Applying Predicate Abstraction to Abstract State Machines

Alessandro Bianchi(✉), Sebastiano Pizzutilo, and Gennaro Vessio

Department of Informatics, University of Bari, 70125, Bari, Italy
{alessandro.bianchi,sebastiano.pizzutilo,
gennaro.vessio}@uniba.it

Abstract. Abstract State Machines (ASMs) represent a general model of computation which subsumes all other classic computational models. Since the notion of ASM state naturally captures the classic notion of program state, ASMs are suitable to be verified through a *predicate abstraction* approach. The aim of this paper is to discuss how predicates over ASM states can support the formal verification of ASM-based models. The proposal can overcome the main limitations that penalize traditional model checking techniques applied to ASMs.

1 Introduction

Abstract State Machines (ASMs) represent a general model of computation which subsumes all other classic computational models [27], [12]. Indeed, they suffice to capture the behavior of wide classes of sequential [20], parallel [8] and distributed algorithms [18]. The origin of this generality lies in the notion of ASM *state*. In classic formalisms, such as finite state machines and Turing machines, states are symbolically represented by (sequence of) symbols belonging to finite alphabets [21]. Conversely, ASM states are syntactically and semantically represented by algebraic structures defined over finite signatures. Therefore, ASM states can model any object of arbitrary complexity [20].

Thanks to their generality, ASMs have been successfully applied for modeling critical and complex systems in a wide range of application domains, and for analyzing their computationally interesting properties, both domain-independent (e.g. the *termination* of the execution, *deadlock*- and *starvation-freedom*, and so on) and domain-specific (e.g. security issues, the movement of robotic arms, synchronization issues of real-time controllers, and so on) [9]. However, despite the advantages they provide, the computational power and the arbitrary complexity of the formalism cause an unavoidable drawback: several computationally interesting properties are undecidable, so the formal verification of ASM-based models cannot be fully automatized [29].

Traditional model checking approaches to the problem of verifying properties typically model systems with finite state machines (or variants) and express the properties to be verified using some temporal logic [5]. Analogously, when model checking techniques are applied to ASMs, the ASM-based model under study is transformed into the input required by the adopted model checker [14], [3], which is in general less

K. Gaaloul et al. (Eds.): BPMDS 2015 and EMMSAD 2015, LNBIP 214, pp. 283–292, 2015.
DOI: 10.1007/978-3-319-19237-6_18

expressive. Therefore, this approach suffers from two main limitations: the loss of expressive power due to the translation of the ASM specifications, and the difficulty in using the declarative notations of temporal logics, considered less comfortable for practitioners with respect to operational specifications of the properties (e.g. [4]).

Our long term research is aimed at providing a theoretical framework for treating properties analysis entirely within the ASM framework, i.e. without translating ASMs, and without using temporal logics. To this end, the present paper takes a step towards the application of a *predicate abstraction* approach for formally analyzing ASMs. Predicate abstraction is a popular and widely used technique for automatizing the verification of programs [19], [11]. It consists in the approximation of the program states into a finite number of predicates defined over these states. In this context, the goal of the present paper is to support the verification of ASM properties through predicates over the states. In this way, the goal of formally verifying ASM models entirely within the ASM framework can be achieved.

The rest of this paper is organized as follows. Section 2 is about related work. Section 3 provides background knowledge about the ASM formalism. Section 4 deals with predicate abstraction for ASMs. Section 5 depicts some illustrative examples. Finally, Section 6 concludes the paper and sketches future developments.

2 Related Work

The ASM formalism allows both manual and automated formal verification of systems. In [9] numerous proofs are provided to illustrate how a modeler can verify properties of a given ASM. These proofs range from simple to complex and, since ASMs are executable machines which lend themselves to traditional inductive proofs, they are often formulated in a mathematical way. For example, in [15] a manual verification calculus based on the Hoare logic is proposed. Conversely, in [14] and [3] the authors use an automatic model checking approach for verifying ASMs. However, the translation of the ASM into the input required by the model checkers may cause a loss of expressive power. Moreover, in all these cases, properties are expressed in some temporal logic [5]. But, the effectiveness of this hybrid approach is not unanimously recognized: several authors emphasize the need of an entirely operational specification of properties within the same formalism, e.g. [22], [4].

ASMs have been successfully used for modeling several systems, often concurrent and distributed, and for investigating their properties. For example, an ASM specification to model concurrency in a Web browser and to prove some consistency properties has been proposed in [17]; ASMs have also been used to model and validate vision-based robot control applications in [25]; and they have been applied for studying Grid systems in [6]. However, all these works are characterized by the lack of a theoretical framework for systematically treating the analyzed properties.

Concerning the application of predicate abstraction to formal methods, other formalisms, such as Petri nets [10], already employs predicates on states whenever properties are to be analyzed. However, Petri nets typically provide only few levels of abstraction, so they are not able to support refinements till to implementation details.

Conversely, the expressive power of ASMs provides a way to describe algorithmic issues in a simple abstract pseudo-code, which can be translated into a high level programming language source code in a quite simple manner [9]. Furthermore, predicate abstraction has been already used within the ASM formalism in [16]; however, the authors use what they call *test predicates* in a way which is different from ours. Indeed, they use predicates on states in order to generate test sequences.

3 Background on ASMs

Abstract State Machines are finite sets of so-called *rules* of the form **if** *condition* **then** *updates* (possibly with the **else** clause) which transform *abstract* states [9]. An ASM state is an algebraic structure, i.e. a domain of objects with functions and relations defined on them. Partial functions are turned into total functions by using the special value *undef*. Moreover, without loss of generality, relations are treated as particular functions that evaluate to *true* or *false*. On the other hand, the concept of rule reflects the notion of transition occurring in traditional transition systems: *condition* is a first-order formula whose interpretation can be *true* or *false*, whereas *updates* is a finite set of assignments of the form $f(t_1, \ldots, t_n) := t$, whose execution consists in changing in parallel the value of the specified functions to the indicated value.

Pairs of function names, fixed by a signature, together with values for their arguments are called *locations*: they abstract the notion of memory unit. Therefore, a state can be viewed as a function that maps locations to their values: the current configuration of locations together with their values determines the current state of the ASM. As usual in computational models, an ASM *step* is a pair (s, s') of states: in a given state, all conditions are checked, so that all updates in rules whose conditions evaluate to *true* are simultaneously executed, and the result is a transition of the machine from that state to another. Note that for the unambiguous determination of the next state, updates must be *consistent*, i.e. no pair of updates must refer to the same location.

A generalization of basic ASMs is represented by Distributed ASMs (DASMs) [9], [18], capable to capture the formalization of multiple agents acting in a distributed environment. Essentially, a DASM is intended as an arbitrary but finite number of independent agents, each executing its own underlying ASM. In a DASM the keyword **self** is used for supporting the relation between local and global states and for denoting the specific agent which is executing a rule.

4 Predicates Over ASM States

Classic computational models, such as finite state machines and Turing machines, represent the current state of the computation with (sequence of) symbols belonging to finite alphabets [21]. This poses a limitation: the representation of states is restricted to a specific data structure. Instead, as explained in Section 3, ASMs allow any algebraic structure to serve as representation of states. This results in a great amount of details specifying the states, so making the analysis of the properties of the whole system more difficult, mainly for what concerns the comprehension of the semantics of each state, with respect to the computational behavior of the modeled system.

For better explaining this issue, the next section will elaborate two examples, both concerning distributed systems: the analysis of *starvation-freedom* [2] and *deadlock-freedom* [28], and the analysis of the computation of an agent capable to play two or more roles at the same time. In the former case, the simple execution of one or more updates does not necessarily involve the change of the locations values in such a way that the process makes real computational progress, so driving to starvation. In fact, an ASM could starve even if the computation continues to evolve through different states. In other words, it is difficult to recognize effective progress. As an extreme case, this computational behavior can lead to deadlock. On the other hand, the second case concerns, for example, the case of a process that acts both as a client with respect to a service, and, simultaneously, as a server with respect to another service. In this case, ASMs easily capture in a same state different computational activities to be run in parallel. However, it is difficult for the modeler to distinguish, inside the same state, what computational branches have been entered or not.

In order to overcome these problems, the need of an abstraction framework capable to capture the semantics of the ASM states arises. More precisely, there is the need to partition the set of locations into subsets and extract from them the locations specifically interesting for the verification purposes. To this end, we apply a predicate abstraction approach. Predicate abstraction is a popular and widely used technique proposed to analyze programs [19], [11]. It aims at generating an abstract model from the concrete system to be verified, so checking the former instead of the latter. Briefly speaking, the system states are mapped to model states according to their evaluation with respect to a finite set of predicates defined over the system states. The model has the same control flow of the original program but it concerns only the predicates over the states. The model can then be used in the place of the original program when performing model checking, theorem proving, or other kinds of verification techniques.

Literature agrees that a program state coincides with the configuration of program variables and their current values, e.g. [24]. Analogously, an ASM state coincides with the configuration of ASM locations and values. So, since there exists a natural parallelism between classic program states and ASM states, predicate abstraction can be applied to ASMs as much as to programs of traditional programming languages. In this context we can apply predicate abstraction to ASMs through the following:

Definition. *A* predicate ϕ over an *ASM* state s *is a first-order formula defined over the locations in* s*, such that* $s \vDash \phi$.

Predicates over the states serve to represent the semantics of each state, i.e. the properties locally satisfied, and can be regarded as a non-injective labeling function that maps predicates to each state. An ASM model can then be equipped with a set of predicates $\Phi = \{\phi_1, \ldots, \phi_n\}$, such that, in the current state, each ϕ_i can be satisfied or not. In this way, the ASM control flow can be represented by the truth value of the predicates over the states, i.e. by composing the local properties of the various states. So, global properties to be verified can be analyzed by focusing on this composition.

Note that our use of predicate abstraction is quite different with respect to the traditional way: instead of extracting abstract models from the ASMs to be verified, our aim is to use predicates over the states in order to support the verification of ASM models. In particular, applying predicate abstraction to ASMs induces the partition of locations we need for expressing the semantics of the states.

5 Two Examples

In order to show the application of predicate abstraction over ASM states two examples are discussed: the Dining Philosophers problem, and the Ad-hoc On-demand Distance Vector (AODV) routing protocol for Mobile Ad-hoc NETworks (MANETs). The first example deals with starvation-freedom and deadlock-freedom analysis: here the same value for a predicate holds for several states. The second example discusses the case of several predicates holding over the same state in the context of an agent playing several different roles within the same system.

5.1 Dining Philosophers

The Dining Philosophers problem, due to Dijkstra [13], is a well-known metaphor for discussing concurrent processes. Five philosophers are sitting around a table with a bowl of spaghetti in the middle. For them, life consists only of two moments: thinking and eating, rigorously using two forks. Since each philosopher has a right fork and a left fork, (s)he thinks till both forks become available, eats for a certain amount of time, then stops eating (putting back both forks on the table), and starts thinking again. The problem is that in between two neighboring philosophers there is only one fork: each one shares a fork with a neighbor. The ASM-based model of Dining Philosophers is in [9]: it is a DASM with a set of *philosophers* = $\{p_1, \ldots, p_5\}$, i.e. the agents of the system, and a set of *forks* = $\{f_1, \ldots, f_5\}$, i.e. their shared resources. The computation evolves through the states characterized by the following predicates:

- `thinking`: $\neg(owner(rightFork(\mathbf{self})) = \mathbf{self} \vee owner(leftFork(\mathbf{self})) = \mathbf{self})$. The philosopher is thinking, so (s)he is waiting for both forks to become available;
- `eating`: $owner(rightFork(\mathbf{self})) = \mathbf{self} \wedge owner(leftFork(\mathbf{self})) = \mathbf{self}$. The philosopher is eating, so (s)he has obtained both forks,

where:

- *rightFork*: *philosophers* → *forks* indicates a philosopher's right fork;
- *leftFork*: *philosophers* → *forks* indicates a philosopher's left fork;
- *owner*: *forks* → *philosophers* denotes the current user of a fork.

Initially, each philosopher p_i thinks, and has fork f_i on the right and fork f_{i-1} on the left, except for p_1 that has fork f_5 on the left. The ASM program for p_i is shown below:

PhilosopherProgram(p_i) =
if *owner*(*rightFork*(**self**)) = *undef* ∧ *ower*(*leftFork*(**self**)) = *undef* **then** {
owner(*rightFork*(**self**)) := **self**
owner(*leftFork*(**self**)) := **self**
}
if *owner*(*rightFork*(**self**)) = **self** ∧ *owner*(*leftFork*(**self**)) = **self then** {
owner(*rightFork*(**self**)) := *undef*
owner(*leftFork*(**self**)) := *undef*
}

Ideally, p_i, denoted by **self**, would like to execute alternatively the two rules above to get and later to release the desired forks. Indeed, even if ASM rules are executed in parallel by definition, the second rule (i.e. the second **if-then** statement in the program above) can be performed if and only if the first rule has been previously executed.

During the waiting for both forks, the computation of p_i can go through four states:

1. (*owner*(*rightFork*(**self**))=philosopher on the right) ∧ (*owner*(*leftFork*(**self**))=*undef*);
2. (*owner*(*rightFork*(**self**))=*undef*) ∧ (*owner*(*leftFork*(**self**)) = philosopher on the left);
3. (*owner*(*rightFork*(**self**)) = philosopher on the right) ∧ (*owner*(*leftFork*(**self**)) = philosopher on the left);
4. (*owner*(*rightFork*(**self**)) = *undef*) ∧ (*owner*(*leftFork*(**self**)) = *undef*).

In all four states, the `thinking` predicate holds. The state changes whenever an update is executed by the neighboring philosophers over the shared locations; however, the ASM could not make a real computational step towards the state characterized by the `eating` predicate. In fact, only state (4) allows the first rule to be executed, so the desired state can be reached. In this particular case, even if the ASM state changes, the computation could not make a real progress, i.e. the process risks to starve.

For verification purposes, predicate abstraction is very suitable for capturing starvation. In particular, starvation could arise if there are rules: (*i*) whose condition concerns functions which represent the dependency of the agent from external resources; and (*ii*) whose execution/non-execution could have effects that does not change the value of the predicate over the states which represents the "waiting for something" issue. In our case, the first rule of the *PhilosopherProgram* shows these issues.

Finally, it is worth noting that, if resource holding holds, the scenario above is affected by the risk of deadlock: each philosopher picks up his/her right fork and waits for the left fork to become available. Thanks to predicate abstraction, this issue can be captured by the following predicate: $owner(rightFork(p)) = p, \forall\, p \in Philosophers$. Therefore, the model is deadlock-free if, during the DASM computation, it is not possible that its global state fulfill the predicate above, i.e. at any moment there must be at least one ASM whose state fulfills ¬(*owner*(*rightFork*(**self**)) = **self**).

5.2 A MANET Routing Protocol

Mobile Ad-hoc NETworks (MANETs) [1] are wireless networks designed for communications among nomadic hosts, in absence of fixed physical infrastructure. Each node plays a twofold role: end-point of a communication session and router supporting other communications. Both activities evolve concurrently. A MANET that adopts the AODV routing protocol [26] can be modeled by a DASM including a homogeneous set of $hosts = \{h_1, \ldots, h_n\}$. Each ASM can be in one of different states, which are characterized by the following predicates over the states:

- `idle`: the host is inactive. Its formula is given by: *wishToInitiate*(**self**, *dest*) = *false* ∧ *receivedRREQ*(**self**, *dest*) = *false* ∧ *isEmpty*(*replies*(**self**)) = *true*, ∀ *dest* ∈ *hosts*;
- `router`: the host has received a control packet directed to it. It is characterized by *receivedRREQ*(**self**, *dest*) = *true*;

- `initiator`: the host has to start a new communication session. It is characterized by *wishToInitiate*(**self**, *dest*) = *true*;
- `forwarding`: the host is forwarding a control packet to another recipient. It is characterized by *isEmpty*(*replies*(**self**)) = *false*.

where:

- *wishToInitiate*: *hosts* × *hosts* → *boolean* indicates whether a new communication session to a destination is required;
- *receivedRREQ*: *hosts* × *hosts* → *boolean* indicates whether an RREQ packet has been received;
- *isEmpty*: *queues* → *boolean* states if a queue of messages is empty or not.

In fact, in order to model broadcasting and unicasting, each host is associated with two queues of messages: *requests* and *replies*, including: RREQ (Route REQuest) packets for requesting a route to a desired destination, and RREP (Route REPly) packets for replying this request, respectively. This allows us to model sending/receiving of packets by means of enqueuing/dequeuing abstract messages into the corresponding queue. In addition, each ASM includes the following functions:

- *routingTable*: *hosts* → PowerSet(*records*), which represents the information about the hosts recorded into the host's routing table;
- *hostsInRT*: PowerSet(*records*) → PowerSet(*hosts*), which returns the set of the hosts stored in a given routing table, for checking information about hosts.

The ASM pseudo-code of the *i*-th host is shown below.

```
HostProgram(h_i) =
  If ¬isEmpty(requests(self)) then {
    RREQ = top(requests(self))
    nextHop = sender of top(requests(self))
    updateRoutingTable(self, RREQ)
    receivedRREQ(self, dest):= true
    Router(RREQ, nextHop)
  }
  if wishToInitiate(self, dest) = true then
    Initiator(dest)
  if ¬isEmpty(replies(self)){
    RREP = top(replies(self))
  if RREP.init ≠ self then{
    nextHop = select c.nextHop ∈ hostsInRT(routingTable(self))
        with RREP.init = c.dest
    updateRoutingTable(self, RREP)
    UnicastRREP(RREP, nextHop)
    dequeue RREP from replies(self)
  }
}
```

It is worth noting that the activation of a host unfolds different computational branches: two of them lead to the execution of the *Router* or *Initiator* submachine, respectively; in the third case, the forwarding of RREPs is executed. In particular, the *Router* submachine models the behavior of the node when it supports communications between other end-points; instead, the *Initiator* submachine models the behavior of the node when it acts as the initiator of a new communication session. Note that if initiator does not know a route to reach destination, then it starts a *route discovery* process aimed at discovering this route. For clarity, an ASM *submachine* is a parameterized rule [9]: it allows the declaration of *local* functions, so that each call of a submachine works with its own instantiation of its local functions.

For verification purposes, predicate abstraction can help in expressing the node's behavior. In our case, the simultaneous fulfillment of different predicates over the same ASM state is very suitable for capturing the intrinsic concurrency of the nodes' computation. Indeed, when the MANET starts operating, each host is idle. But, during the normal execution of the MANET, a host can, for example, fulfill the `router` predicate with respect to a destination, but at the same time it can fulfill other predicates, e.g. `initiator`, for what concerns other destinations. The values of the arguments help in distinguishing the various cases.

Moreover, predicate abstraction can help in investigating some specific properties for MANETs: the correctness of the activities of sending/receiving packets, the starvation-freedom of initiator when it starts a route discovery process, and so forth. For example, concerning the starvation issue, the presence of a timeout in the *Initiator* submachine allows initiator to escape the waiting for RREPs if a route to destination cannot be found. So, for that specific destination, after the timeout expiration, the ASM state does not fulfill the `initiator` predicate but the `idle` predicate.

For the purposes of the present work, it is not necessary to further detail the *updateRoutingTable* and *UnicastRREP* rules, as well the *Router* and *Initiator* submachines. The interested reader can find the full specification of the model and the proof of its correctness in [7].

6 Conclusion

This paper proposes predicate abstraction over ASM states. The proposed approach can support the verification of ASM-based models by overcoming the main limitations that penalize classic model checking techniques applied to ASMs. In fact, applying predicate abstraction to ASMs enables the analysis of the global properties of the system to be verified through a representation of its local properties through predicates over the states. In this way, the analysis is executed entirely within the ASM framework, without the need of less expressive models and without the burden of temporal logics. Possible applications are represented by various kinds of critical and complex systems that can benefit from a formal approach: Internet-based services, security protocols, Cloud, Grid and mobile systems, and so on.

It is worth specifying that researchers usually distinguish between two classes of properties [23]. *Safety* properties specify that "something bad never happens", e.g. deadlock-freedom. Instead, liveness properties stipulate that "something good eventually happens", e.g. starvation-freedom. From this point of view, safety properties

require that certain predicates over the states, which represent a “bad thing”, must never be satisfied, or, alternatively, their negation must always hold during the computation. Conversely, liveness properties require that certain predicates over the states, which represent a “good thing”, must eventually be satisfied during the computation. Future directions of this research should investigate specific features of predicate abstraction with respect to the specific kinds of properties to be analyzed.

Acknowledgements. This work has been partially funded by the Italian Ministry of Education, University and Research, within the “Piano Operativo Nazionale” PON02_00563_3489339.

References

1. Agrawal, D.P., Zeng, Q.A.: Introduction to Wireless and Mobile Systems. Thomson Brooks/Cole (2003)
2. Alpern, B., Schneider, F.B.: Defining Liveness. Information Processing Letters **21**(4), 181–185 (1985)
3. Arcaini, P., Gargantini, A., Riccobene, E.: AsmetaSMV: AWay to link high-level ASM models to low-level NuSMV specifications. In: 2th International Conference on Abstract State Machines, Alloy, B and Z, pp. 61–74 (2010)
4. Arcaini, P., Gargantini, A., Riccobene, E.: CoMA: conformance monitoring of java programs by abstract state machines. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 223–238. Springer, Heidelberg (2012)
5. Baier, C., Katoen, J.P.: Principles of Model Chacking. The MIT Press (2008)
6. Bianchi, A., Manelli, L., Pizzutilo, S.: An ASM-based Model for Grid Job Management. Informatica (Slovenia) **37**(3), 295–306 (2013)
7. Bianchi, A., Pizzutilo, S., Vessio, G.: Suitability of Abstract State Machines for Discussing Mobile Ad-hoc Networks. Global Journal of Advanced Software Engineering **1**, 29–38 (2014)
8. Blass, A., Gurevich, Y.: Abstract State Machines Capture Parallel Algorithms. ACM Transactions on Computational Logic **4**(4), 578–651 (2003)
9. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer (2003)
10. Chen, Z., Zhou, C., Ding, D.: Automatic abstraction refinement for petri nets verification. In: 10th Int. Workshop on High-Level Design, Validation and Test, pp. 168–174 (2005)
11. Das, S., Dill, D.L., Park, S.: Experience with predicate abstraction. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 160–171. Springer, Heidelberg (1999)
12. Dershowitz, N.: The Generic Model of Computation. Electronic Proceedings in Theoretical Computer Science (2013)
13. Dijkstra, E.W.: Hierarchical Ordering of Sequential Processes. ACTA Informatica **1**(2), 115–138 (1971)
14. Farahbod, R., Glässer, U., Ma, G.: Model Checking CoreASM Specifications. In: 14th International ASM Workshop (2007)
15. Gabrisch, W.: A Hoare-Style Verification Calculus for Control State ASMs. In: 5th Balkan Conference on Informatics, pp. 205–210 (2012)

16. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using spin to generate testsfrom ASM specifications. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 263–277. Springer, Heidelberg (2003)
17. Gervasi, V.: An ASM model of concurrency in a web browser. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 79–93. Springer, Heidelberg (2012)
18. Glausch, A., Reisig, W.: An ASM-characterization of a class of distributed algorithms. In: Abrial, J.-R., Glässer, U. (eds.) Rigorous Methods for Software Construction and Analysis. LNCS, vol. 5115, pp. 50–64. Springer, Heidelberg (2009)
19. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: 9th International Conference on Computer Aided Verification, pp. 72–83 (1997)
20. Gurevich, Y.: Sequential Abstract State Machines Capture Sequential Algorithms. ACM Transactions on Computational Logic **1**(1), 77–111 (2000)
21. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)
22. Klai, K., Desel, J.: Checking soundness of business processes compositionally using symbolic observation graphs. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 67–83. Springer, Heidelberg (2012)
23. Kindler, E.: Safety and Liveness Properties: A Survey. EATCS Bulletin **53**, 268–272 (1994)
24. Laplante, P.: Dictionary of Computer Science, Engineering and Technology. CRC Press (2000)
25. Luzzana, A., Rossetti, M., Righettini, P., Scandurra, P.: Modeling synchronization/communication patterns in vision-based robot control applications using ASMs. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 331–335. Springer, Heidelberg (2012)
26. Perkins, C.E., Belding-Royer, E.M., Das, S.R.: Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 http://tools.ietf.org/html/rfc3561 (2003)
27. Reisig, W.: The Expressive Power of Abstract State Machines. Computing and Informatics **22**, 209–219 (2003)
28. Singhal, M.: Deadlock Detection in Distributed Systems. IEEE Computer **22**(11), 37–48 (1989)
29. Spielmann, M.: Automatic verification of abstract state machines. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 431–442. Springer, Heidelberg (1999)