ENGINEERING CONCURRENT AND REACTIVE SYSTEMS WITH DISTRIBUTED REAL-TIME ABSTRACT STATE MACHINES

Bridging the gap between formal and empirical approaches

Uwe Glässer and Mona Vajihollahi

*School of Computing Science, Simon Fraser University, Burnaby, BC, Canada V5A 1S6
{glaesser/mvajihol@cs.sfu.ca}*

Abstract: This paper revisits the distributed real-time abstract state machine (ASM) paradigm as a feasible, yet robust, approach to high-level specification and design of distributed embedded systems. The flexibility in modeling reactive system behavior, the well defined underlying concurrency framework, and the ability to gradually sharpen requirements into specifications, inspires viewing the ASM paradigm as an *agile* formalization method that directly supports fundamental practical needs in modeling such systems.

Key words: distributed embedded systems, requirements specification, agile formalization.

1. INTRODUCTION

Distributed embedded systems implicate decentralized control structures and asynchronous communication protocols. As such, they are characterized by their concurrent and reactive behavior, making it particularly difficult to predict dynamic system properties with a sufficient degree of detail and precision under all circumstances. Typical examples are embedded control systems and distributed protocol architectures used in automotive control, industrial automation, e-business applications and wireless communication. The inherent complexity and intricate nature of such systems clearly demand for reliable and predictable design approaches in order to establish the key system attributes in early design phases prior to coding. Unfortunately, it is common practice, even when dealing with complex concurrent and reactive

behavior, to deeply rely on informal requirements—regardless of the lessons that should have been learned from previous experiences:

“There is a strong tendency on the part of people in general, and engineers in particular, to pass quickly and casually through the problem-definition phase of system design and focus almost immediately on solutions. A satisfactory solution to the right problem is often better than an excellent solution to the wrong problem.”[6]

Classical engineering disciplines, in contrast, widely use well established formalisms (such as the circuit diagrams and blueprints in electrical and mechanical engineering) for gaining the mathematical precision needed to sharpen informal requirements into reliable specifications. Inevitably, the evolution of systems design will eventually lead to a shift in the design culture towards mathematically well founded approaches. Not yet clear is *how* formalization can effectively be used as a design instrument that serves the fundamental practical needs.

The goal of this paper is to critically review the abstract state machine (ASM) paradigm [14], [4] regarding its potential for high level specification and validation of concurrent and reactive systems. Specifically, we discuss fundamental issues related to suitability, practicability and robustness of the approach. How can one gradually sharpen requirements into specifications? How to draw the boundary between formal and informal specifications, and how can they complement each other?

ASM models have been used extensively for specification and reverse engineering of hardware and software architectures (e.g. [11],[7],[12],[1]), semantic foundations of system modeling languages (including SDL [9], SystemC [16] and VHDL [5]) and programming languages (like Java [17]), and a variety of protocols (including real-time aspects [15]). A computation model that is frequently used for concurrent and reactive systems is the *distributed ASM* (DASM) model, a generalization of all other ASM models. Practical applications dealing with distributed embedded systems often use a restricted form obtained by imposing additional real-time constraints [9], [12],[1]. In this paper, we concentrate on the DASM model and its variant, the real-time DASM model.

Section 2 discusses the idea of agile formalization. Section 3 illustrates the semantics of concurrency, reactivity and time. Section 4 exemplifies the embedding of an ASM into its environment. Section 5 concludes the paper.

2. AGILE FORMALIZATION

This section briefly illuminates some fundamental aspects of combining a formal semantics framework with informally described requirements.

Informal descriptions are useful and necessary for dealing with abstract requirements in early design phases. However, they are usually ambiguous, incomplete, and often even inconsistent. Furthermore, informal descriptions are not *executable* and thus provide very limited support for experimental validation of the correctness and completeness of requirements. To eliminate such deficiencies, one would like to gradually sharpen informal requirements into precise specifications (effectively turning “English” into mathematics) with a degree of detail and precision as needed.

Formal methods indeed seem to work best at the requirements level [2]. However, formalization may not help us to identify missing requirements, and *“difficulties caused by lack of understanding of the real world situation are not eliminated by use of FMs; instead the misunderstanding gets formalized into the specifications, and may even be harder to recognize simply because formal definitions are harder to read by the clients”* [2]. Still, formalization prior to coding results in a much better understanding of the requirements in early design phases, where it is most meaningful, and thus can indeed improve the quality of the resulting software dramatically.

Viewing the ASM paradigm in the light of *agile* software development [8], the very nature of this formalism clearly is that of a “light-weight” formal method (in the spirit of agile methods). Oriented towards practical system design needs, the ASM formalism emphasizes feasibility, flexibility and robustness instead of heavy mathematical machinery. As such, it offers a good compromise between mathematical elegance and practical relevance. This is accomplished by combining a universal abstraction for representing system states with a minimal state transition language. Since there is no a priori fixed language for defining the initial states of an ASM, one may use whatever language is appropriate (e.g., first-order logic, algebraic languages, etc.). Conceptually, this even allows for combining formal and informal notations in the form of literate specifications where this is appropriate.

The idea of ‘agile formalization’ materializes in the notion of *ASM ground model* as discussed in [3],[4]. Particularly interesting for dealing with concurrent and reactive behavior are the underlying abstraction mechanisms for modeling the continuous interaction between a system and the *operational environment* into which the system is embedded—especially as one usually starts from a fuzzy understanding of how actions and events in this environment actually affect the system behavior.

3. CONCURRENCY, REACTIVITY, TIME

Abstract state machines (originally called *evolving algebras*) have been introduced by Yuri Gurevich in [14]. We assume here some familiarity with

the basic ASM concepts and will recall only definitions that we explicitly address in this paper. For further details on theoretical foundations, see [13].

3.1 Concurrent Computations

A DASM A is defined over a given vocabulary V by a program Π_A and a non-empty set I_A of initial states. An initial state of A specifies a valid interpretation of V over some potentially infinite base set X . A consists of a collection of autonomously operating *agents* from some finite domain $AGENT$. Intuitively, the agents of A model the concurrent control threads in distributed computations of A . Agents interact with each other by reading and writing shared locations of global machine states. The underlying semantic model regulates such interactions so that potential conflicts are resolved according to the definition of *partially ordered runs*.

A (partially ordered) *run* ρ of a distributed ASM A is given by a triple (M, λ, σ) satisfying all of the following four conditions [14]:

1. M is a partially ordered set of moves where each move has only finitely many predecessors.
2. λ is a function on M associating agents with moves such that the moves of any single agent of A are linearly ordered.
3. σ assigns a state of A to each initial segment Y of M , where $\alpha(Y)$ is the result of performing all moves in Y ; $\alpha(Y)$ is an initial state if Y is empty.
4. The *coherence condition*: If x is a maximal element in a finite initial segment X of M and $Y = X - \{x\}$ then $\lambda(x)$ is an agent in $\alpha(Y)$ and $\alpha(X)$ is obtained from $\alpha(Y)$ by firing $\lambda(x)$ at $\alpha(Y)$.

While the above definition is elegant and concise, it may not be obvious how one can imagine DASM runs in concrete applications, and, particularly, what does the coherent constraint mean?

Intuitively, a run specifies a class of possible executions of A . In general, there is more than one partial order required to specify all the possible executions of A . For a given partial order ρ , there may be more than one (even infinitely many) actual executions of the underlying DASM model that can be derived from ρ . Of particular interest among these are the *linearly* ordered runs, each of which is the result of a linear transformation applied on ρ (effectively ordering all those moves that are incomparable under ρ in an arbitrary way). In that respect, there are two interesting observations [14]:

- All *linearizations* of the same finite initial segment of ρ have the same final state.
- A property holds in every reachable state of a run ρ if and only if it holds in every reachable state of every linearization of ρ .

To further illustrate the meaning of the coherence condition in the above definition, Section 3.2 considers two simple but meaningful examples.

3.2 Sample DASM models

We illustrate here fundamental semantic aspects of partially ordered runs by means of two sample DASMs, where the first one is derived from [10].

Example 1. Suppose that we have three propositional variables (dynamic nullary relation symbols) *door*, *window* and *light*. Intuitively *door* = *true* means that “the door is open”, *window* = *true* means that “the window is open” and *light* = *true* means that “the light is on”. Now, consider a DASM consisting of three agents: a *door manager* (agent *d*), a *window manager* (agent *w*) and a *light manager* (agent *l*). The door manager opens the door only when the window is closed (move *x*), the window manager opens the window only when the door is closed (move *y*), and the light manager turns on the light when either the door or the window is closed (move *z*).

WindowManagerProgram $\equiv$ **if** $\neg$ door **then** window := true
 DoorManagerProgram $\equiv$ **if** $\neg$ window **then** door := true
 LightManagerProgram $\equiv$ **if** $\neg$ door **or** $\neg$ window **then** light := true

Figure 1 shows all the possible DASM runs assuming that in the initial state S_0 the door and the window are closed and the light is turned off. There are six possible runs (M_1 – M_6), yielding to two different final states (S_4 , S_5).

$M_1 = (\{x, z\}, \langle \rangle)$, $M_2 = (\{x, z\}, \langle x < z \rangle)$, $M_3 = (\{x, z\}, \langle z < x \rangle)$,
 $M_4 = (\{y, z\}, \langle \rangle)$, $M_5 = (\{y, z\}, \langle y < z \rangle)$, $M_6 = (\{y, z\}, \langle z < y \rangle)$.

We cannot have $x < y$ because *w* is disabled in the state S_1 obtained from S_0 by performing *x*. Similarly we cannot have $y < x$ because *d* is disabled in the state S_3 obtained from S_0 by performing *y*. Finally, we also cannot have a run where *x* and *y* are incomparable, that is neither $x < y$ nor $y < x$. This follows from the fact that all the linearization of such a run must result in the same state (thus it is impossible to go from S_0 to S_6 or S_7 , or from S_2 to S_7).

Example 2. Suppose a single *producer* agent placing items, one by one, into a queue. Two *consumer* agents concurrently attempt to remove these items by popping the head of the queue. This example shows the effect of the coherence condition in the presence of a race condition (between the two consumers simultaneously trying to remove the same item of the queue).

We abstract from the details of adding items to the queue and removing items from it. In each step of the producer, it adds a single new item to the queue (move *p*). In each step of a consumer, it removes the head item if the queue is nonempty (moves *c1*, *c2*). The programs of the producer agent and the consumer agents can be written as follows.

ProducerProgram $\equiv$ ADD_ITEM(queue, newItem)
 ConsumerProgram $\equiv$ **if** queue $\neq$ empty **then** item := headItem(queue)

Figure 1. All possible runs of Example 1

Figure 2. Some segment of possible runs of Example 2

In the initial state the queue is empty. The most important property of this DASM is that there is no run where $c1$ and $c2$ are incomparable. Note that if both consumers would make an attempt to remove the same head item at the same time (incomparable $c1$ and $c2$), this would not cause conflicting update operations on the queue; rather it would produce a logical conflict

(notably, a duplication of this item). The coherence condition prohibits this behavior as any linear execution of such a run, for instance $c1 < c2$, can not produce the same result. *Figure 2* shows some segment of possible runs of this DASM and helps clarifying this argument. Clearly, it is not possible to go from states S_2 and S_3 to S_4 ; hence $c1$ and $c2$ are not incomparable.

3.3 Reactivity and Time

Externally controlled actions and events that may affect the system behavior in any way are encapsulated in abstract interface functions, called *monitored functions*. Such functions change their values dynamically under control of the system environment and, by definition, are read-only within the model. A typical example is a real time extension of the DASM model¹ based on a notion of *global system time* as implied by a globally accessible system clock. One can model such a clock by a nullary function *now* taking values in a linearly ordered domain *TIME*. As an integrity constraint on *now*, we assume that the values of *now* change monotonically over DASM runs.

monitored *now*: *TIME*

Conceptually, monitored functions formalize abstract interfaces making the boundary between the system and its environment explicit and visible. Also, they have a crucial role in shaping abstraction levels by allowing us to move irrelevant details to the “background”. In practice, however, it is often not obvious how to delineate the system from the environment and the background as this problem is tightly linked to the question of how to find the “right” level of abstraction. We try to give some answers below.

4. AN ABSTRACT CASH MACHINE CONTROL

In this section, we abstractly model an automated transaction machine, or ATM, to illustrate the role of ground models in defining the embedding of a system into a given operational environment.

4.1 Informal requirements for ATM transactions

Assume an asynchronous interaction model between three autonomously operating entities involved in ATM transactions, namely: an *ATM manager*, an *authentication manager*, and an *account manager*. For simplicity, we restrict here on withdrawal transactions. A withdrawal transaction requires

¹ Real time behavior actually imposes additional constraints on DASM runs, for instance, to ensure that the agents react instantaneously (see [15] for details).

the following logical steps: (1) *input the bank card and PIN code*; (2) *check the validity of both the bank card and PIN code*; (3) *input the amount to be withdrawn*; (4) *check the account balance against the credit line*; (5) *on approval update the account balance*; (6) *output cash or denial notification*.

Assuming an unreliable communication medium, timeout mechanisms may cause the cancellation of a transaction spontaneously at any time.

4.2 ATM abstract machine model

Basically, the ATM control forms a distributed embedded system which should be modeled in terms of a DASM consisting of three separate agents (each of which represents one of the interacting entities). Nonetheless, one may start by modeling only the ATM manager assuming the roles of both the authentication manager and the account manager to be marginal. That is, the latter two entities are considered to be part of the global environment of the ATM manager. This view allows us to focus on the key behavioral aspects first. In subsequent refinement steps, this model then can easily be extended to a DASM by making the behavior of the two other agents explicit.

Intuitively, an ATM *activation event* occurs whenever a user requests the service. The user then enters the card number, PIN code and the desired amount. Beyond reading this data from the global environment, the machine can also perform more complex interactions with the global environment to get other, non-trivial information like authentication approval/rejection and transaction approval/denial. The abstraction mechanisms allow us not only to define (and decide about) the environment, but also enable us to freely choose the level of detail and precision. Here, we abstract from all the interactions between the ATM and its global environment (including the user and the two other services) by assuming that they take place in the background (called the *local* environment). The ATM agent communicates with its local environment through various monitored functions. Henceforth, we refer to the local environment as environment. In this initial ATM control model, neither the behavior of the authentication manager nor that of the account manager will be formally defined; still, we have of course some idea of the operations they perform and the constraints that do apply.

The machine is idle in the initial state. `ActivationEvent` is a monitored Boolean-valued function. Two other monitored functions `getCardData` and `getPinCode` serve to obtain the input data as soecified by the user.

```
ATMIdle ≡ if ActivationEvent then
    data := getCardData, code := getPinCode, mode := checking
```

This way, a series of interactions between the ASM and the environment takes place, and in each step some required information, ranging from the

requested withdrawal amount to the user authentication, is obtained from the environment and is used to perform the operation. Authenticated is another important monitored function that abstractly provides the authentication service to the ATM abstract machine. Once the information is authenticated, the ATM obtains the requested amount.

ATMChecking $\equiv$ if $\text{mode} = \text{checking} \wedge \text{Authenticated}(\text{date}, \text{code})$ then $\text{amnt} := \text{getWithdrawAmount}, \text{mode} := \text{Processing}$
--

While in the processing mode, the abstract machine checks whether or not the requested amount gets approved by the account management service (as indicated by a monitored predicate ValidTransaction).

ATMProcessing $\equiv$ if $\text{mode} = \text{processing} \wedge \text{ValidTransaction}(\text{data}, \text{amnt})$ then $\text{ReleaseCash}(\text{amnt}), \text{UpdateBalance}(\text{data}, \text{amnt}), \text{mode} := \text{idle}$

On the other hand, if the card information is invalid or the requested amount exceeds the approved credit line, the transaction is cancelled and an error notification is generated.

ATMError $\equiv$ if $\neg \text{Authenticated}(\text{date}, \text{code}) \vee \neg \text{ValidTransaction}(\text{data}, \text{amnt})$ then $\text{OutputErrorNotification}, \text{mode} := \text{idle}$
--

This example also makes use of another convenient feature. ReleaseCash, UpdateBalance and OutputErrorNotification are parts of the model that are meant to perform the final operations. However, as we do not want to deal with the details of such operations at this level of abstraction, we left the definition of these rule abstract. The complete definition of these rules thereby is left to the next refinement step.

Furthermore, a transaction in progress may be canceled anytime due to an externally caused timeout event. CancellationEvent is a monitored Boolean-valued function indicating the occurrence of a timeout. Hence, the complete behavior of the ATM control is described as follows.

ATMProgram $\equiv$ if $\text{mode} = \text{idle}$ then ATMIdle else if CancellationEvent then $\text{OutputCancellationNotification}, \text{mode} := \text{idle}$ else ATMChecking, ATMProcessing, ATMError
--

5. CONCLUSIONS

Based on experience from extensive applications of the DASM paradigm to modeling distributed embedded systems and communications software, we review here its potential for agile formalization and literate specification. Specifically, we consider the problem of defining the boundary between a system and its environment in semantic modeling of concurrency, reactivity and time. This research contributes to the effective use of formal methods as practical system design tools rather than toys for academic pleasure.

REFERENCES

- [1] Ch. Beierle, E. Börger, I. Đurdanovic, U. Glässer and E. Riccobene. Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code. In J.-R. Abrial, E. Börger and H. Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of LNCS, State-of-the-Art Survey, 52-78, Springer-Verlag, 1996
- [2] D. Berry. Formal Methods: The Very Idea – Some thoughts about why they work when they work. *Science of Computer Programming* 42(1): 11-27 (2002)
- [3] E. Börger. High Level System Design and Analysis using Abstract State Machines. In D. Hutter et al., eds., *Current Trends in Applied Formal Methods* (FM-Trends 98). Springer LNCS 1641, pp. 1-43, 1999
- [4] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003
- [5] E. Börger, U. Glässer and W. Müller. Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines. In C. Delgado Kloos and Peter T. Breuer, editors, *Formal Semantics for VHDL*, Kluwer Academic Publishers, 1995, 107-139
- [6] N. Dorny. Systems Engineering. University of Pennsylvania, Philadelphia, PA, URL: www.seas.upenn.edu/~dorny/se.htm#si
- [7] R. Farahbod, U. Glässer and M. Vajihollahi. Specification and Validation of the Business Process Execution Language for Web Services. In *Proc. of the 11th International Workshop on Abstract State Machines* (ASM'2004), volume 3065 of LNCS, pages 69-86, Springer-Verlag, 2004
- [8] M. Fowler. The New Methodology, April 2003, online reference at URL: www.martinfowler.com/articles/newMethodology.html
- [9] U. Glässer, R. Gotzhein and A. Prinz. Formal Semantics of SDL-2000: Status and Perspectives. *Computer Networks* 42(3): 343-358, Elsevier, 2003
- [10] U. Glässer, Y. Gurevich and M. Veanes. An Abstract Communication Model. Microsoft Research, Technical Report, MSR-TR-2002-55, May 2002
- [11] U. Glässer, Y. Gurevich and M. Veanes. Abstract Communication Model for Distributed Systems. To appear in *IEEE Transactions on Software Engineering*.
- [12] U. Glässer and M. Veanes. Universal Plug and Play Machine Models. In B. Kleinjohann et al. (Eds.): *Design and Analysis of Distributed Embedded Systems*, IFIP Conference Proceedings 219, Kluwer Academic Publishers, 2002
- [13] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms, *ACM Transactions on Computational Logic*, 1(1): 77-111, July 2000
- [14] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, Oxford University Press, 1995, pages 9-36
- [15] Gurevich and J. Huggins: The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions. In H. Kleine Büning, editor, *Computer Science Logic*, volume 1092 of LNCS, pages 266-290, Springer, 1996
- [16] W. Müller, J. Ruf and W. Rosenstiel. An ASM Based SystemC Simulation Semantics. In W. Müller, W. Rosenstiel and J. Ruf. *SystemC – Methodologies and Applications*, Kluwer Academic Publishers, 2003
- [17] R. Stärk, J. Schmid, and E. Börger, *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001