Towards an ASM-based Characterization of the Deadlock-freedom Property

Alessandro Bianchi, Sebastiano Pizzutilo and Gennaro Vessio
Department of Informatics, University of Bari, Bari, Italy

Keywords: Abstract State Machines, Deadlock, Properties Analysis.

Abstract: The present paper investigates the effectiveness of the Abstract State Machine (ASM) formalism into studying the *deadlock-freedom* property in distributed systems. To this end, the well-known Dining Philosophers problem, prone to deadlock, is here modeled through ASMs and deadlock is studied. The experience suggests a provisional reformulation of the classic necessary conditions for deadlock in terms of ASMs.

1 INTRODUCTION

The problem of detecting and preventing deadlock in distributed systems has been faced by both academia and practitioners since a long time; however, it is far from a generally adopted solution. This is due to the intrinsic feature of the deadlock analysis problem to be semi-decidable. In order to deepen into the analysis of this issue, the present paper is aimed at investigating the adequateness of the Abstract State Machine formalism (simply ASM in the following) (Gurevich, 2000) in treating this problem. Provisional results on applying ASMs to *starvation-freedom* analysis are provided in (Bianchi et al., 2016).

In comparison with other well-known approaches, we focus on ASMs because of the advantages they provide under several viewpoints. When expressivity is considered, ASMs represent a general model of computation which subsumes all other classic computational models (Gurevich, 1995). Considering methodological issues, the ASM formalism has been successfully applied for the design and analysis of critical and complex systems in several domains, and a specific development method got prominence in the last years (Börger and Stärk, 2003). Finally, considering the implementation point of view, the capability of translating formal specifications into executable code, in order to conduct simulations of the models, is provided by tools like CoreASM (Farahbod et al., 2007a).

For the purposes of the present work we assume that a deadlock happens when processes require the access to resources held by other processes, but they themselves hold resources that the other processes need. Until deadlock is not resolved, processes are blocked indefinitely. We recall that (Coffman et al., 1971) stated four general necessary conditions, that must hold simultaneously, for a deadlock to occur:

- *Mutual Exclusion*: processes claim exclusive control of the resources they require.
- *Resource Holding*: processes hold resources already allocated to them while waiting for the other requested resources.
- *No Preemption*: resources cannot be removed from the processes holding them until they are used to completion.
- *Circular Wait*: a circular chain of processes exists so that each process holds (at least) one resource that is requested by the next process in the chain. In other words, there is a set of processes $\{P_1, \ldots, P_n\}$ such that P_i is waiting for a resource held by $P_{i+1} \bmod n$.

In this paper the well-known Dining Philosophers problem (Dijkstra, 1971), prone to deadlock, is modeled with ASMs and deadlock is studied. The analysis of the model allows us to derive the characteristics of the model driving the risk of deadlock, so providing a provisional reformulation of the classic necessary conditions for deadlock in terms of ASMs.

The rest of this paper is organized in the following fashion. Section 2 is about related work. Section 3 provides background knowledge on the ASM framework. Section 4 deals with the ASM-based Dining

Bianchi, A., Pizzutilo, S. and Vessio, G.
Towards an ASM-based Characterization of the Deadlock-freedom Property.
DOI: 10.5220/0006002901230130
In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT*, pages 123-130
ISBN: 978-989-758-194-6
Copyright © 2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

Philosophers problem and its analysis. Section 5 discusess the experience. Finally, Section 6 concludes the paper.

2 RELATED WORK

The capability of Abstract State Machines to subsume all other classic computational models has been stated in several works, e.g. (Gurevich, 1995), (Reisig, 2003), (Dershowitz, 2013). Thanks to this generality, an ASM sequential thesis has been proved by (Gurevich, 2000). It states that ASMs suffice to capture the behavior of wide classes of sequential systems at any desired level of abstraction. Research effort has then been devoted to extend this thesis to parallel machines (Blass and Gurevich, 2003) and concurrent computations (Glausch and Reisig, 2009), (Börger and Schewe, 2015). The latter results seem to comprise a large class of distributed algorithms. Although no theoretical result proves that ASMs suffice to capture the behavior of all classes of distributed algorithms, they have shown to be sufficiently expressive to model concurrency in many applications: network consensus, master-slave agreement, leader election, phase synchronization, load balance, mobile ad-hoc networks, and so on. These observations justify the suitability of the ASM framework in analyzing systems properties in a wide range of domains, included deadlock-freedom.

In the state of the art, ASMs already support both manual and automatic formal verification of systems. Concerning manual analysis, ASMs are machines equipped with a notion of run that lend themselves to traditional mathematical reasoning or manual simulation. Such proofs range from simple to complex and are conceived for being used by human experts. An interesting manual approach is provided in (Gabrisch and Zimmermann, 2012), where a verification calculus based on the Hoare logic is proposed. However, the calculus only considers partial correctness, i.e. the result of the computation is what was expected, and is only tailored for a specific class of ASMs. Moreover, it is worth noting that a logic for ASMs exists (Stärk and Nanchen, 2001). However, it does not provide operational characterizations of properties, such as deadlock-freedom. The ASM notion of run is very helpful for supporting the practitioners' work, independently from the possibility of developing automatic verification mechanisms. Nevertheless, since it requires human effort, the manual approach does not offer absolute guarantee and is error-prone.

Concerning automatic analysis, several examples of model checking techniques applied to ASMs exist, e.g. (Del Castillo and Winter, 2000), (Farahbod et al., 2007b), (Arcaini et al., 2010), (Rafe and Doostali, 2012). However, the Turing-completeness of the formalism (Gurevich, 2000) causes an unavoidable drawback: properties are, in general, undecidable, so the formal verification of ASM specifications cannot be fully automatized (Spielmann, 1999). For this reason, the translation of the ASM under study into the input required by the adopted model checker may cause a loss of expressive power.

For what specifically concerns deadlock analysis, deadlock was pointed out for the first time in (Dijkstra, 1968). Thereafter, many studies have been devoted to investigate this issue from different aspects, for example in operating systems (Kameda, 1980), distributed databases (Knapp, 1987) and communication networks (Brookes and Roscoe, 1991). More recently, deadlock analysis has gained much attention in the context of formal methods, for example in model checking (Bingham et al., 2013) and π-calculus (Padovani, 2014). According to (Singhal, 1989), all these issues can be classified in two types of *distributed* deadlock: *resource* deadlock and *communication* deadlock. The first one emerges in multi-process systems with shared resources. It is worth noting that the necessary conditions provided in (Coffman et al., 1971) only capture this type of deadlock. Conversely, a communication deadlock emerges when resources are replaced by messages the processes wait for. A set of processes is then deadlocked if each process is waiting for a message another process must send, but no process ever sends a message. Moreover, some authors, e.g. (Holt, 1972), also consider *single* deadlocks: a single process may deadlock if it waits for an external event executed by an external entity, such as the operating system, but this event never occur. In this work we are only interested into studying distributed resource deadlocks in the sense of (Coffman et al., 1971).

3 THE ASM FRAMEWORK

An Abstract State Machine is a tuple (Σ, S, R, P_M) (Börger and Stärk, 2003). Σ is a signature: a finite collection of function names each characterized by an arity, that is the number of arguments that function takes. Partial functions are turned into total functions by using the special value *undef*. Instead, relations are expressed as particular functions that always evaluate to *true*, *false* or *undef*. S is a finite set of *abstract* states. The concept of abstract state extends the usual notion of *state* occurring in finite state machines: it is an algebra over the signature Σ, i.e. a non-empty set of objects of arbitrary complexity together with interpretations of the functions in Σ. R is a finite set of so-called *rules* of the form **if** *condition* **then** *updates* which transform the states of the machine. 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. P_M is a distinguished rule of arity zero, called the *main rule* or *program* of the machine, which represents the starting point of the computation.

Pairs of function names together with values for their arguments are called *locations*: they abstract the notion of memory unit. Since 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. In order to better understand the semantics of the states with respect to the computational behavior of the modeled system, it is worth remarking that each ASM state can be characterized by one or more predicates over the states. More precisely, in (Bianchi et al., 2015) we defined a predicate ϕ over an ASM state s as a first-order formula defined over the locations in s, such that $s \vDash \phi$. Each predicate allows us to focus on the subsets of locations that turn out to be interesting for verification purposes.

The execution of an ASM consists in iterating computational steps. An ASM *computational step* in a given state consists in executing all rules whose condition is *true* in that state. Since different updates could affect the same location, it is necessary to impose a consistency requirement: a set of updates is said to be *consistent* if it contains no pair of updates referring to the same location. Therefore, if the updates are consistent, the result of a computational step is the transition of the machine from the current state to another. Otherwise, the computation does not yield a next state. An ASM *run* is so a (possibly infinite) sequence of steps: the computational step is iterated until no more rule is applicable.

The aforementioned notions refer to the so-called *basic* ASMs. A generalization of basic ASMs is represented by Distributed ASMs (DASMs), capable of capturing 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. A *distributed run* of a DASM is a partially ordered set of the runs of its ASMs: the underlying synchronization scheme reflects causal dependencies, determining which agent's move comes before — a move is a single computational step of an individual agent —, and is only restricted by the consistency condition, which is indispensable (Gurevich and Rosenzweig, 2000). Unfortunately, the notion of partially ordered run makes difficult to define univocally the global state of the computation of a DASM. Roughly speaking, a global state corresponds to the union of the signatures of each ASM together with interpretations of their functions. 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.

Finally, note that there is a distinction among functions, depending on the different roles that locations can assume in a given ASM. A primary distinction concerns *basic* functions, intended as elementary, and *derived* functions, whose values are defined in terms of other (basic or derived) functions, but neither the ASM nor the environment (more generally, other ASMs in the case of DASMs) can update them: they are automatically updated as a side effect of the updates over the functions from which they derive. In addition, basic functions are classified into *static*, whose values never change during a run, and *dynamic*, for which values change as a consequence of the updates executed by the ASM or by its environment. Furthermore, dynamic functions can be: *controlled*, if directly updated only by the ASM; *monitored*, if directly updated only by the environment and only read by the ASM; *shared*, which are both controlled and monitored; *out*, which are updated but never read by the ASM (they are monitored by the environment).

4 DINING PHILOSOPHERS

The Dining Philosophers problem, due to (Dijkstra, 1971), is one of the most illustrative examples in the field of concurrency for explaining deadlock (and starvation). Five philosophers are sitting around a table with a bowl of spaghetti in the middle. For the philosophers life consists only of two moments: thinking and eating, rigorously with two forks. More precisely, since each philosopher has a pair of a right fork and a left fork, (s)he behaves as follows: (s)he thinks till the right fork becomes available, grabs the right fork, waits till the left fork becomes available, grabs the left fork, 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 philosopher shares his/her right and left fork with his/her corresponding right and left neighbors, respectively.

4.1 ASM-based Model

The ASM-based model of the Dining Philosophers problem here described is elaborated with respect to both its general statement (Dijkstra, 1971) and its discussion in terms of ASMs (Börger and Stärk, 2003). Moreover, some aspects tailored to our purposes, such as the predicates over the states, are included. The problem can be simply modeled by a DASM *DiningPhilosophers* composed by a homogeneous set of agents: each of them behaves according to the same underlying ASM. More precisely, we have a set of $philosophers = \{p_1, \ldots, p_5\}$, representing the agents of the system, and a set of $forks = \{f_1, \ldots, f_5\}$, representing their shared resources.

The functions in each ASM signature Σ are:

- *rightFork*: *philosophers* → *forks*, static function indicating a philosopher's right fork;
- *leftFork*: *philosophers* → *forks*, static function indicating a philosopher's left fork;
- *owner*: *forks* → *philosophers* ∪ {*undef*}, dynamic shared function stating the current user of a fork;
- *hungry*: *philosophers* → *boolean*, dynamic shared function stating if a philosopher is hungry or not.

The main rule is:

```
DinPhilMainRule = {
      rightFork(p_1) := f_1
      leftFork(p_1) := f_5
      rightFork(p_2) := f_2
      leftFork(p_2) := f_1
      rightFork(p_3) := f_3
      leftFork(p_3) := f_2
      rightFork(p_4) := f_4
      leftFork(p_4) := f_3
   rightFork(p_5) := f_5
      leftFork(p_5) := p_4

      forall f_i in forks do
            owner(f_i) := undef

      forall p_i in philosophers do {
            Hungry(p_i)
            if hungry(p_i) then
                  PhilosopherProgram(p_i)
      }
}
```

Note that the **forall** construct is used for expressing the simultaneous execution of a set of updates and rules satisfying a given condition.

DinPhilMainRule assigns values to the static locations: each philosopher p_i has fork f_i on his/her right and fork f_{i-1} on his/her left, except for p_1 that has fork f_5 on his/her left. Then, *DinPhilMainRule* assigns *undef* to each shared location representing the holding of a fork. This means that, at the beginning of the computation, *DiningPhilosophers* is in the initial global state S_0 in which $\forall f_i \in forks$, $owner(f_i) = undef$, i.e. each philosopher does not hold any fork. Finally, *DinPhilMainRule* runs all the basic ASMs capturing the behavior of each philosopher. Note that the ASM related to the *i*-th philosopher, p_i, is run only if *hungry*(p_i) evaluates to *true*, i.e. if the philosopher is hungry. For the purposes of the present work, there is no need to further detail the *Hungry* rule: it can be conceived has an external entity that decides if p_i is hungry or not.

The basic ASM of the *i*-th philosopher, shown below, is composed by rules (r.1), (r.2) and (r.3):

```
PhilosopherProgram(p_i) = {
      if owner(rightFork(self)) = undef then          (r.1)
            owner(rightFork(self)) := self

      if owner(rightFork(self)) = self ∧
      owner(leftFork(self)) = undef then              (r.2)
            owner(leftFork(self)) := self

      if owner(rightFork(self)) = self ∧
      owner(leftFork(self)) = self then {
            Eat (self)                                (r.3)
            owner(rightFork(self)) := undef
            owner(leftFork(self)) := undef
      }
}
```

The computation of each ASM, representing p_i, can evolve through seven states:

- s_0: (*owner*(*rightFork*(**self**)) = *undef*) ∧ (*owner*(*leftFork*(**self**)) = *undef*);
- s_1: (*owner*(*rightFork*(**self**)) = *undef*) ∧ (*owner*(*leftFork*(**self**)) = p_{i-1} mod n);
- s_2: (*owner*(*rightFork*(**self**)) = p_{i+1} mod n) ∧ (*owner*(*leftFork*(**self**)) = *undef*);
- s_3: (*owner*(*rightFork*(**self**)) = p_{i+1} mod n) ∧ (*owner*(*leftFork*(**self**)) = p_{i-1} mod n);
- s_4: (*owner*(*rightFork*(**self**)) = **self**) ∧ (*owner*(*leftFork*(**self**)) = *undef*);
- s_5: (*owner*(*rightFork*(**self**)) = **self**) ∧ (*owner*(*leftFork*(**self**)) = p_{i-1} mod n);

- s_6: (*owner*(*rightFork*(**self**)) = **self**) ∧ (*owner*(*leftFork*(**self**)) = **self**).

As mentioned above, initially, each philosopher is in state s_0, i.e. for each philosopher both *owner*(*rightFork*(**self**)) and *owner*(*leftFork*(**self**)) evaluate to *undef*. The initial global state S_0 is then characterized by the composition of all the local states s_0. Note that the transition from one local state to another is not only determined by the own computation of p_i; it is also an effect of the updates executed by the neighboring philosophers over the shared locations.

These local states can be characterized by the following predicates over the states:

- `thinking`: ¬(*owner*(*rightFork*(**self**)) = **self** ∨ *owner*(*leftFork*(**self**)) = **self**). The philosopher is thinking, so (s)he is waiting for the right fork to become available. This predicate holds in states from s_0 to s_3;
- `holdingRightFork`: *owner*(*rightFork*(**self**)) = **self** ∧ ¬(*owner*(*leftFork*(**self**)) = **self**). The philosopher is holding his/her right fork, so (s)he is waiting for the left fork to become available. This predicate holds in states s_4 and s_5;
- `eating`: *owner*(*rightFork*(**self**)) = **self** ∧ *owner*(*leftFork*(**self**)) = **self**. The philosopher has obtained both forks, so (s)he is eating. This predicate only holds in state s_6.

In the model above, rule (r.3) states that each philosopher, after obtaining both forks, executes the *Eat* rule, then releases them. Since the eating process is outside the resource allocation problem driving the risk of deadlock, the *Eat* rule does not need to be further specified.

4.2 Analysis of the Model

In order to verify that *DiningPhilosophers* DASM is affected by the risk of deadlock, let's analyze the four necessary conditions upon the model.

Mutual Exclusion. The claiming of exclusive control of the resources is expressed by the grabbing of each fork. In *DiningPhilosophers* this is accomplished by the execution of rules (r.1) and (r.2) for the right and the left fork, respectively. Each rule can be executed only if the respective fork is available, i.e. if *owner*(*right*/*leftFork*(**self**) = *undef*). Note that when the *i*-th philosopher p_i grabs a fork, that fork cannot be accessed by the respective neighboring philosopher. In fact, if two neighboring philosophers p_i and p_{i+1} simultaneously try to access the same fork *f*, a consistency violation occurs. More precisely, if *f* is the right fork for p_i and the left fork for p_{i+1}, if p_i executes its own rule (r.1) and p_{i+1} executes its own rule (r.2) at the same time, then the two rules would access the same location dealing with the ownership of the fork, so producing an inconsistency upon it. Therefore, the mutual exclusion condition holds in *DiningPhilosophers*.

Resource Holding. When p_i has grabbed a fork, (s)he waits until (s)he can grab the second one; then the forks are released only after the completion of the eating process. In *DiningPhilosophers* this is expressed by the update of rule (r.3), which can be executed only when both forks have been grabbed by p_i, i.e. when *owner*(*right*/*leftFork*(**self**)) = **self**. Therefore, the resource holding condition holds in *DiningPhilosophers*.

No Preemption. No one (or nothing) can forcibly remove a fork from a philosopher that is holding it. In *DiningPhilosophers* this is expressed by the lack of any rule allowing a philosopher to become the owner of a fork which is held by another philosopher. In fact, all updates expressing the grabbing of the forks (*owner*(*right*/*leftFork*(**self**)) := **self**) only appear in rules guarded by conditions stating the availability of the forks (if *owner*(*right*/*leftFork*(**self**)) = *undef*). Therefore, the no preemption condition holds in *DiningPhilosophers*.

Circular Wait. Due to the configuration of the static locations in the initial global state represented by the main rule *DinPhilMainRule*, the circular chain is set when all agents are executing rule (r.2). In this way, each p_i waits for the fork held by p_{i-1} mod *n*. Therefore, the circular wait condition holds in *DiningPhilosophers*.

In conclusion, all the four conditions could hold separately: if they occur simultaneously a risk of deadlock exists. Note that in *DiningPhilosophers* DASM, in every run in which the *PhilosopherProgram*(p_i) is invoked for the *i*-th philosopher, the mutual exclusion condition always holds. Both the resource holding and no preemption conditions hold when p_i grabs a fork. The circular wait condition holds when each p_i grabs its right fork. Therefore, in the latter case all conditions are simultaneously satisfied. This is formally stated by the following:

Theorem. *There exists* (*at least*) *an admissible run* M *of* DiningPhilosophers *such that* DiningPhilosophers *is deadlocked*.

Proof. In the case that for all philosophers *hungry*(p_i) evaluates to *true*, then each p_i executes its own *PhilosopherProgram*(p_i), so each p_i executes its own rule (r.1). Therefore, the four necessary conditions for deadlock hold simultaneously. Let *M* be a run of *DiningPhilosophers* such that all p_i simultaneously execute their own rule (r.1). Then, *DiningPhilosophers* reaches a global state S_k, in which all the conditions guarding every rule evaluate to *false* for each *PhilosopherProgram*(p_i), so no further rule can be executed. Therefore, *DiningPhilosophers* is deadlocked. □

In S_k all philosophers indefinitely wait for each other to release the possessed fork, i.e. the local state of each (*ASM*, p_i) satisfies the `holdingRightFork` predicate over the states, so $\forall\, p_i \in$ *philosophers*, *owner*(*rightFork*(p_i)) = p_i.

5 DISCUSSION

The analysis of the model allows us to derive some considerations to address deadlock issues with ASMs. Preliminarily, it is worth noting that the classic necessary conditions for deadlock implicitly apply to multi-process systems with shared resources. In other words, a deadlock emerges from the interaction among multiple processes when resources are shared among them. As consequence, we must first restrict our focus only on DASMs. In fact, as stated in (Gurevich, 1995), only DASMs allow for the representation of multi-agent systems: each agent models the behavior of a single process of the system under study.

Secondly, for what concerns resources, they can be modeled by the concept of function. Let's recall that the ASM functions belong to particular classes. In accordance with this classification, static functions surely do not impact deadlock, because their values never change during a run. Controlled functions can also be excluded in that their values can be managed by an ASM because set inside it. Although monitored functions indicate that the behavior of an ASM is affected by the other agents, they are not updatable by the ASM they belong to, so they cannot represent a resource that the ASM shares with its environment. Shared functions, as the name suggest, surely can model a shared resource: indeed, they are directly updatable by the rules of the ASM they belong to and by the environment, and can be read by both. Out functions are not involved in a deadlock, because their values are produced by the ASM, but never used. Finally, more detailed analysis should be executed when derived functions appear. In fact, since they cannot be managed inside the ASM, because their values depend on other functions, the latter must be investigated. More precisely, a derived function must be taken into account if it is defined on a shared function or (recursively) on a derived function defined on a shared function. For the sake of simplicity, in the following we only consider shared functions.

The previous observations suggest a provisional generalization of the necessary conditions for deadlock in terms of ASMs.

The Mutual Exclusion Lemma (Börger and Schewe, 2015) expresses that, thanks to the partial order of moves guaranteed by the concept of distributed run, the possible updates of two agents over a same shared location at the same time is never allowed. In other words, in general in a DASM, the access to shared resources is exclusive, so no further scheduling policy is needed.

Therefore, we can suspect that a deadlock can occur in a DASM model when only the following three necessary conditions hold simultaneously:

Resource Holding. There is (at least) one rule *r* whose update concerns the updating of a shared location *loc* to **self**. It is in the form **if** *condition* **then** *loc* := **self**. This rule represents the action of holding a shared resource by a process: we say that the agent which executes *r holds* the shared location *loc*. Moreover, there is (at least) one rule *r'*: (*i*) whose condition evaluates to *true* only if *r* has been previously executed; and (*ii*) whose update concerns the updating of a new shared location *loc'* to **self**. It is in the form **if** *condition* ∧ (*loc* = **self**) **then** *loc'* := **self**. This rule represents the waiting for a requested resource while another resource has already been allocated to the process: we say that the agent which is unable to execute *r' waits for* the shared location *loc'*. Note that both in *r* and *r' condition* is to be intended as a first-order formula of arbitrary complexity. Moreover, note that the waiting can be semantically represented by a predicate over the states ϕ in the general form *loc* = **self** ∧ ¬(*loc'* = **self**).

No Preemption. The conditions guarding the above mentioned rules *r* and *r'* evaluate to *true* only if *loc* and *loc'* evaluate to *undef*. Therefore, they are in the form **if** *condition* ∧ (*loc* = *undef*) **then** *loc* := **self** and **if** *condition* ∧ (*loc* = **self**) ∧ (*loc'* = *undef*) **then** *loc'* := **self**, respectively. More generally, no rule in the model updates to **self** a shared location *loc* if *loc* does not evaluate to *undef*. This condition captures the impossibility to allocate a resource, already allocated to a process, to another process.

Circular Wait. A circular chain of *agents* $\{a_1, \ldots, a_n\}$ exists such that a_i is *waiting for* a shared location *held* by a_{i+1} mod n. Semantically, a circular chain arises when (at least) a global state S_k, resulting by the composition of the local states $s_1, \ldots, s_n$, each satisfying the above mentioned predicate over the states ϕ, is reachable from the initial global state S_0.

A DASM is then deadlock-free if any of the three necessary conditions for deadlock is denied. Resource holding can be denied by imposing each process to request all the resources it needs at once. In ASM terms, rules r and r' above must be unified in a single rule whose update concerns the updating of all the shared locations each ASM waits for at once. More generally, all the shared locations an agent waits for must be updated during the execution of a unique rule. This rule can be conceived to be in the general form **if** *condition* **then** loc_1 := **self**, …, loc_n := **self**, where $loc_1, \ldots, loc_n$ represent all the resources the agent needs. Note that, in this way, the predicate ϕ, representing the waiting for the remaining resources, can no longer be satisfied.

Instead, the following strategy can be adopted in order to enable preemption: if a process holding a resource is denied a further request, that process must release the already acquired resource. If necessary, the process can request the resources it needs at a later time. A way to achieve this can be adding the following rule: **if** *condition* ∧ (*loc* = **self**) ∧ ¬(*loc'* = *undef*) **then** *loc* := *undef*. This rule allows an agent to release the already acquired *loc* if *loc'* is not set to *undef*.

Finally, circular wait can be denied by imposing a linear order of requests so that processes are forced to require the resources they need in that order. According to Lamport's Bakery algorithm (Lamport, 1974), a possible solution is adding to the model a *scheduler* agent that decides the order in which the disputed shared location can be updated by the other agents. For example, the *scheduler* agent can assign a ticket to each process, compare tickets and let the process with the smallest one to access the resources. In this way, all processes access the resources alternately, so the resource allocation graph can never have a cycle.

6 CONCLUSIONS

In order to investigate the deadlock-freedom property, the study here presented has focused on modelling and analyzing the Dining Philosophers problem using Abstract State Machines. It is worth remarking that, albeit simple, the Dining Philosophers problem is adequately general in that it enables an abstract description of deadlock that can be further refined to any specific domain.

The analysis of the model has allowed us to provide a preliminary reformulation of the classic necessary conditions for deadlock in terms of ASMs. The results obtained are encouraging for the purposes of our research because an entirely operational ASM-based characterization of deadlock could allow modelers to treat the analysis of deadlock inside the ASM framework before adopting it in conjunction with *hybrid* model checking approaches. In fact, thanks to such a characterization, developers can recognize the risk of deadlock in an ASM model of a system, so they can re-model it in advance, before its development, with evident effort savings.

Nevertheless, our approach presents drawbacks. As other manual techniques, it is human-based, so is error-prone and requires expertise in order to find an appropriate abstraction of the system to be verified. In other words, any analysis is as good as the model is. Furthermore, because of decidability issues, it cannot be completely automatized, even if automatic tools can support it.

Future developments of our research should generalize the findings of this paper with the aim to formally prove the necessary conditions that enable deadlock inside ASMs.

Moreover, it is worth noting that, in this paper, we have only focused on distributed resource deadlocks. Therefore, future developments should also consider both distributed communication deadlocks and single deadlocks. To this end, it is worth remarking that monitored functions can represent the receipt of a message and the occurrence of an external event, so they must be taken into account in deadlock analysis. Moreover, the focus can no longer be restricted only on DASMs, but also basic ASMs interacting with the environment must be studied.

REFERENCES

Arcaini, P., Gargantini, A., Riccobene, E., 2010. Asmeta-SMV: A Way to Link High-Level ASM Models to Low-Level NuSMV Specifications, *Proc. of the 2nd International Conference on Abstract State Machines, Alloy, B and Z*, pp. 61–74.

Bianchi, A., Pizzutilo, S., Vessio, G., 2015. Applying Predicate Abstraction to Abstract State Machines, *Enterprise, Business-Process and Information Systems Modeling*, *LNBIP 214*, *Springer*, pp. 283–292.

Bianchi, A., Pizzutilo, S., Vessio, G., 2016. Reasoning on Starvation in AODV using Abstract State Machines, *Journal of Theoretical and Applied Information Technology*, 84(1), pp. 140–149.

Bingham, B., Bingham, J., Erickson, J., Greenstreet, M., 2013. Distributed Explicit State Model Checking of Deadlock Freedom, *Proc. of the 25th International Conference on Computer Aided Verification*, pp. 235–241.

Blass, A., Gurevich, Y., 2003. Abstract State Machines Capture Parallel Algorithms, *ACM Transactions on Computational Logic*, 4(4), pp. 578–651.

Börger, E., Schewe, K.D., 2015. Concurrent abstract state machines, *Acta Informatica*, pp. 1–24.

Börger, E., Stärk, R., 2003. Abstract State Machines: A Method for High-Level System Design and Analysis, *Springer-Verlag*.

Brookes, S.D., Roscoe, A.W., 1991. Deadlock analysis in networks of communicating processes, *Distributed Computing*, 4(4), pp. 209–230.

Coffman, E.G., Elphick, M.J., Shoshani, A., 1971. System Deadlocks, *Computing Surveys*, 3(2), pp. 67–78.

Del Castillo, G., Winter, K., 2000. Model Checking Support for the ASM High-Level Language, *Proc. of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 331–346.

Dershowitz, N., 2013. The Generic Model of Computation, *Electronic Proceedings in Theoretical Computer Science*.

Dijkstra, E.W., 1968. Cooperating sequential processes, Genuys, F., ed., *Programming Languages*, *Academic Press*, pp. 43–112.

Dijkstra, E.W., 1971. Hierarchical Ordering of Sequential Processes, *Acta Informatica*, 1(2), pp. 115–138.

Farahbod, R., Gervasi, V., Glässer, U., 2007. CoreASM: An Extensible ASM Execution Engine, *Fundamenta Informaticae*, 77(1-2), pp. 71–103.

Farahbod, R., Glässer, U., Ma, G., 2007. Model Checking CoreASM Specifications, Prinz, A., ed., *14th International ASM Workshop*.

Gabrisch, W., Zimmermann, W., 2012. A Hoare-Style Verification Calculus for Control State ASMs, *Proc. of the 5th Balkan Conference on Informatics*, pp. 205–210.

Glausch, A., Reisig, W., 2009. An ASM-Characterization of a Class of Distributed Algorithms, Abrial, J.R., Glässer, U., eds., *Rigorous Methods for Software Construction and Analysis*, pp. 50–64.

Gurevich, Y., 1995. Evolving Algebras 1993: Lipari Guide, Börger, E., ed., *Specification and Validation Methods*, *Oxford University Press*, pp. 9–36.

Gurevich, Y., 2000. Sequential Abstract State Machines Capture Sequential Algorithms, *ACM Transactions on Computational Logic*, 1(1), pp. 77–111.

Gurevich, Y., Rosenzweig, D., 2000. Partially ordered runs: A case study. In: *Abstract State Machines - Theory and Applications*, pp. 131–150.

Holt, R.C., 1972. Some Deadlock Properties of Computer Systems, *ACM Computing Surveys*, 4(3), pp. 179–196.

Kameda, T., 1980. Testing Deadlock-Freedom of Computer Systems, *Journal of the ACM*, 27(2), pp. 270–280.

Knapp, E., 1987. Deadlock detection in distributed databases, *ACM Computing Surveys*, 19(4), pp. 303–328.

Lamport, L., 1974. A New Solution of Dijkstra's Concurrent Programming Problem, *Communications of the ACM*, 17(8), pp. 453–455.

Padovani, L., 2014. Deadlock and lock freedom in the linear π-calculus, *Proc. of CSL-LICS '14*, article no. 72.

Rafe, V., Doostali, S., 2012. ASM2Bogor: An approach for verification of models specified through Asmeta language, *Journal of Visual Languages and Computing*, 23(5), pp. 287–298.

Reisig, W., 2003. The Expressive Power of Abstract State Machines, *Computing and Informatics*, 22, pp. 209–219.

Singhal, M., 1989. Deadlock Detection in Distributed Systems, *Computer*, 22(11), pp. 37–48.

Spielmann, M., 1999. Automatic Verification of Abstract State Machines, *Proc. of the 11th International Conference on Computer Aided Verification*, pp. 431–442.

Stärk, R.F., Nanchen, S., 2001. A Logic for Abstract State Machines, *Journal of Universal Computer Science*, 7(11), pp. 981–1006.