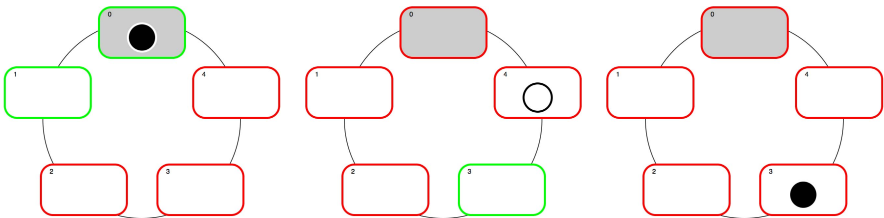

Fig. 3. Three steps of the Distributed Termination Protocol specification

4 Applications

WebASM offers a fully self-contained, zero-install environment for executing and animating CoreASM specifications. By self-contained we mean that the whole environment is contained in a single web page – there is no need of a web server, although if so desired one can be used to serve the page remotely. By zero-install we mean that there is no need of any special software on the client computer: a standard web browser (capable of executing Java applets) suffices.

Both these features, united to the convenience of HTML/CSS rendering capabilities, make WebASM ideally suited to occasional users and ASM newcomers, as they greatly reduce technical barriers to entry.

WebASM is in particular suited for teaching. In teaching algorithms, it provides a double advantage due to the pseudocode-over-abstract data syntax of ASM, and to the positive reinforcement obtained by showing the algorithm's progress via graphical means. In teaching formal modelling, WebASM allows for experimenting with specifications, which can be modified and animated (and thus, tested) interactively.

Also in a teaching context, WebASM can be used to prepare exercises, where the signature and graphic visualisation for a certain problem are given by the instructor, and the task set on students is to write ASM rules to accomplish the desired behaviour. One could envision an entire course based on a number of web pages, each allowing students to experiment with different specifications.

Finally, WebASM provides an alternative to traditional scripting languages for the web. Instead of programming some desired behaviour in languages such as JavaScript or VBScript, a developer could provide a ground model as a CoreASM specification, and have it executed behind the scenes by WebASM. In such a setup, conformance of the implementation to its ASM specification would be guaranteed by construction.

5 Conclusions and Future Work

We have presented WebASM, a self-contained, zero-install, interactive, graphical execution engine for CoreASM specifications which can be run entirely in any standard web browser.

A set of APIs allow accessing the ASM state and controlling the ASM computation from JavaScript code, thus enabling interactive, graphical visualisation of the progress of the computation. The resulting environment is well suited to quick experimentation with specifications and algorithms.

As a work in progress, WebASM can be extended in several directions; the most promising of which are (1) providing a graphical highlight of the rules being executed at each step, in addition to visualising the state resulting from their execution, (2) providing a better editor, with support for syntax highlighting and code completion, and (3) empowering users to visually build graphical representations of the state, by providing a palette of tools to draw graphical elements and link their appearance to elements of the ASM state.

Some of these improvements, and especially (3), we consider as crucial to the full realization of the promises of WebASM. Currently, a modicum of web programming prowess is required to define the graphical representation of the state and the mapping between locations of the ASM state and the DOM elements depicting them. Ideally, the mapping could be defined – at least in most standard cases – by a simple "property sheet"-style editing interface.

In a teaching setting, (3) is geared towards the instructor preparing exercises (on a given problem) for students. In contrast, (1) and (2) are geared towards the students, in that improvements in these areas would directly lead to more effective feedback and ease of experimentation with changing the ASM specification for the given problem. We are currently in the process of implementing (1) and (2) for the first public release of the tool.

References

1. Börger, E.: The origins and the development of the ASM method for high level system design and analysis. Journal of Universal Computer Science 8(1), 2–74 (2002)
2. Börger, E., Stärk, R.: Abstract State Machines – A Method for High-Level System Design and Analysis. Springer (2003)
3. Dijkstra, E., Feijen, W., van Gasteren, A.: Derivation of a termination detection algorithm for distributed computations. Information Processing Letters 16, 217–219 (1983)
4. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An extensible ASM execution engine. Fundamenta Informaticae 77, 71–103 (2007)
5. Franklin, R.: On an improved algorithm for decentralized extrema finding in circular configurations of processors. Commun. ACM 25(5), 336–337 (1982)
6. Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for abstract state machines. Journal of Universal Computer Science 14(12), 1949–1983 (2008)
7. Gervasi, V., Riccobene, E.: From English to ASM: On the process of deriving a formal specification from a natural language one. Dagstuhl Reports 3(9), 85–90 (2014)
8. Gurevich, Y., Rossman, B., Schulte, W.: Semantic essence of AsmL. Theor. Comput. Sci. 343(3), 370–412 (2005)
9. Schmid, J.: Refinement and Implementation Techniques for Abstract State Machines. PhD thesis, University of Ulm, Germany (2002)