Modelling and Analyzing Adaptive Self-Assembly Strategies with Maude *

Roberto Bruni[1], Andrea Corradini[1], Fabio Gadducci[1], Alberto Lluch Lafuente[2], and Andrea Vandin[2]

[1] Dipartimento di Informatica, Università di Pisa, Italy
{bruni,andrea,gadducci}@di.unipi.it
[2] IMT Institute for Advanced Studies Lucca, Italy
{alberto.lluch,andrea.vandin}@imtlucca.it

Abstract. Building adaptive systems with predictable emergent behavior is a challenging task and it is becoming a critical need. The research community has accepted the challenge by introducing approaches of various nature: from software architectures, to programming paradigms, to analysis techniques. We recently proposed a conceptual framework for adaptation centered around the role of control data. In this paper we show that it can be naturally realized in a reflective logical language like Maude by using the Reflective Russian Dolls model. Moreover, we exploit this model to specify and analyse a prominent example of adaptive system: robot swarms equipped with obstacle-avoidance self-assembly strategies. The analysis exploits the statistical model checker PVesta.

Keywords: Adaptation, autonomic, self-assembly, swarms, ensembles, Maude

1 Introduction

How to engineer autonomic system components so to guarantee that certain goals will be achieved is one of todays' grand challenges in Computer Science. First, autonomic components run in unpredictable environments, hence they must be engineered by relying on the smallest possible amount of assumptions, i.e. as *adaptive* components. Second, no general formal framework for adaptive systems exists that is widely accepted. Instead, several adaptation models and guidelines are presented in the literature that offer ad hoc solutions, often tailored to a specific application domain or programming language. Roughly, there is not even a general agreement about what "adaptation" is. Third, it is not possible to mark a b/w distinction between failure and success, because the randomized behaviour of the system prevents an absolute winning strategy to exist. Fourth, efforts spent in the accurate analysis of handcrafted adaptive components are unlikely to pay back, because the results are scarcely reusable when the components software is frequently updated or extended with new features.

* Research supported by the European Integrated Project 257414 ASCENS.

We address here some of the above concerns, presenting the methodology we have devised for prototyping well-engineered self-adaptive components. Our case study consists of modeling and analyzing self-assembly strategies of robots whose goal is crossing a hole while navigating towards a light source. We specified such robots with Maude, exploiting on one hand the Reflective Russian Dolls (RRD) model [21] and on the other hand the conceptual framework we proposed in [6], which provides simple but precise guidelines for a clean structuring of self-adaptive systems. We report also on the results of the analysis of our model using PVesta [2], as well as on relevant aspects of our experience using it.

When is a software system adaptive? Self-adaptation is a fundamental feature of autonomic systems, that can specialize to several other so-called self-* properties (like self-configuration, self-optimization, self-protection and self-healing, as discussed e.g. in [10]). Self-adaptive systems have become a hot topic in the last decade: an interesting taxonomy of the concepts related to self-adaptation is presented in [18]. Several contributions have proposed reference models for the specification and structuring of self-adaptive software systems, ranging from architectural approaches (including the well-known MAPE-K [9, 10, 12], FORMS [23], the adaptation patterns of [7], and the already mentioned RRD [21]), to approaches based on model-based development [24] or model transformation [11], to theoretical frameworks based on category theory [17] or stream-based systems [5].

Even if most of those models have been fruitfully adopted for the design and specification of interesting case studies of self-adaptive systems, in our view they missed the problem of characterizing *what is adaptivity* in a way that is independent of a specific approach. We have addressed this problem in [6], where we have proposed a very simple criterion: a software system is *adaptive* if its behaviour depends on a precisely identified collection of *control data*, and such control data can be modified at run time. We discuss further this topic in §3.

Is Maude a convenient setting to study self-adaptation? A "convenient" framework must provide a reusable methodology for modelling self-adaptive systems independently of their application domain together with a flexible analysis toolset to investigate formal properties of the semantics of such systems. There are several reasons why we think that Maude [8] is a good candidate. First, the versatility of rewrite theories can offer us the right level of abstraction for addressing the specification, modelling and analysis of self-adaptive systems and their environments within one single coherent framework. Second, since Maude is a rule-based approach, the control-data can be expressed naturally as a sub-set of the available rules and the reflection capability of Maude can be exploited to express control-data manipulation via ordinary rewrite rules, along the so-called *tower of reflection* and its modular realization as the RRD approach [14]. Third, the conceptual framework for adaptation described in [6], to be further elaborated in §4, facilitates early and rapid prototyping of self-adaptive systems, to be simulated. Fourth, the formal analysis toolset of Maude can support simulations and analysis over the prototypes. In particular, given the probabilistic nature of adaptive systems, where absolute guarantees cannot be proved, we think that

the statistical model checker PVesta [2] is useful, because it allows to conduct analysis that are parametric w.r.t. the desired level of statistical confidence.

Pragmatically, the possibility to rapidly develop and simulate self-adaptive systems and to compare the behaviour emerging from different adaptation strategies at the early stages of software development is very important for case studies like the robotic scenario described in the next paragraphs. Indeed, such physical devices require specialized programming skills and their experimentation in real world testing environments involves long time consumption (6 hours or more for each run) and only a limited number of pieces is available (around 25 units) because their maintenance is expensive. Also, their hardware (both mechanic and electronic parts) and software are frequently updated, making it harder to build, to maintain and to rely on sophisticated simulators that can take as input exactly the same code to be run on the robots. Even when this has been attempted, the tests conducted on the real systems can differ substantially from the simulated runs. Thus, early simulation on prototypes can at least speed-up debugging and dispense the programmers from coding lowest-performance strategies.

Synopsis. In §2 we present the case study analysed in this paper. In §3 we summarize the conceptual framework for adaptation proposed in [6], along which we design adaptive systems in Maude. The general guidelines and principles to be exploited in Maude for modelling self-adaptive systems are described in §4, together with the software architecture used to realize our conceptual framework. In §5 we illustrate the concrete implementation of the case study, while the experimentations are described in §6; for the sake of presentation, we focus on one of the self-assembly strategies. Some concluding remarks and ongoing research avenues are discussed in §7.

We assume the reader to have some familiarity with the Maude framework.

2 Case Study: Self-Assembly Robot Swarms

Self-assembly robotic systems are formed by independent robots capable to connect physically when the environment prevents them from reaching their goals individually. Self-assembly is a contingency mechanism for environments where versatility is a critical issue and the size and morphology of the assembly cannot be known in advance. Thus, self-assembly units must be designed in a modular way and their logic must be more sophisticated than, say, that of cheaper pre-assembled units. Such features make the self-assembly robot swarm a challenging scenario to engineer.

In [16], different self-assembly strategies are proposed to carry out tasks that range from hill-crossing and hole-crossing to robot rescue: case by case, depending e.g. on the steep of the hill, the width of the hole, the location of the robot to be rescued, the robots must self-assemble because incapable to complete the task individually. We focus on the *hole-crossing scenario* as a running case study, where "the robots in the swarm are required to cross a hole as they navigate to a light source" and depending on the width of the hole "a single unit by itself will fall off into the crevice, but if it is a connected body, falling can be prevented".

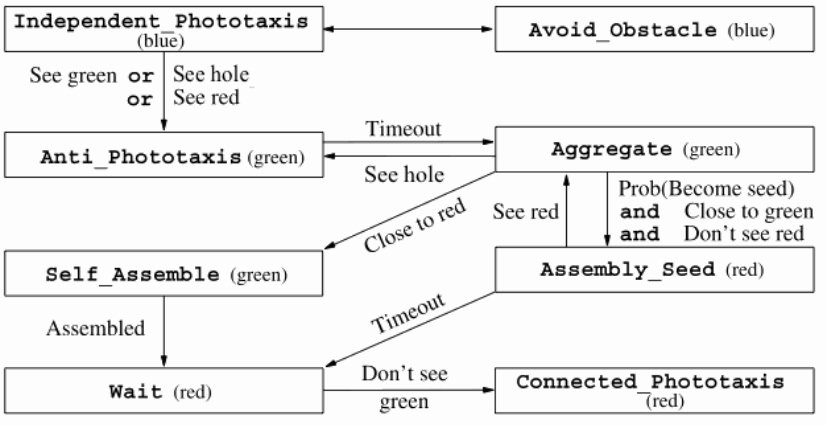

Fig. 1. Excerpt of the basic self-assembly response strategy (borrowed from [16]).

The experiments in [16] were conducted on the SWARM-BOT robotic platform [15], whose constituents are called s-bots (see Fig. 6, bottom right). Each s-bot has a traction system that combines tracks, wheels and a motorised rotation system, has several sensors (including infra-red proximity sensors to detect obstacles, ground facing proximity sensors to detect holes, and a 360 degrees view thanks to a camera turret), and is surrounded by a transparent ring that contains eight RGB colored LEDs (Light Emitting Diodes) distributed uniformly around the ring. The LEDs can provide some indications about the internal state of the s-bot to (the omni-directional cameras of) nearby s-bots. For example, the green color can be used to signal the willingness to connect to an existing ensemble, and the red color can be used for the willingness to create a new assembly. The ring can also be grasped by other s-bots thanks to a gripper-based mechanism.

Roughly, the strategies described in [16] are: (i) the *independent execution* strategy, where s-bots move independently one from the other and never self-assemble; (ii) the *basic self-assembly response* strategy (see below), where each s-bot moves independently (blue light) until an obstacle is found, in which case it tries to aggregate (green light) to some nearby assembly, if some available, or it becomes the *seed* of a new assembly (red light); (iii) the *preemptive self-assembly* strategy, where the s-bots self-assemble irrespectively of the environment and not by emergency as in the basic self-assembly response; (iv) the *connected coordination* strategy, where the sensing and actuation of assembled robots is coordinated according to a leader-follower architecture.

The experiments were conducted with different strategies in different scenarios (with holes of different dimension and random initial positions of the s-bots) and repeated for each strategy within each scenario (from a minimum of 20 times and 2 s-bots to a maximum of 60 times and 6 s-bots). Videos of the experiments described in [16] are linked from the web page describing our Maude implementation: `http://sysma.lab.imtlucca.it/tools/ensembles`.

Basic self-assembly response strategy. We describe here the *basic self-assembly strategy* of [16], which is the strategy on which we will focus in the rest of the paper. The finite state machine of the strategy is depicted in Fig. 1. Each state contains its name and the color of the LEDs turned on in that state, while transitions are labelled with their firing condition.

This controller is executed independently in each individual s-bot (a concrete one in [16], or a software abstraction in this work). In the starting state (`Independent_Phototaxis`) each s-bot turns on its blue LEDs, and navigates towards the target light source, avoiding possible obstacles (e.g. walls or other robots). If an s-bot detects a hole (through its infra-red ground sensors), or sees a green or red s-bot, then it switches to state `Anti_Phototaxis`, i.e. it turns on its green LEDs and retreats away from the hole.

After the expiration of a timeout, the s-bot passes to state `Aggregate`: it randomly moves searching for a red (preferably) or a green s-bot. In case it sees a red s-bot, it switches to state `Self_Assemble`, assembles (grabs) to the red s-bot, turns on its red LEDs and switches to state `Wait`. If instead it sees a green s-bot, with probability `Prob(Become seed)` it switches to state `Assembly_Seed`, turns on its red LEDs, and becomes the seed of a new ensemble. Once in state `Assembly_Seed`, the s-bot waits until a timeout expires and switches to state `Wait`, unless it sees another red s-bot, in which case it reverts to state `Aggregate`. Once no green s-bots are visible, assembled "waiting" s-bots switch to state `Connected_Phototaxis` and navigate to the light source.

3 A Framework for Adaptation

Before describing how we modeled and analysed the scenario we just presented, let us explain some guidelines that we followed when designing the system. The main goal was to develop a software system where the adaptive behaviour of the robots is explicitly represented in the system architecture. To this aim, we found it necessary to first understand *"when is a software system adaptive"*, by identifying the features distinguishing such systems from ordinary ("non-adaptive") ones.

We addressed this problem in [6], proposing a simple structural criterion to characterize adaptivity. Oversimplifying a bit, according to a common *black-box* perspective, a software system is "self-adaptive" if *it can modify its behaviour as a reaction to a change in its context of execution*. Unfortunately this definition is hardly usable: accordingly to it almost any software system can be considered self-adaptive. Indeed, any system can *modify its behaviour* (e.g. executing different instructions, depending on conditional statements) as a *reaction to a change in the context of execution* (like the input of a data from the user).

We argue that to distinguish situations where the modification of behaviour is part of the application logic from those where they realize the adaptation logic, we must follow a *white-box* approach, where the internal structure of a system is exposed. Our framework requires to make explicit that the behavior of a component depends on some well identified *control data*. We define *adaptation* as the *run-time modification of the control data*. From this definition we derive

that a component is called *adaptable* if it has a clearly identified collection of control data that can be modified at run-time. Further, a component is *adaptive* if it is adaptable and its control data are modified at run-time, at least in some of its executions; and it is *self-adaptive* if it can modify its own control data.

Under this perspective, and not surprisingly, any computational model or programming language can be used to implement an adaptive system, just by identifying the part of the data governing the behavior. Consequently, the nature of control data can greatly vary depending on the degree of adaptivity of the system and on the computational formalisms used to implement it. Examples of control data include configuration variables, rules (in rule-based programming), contexts (in context-oriented programming), interactions (in connector-centered approaches), policies (in policy-driven languages), aspects (in aspect-oriented languages), monads and effects (in functional languages), and even entire programs (in models of computation exhibiting higher-order or reflective features).

In [6] we discussed how our simple criterion for adaptivity can be applied to several of the reference models we mentioned in the introduction, identifying what would be a reasonable choice of control data in each case. Interestingly, in most situations the explicit identification of control data has the effect of revealing a precise interface between a managed component (mainly responsible for the application logic) and a control component (encharged of the adaptation logic). As a paradigmatical example, consider the MAPE-K architecture [9], according to which a self-adaptive system is made of a component implementing the application logic, equipped with a control loop that *monitors* the execution through sensors, *analyses* the collected data, *plans* an adaptation strategy, and finally *executes* the adaptation of the managed component through effectors; all the phases of the control loop access a shared *knowledge* repository. Applying our criterion to this model suggests a natural choice for the control data: these must include the data of the managed component that are modified by the execute phase of the control loop. Clearly, by our definitions the managed component is adaptive, and the system made of both component and control loop is self-adaptive.

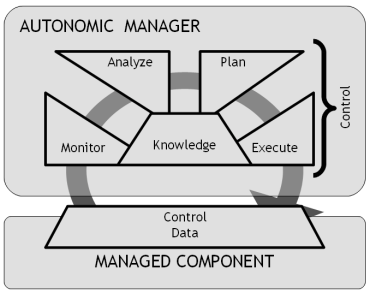

Fig. 2. Control data in MAPE-K.

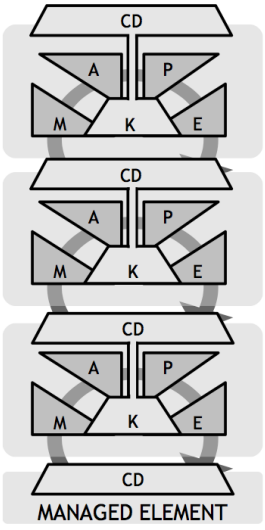

Fig. 3. Tower of adaptation.

The construction can be iterated, as the control loop itself could be adaptive. Think e.g. of an adaptive component which follows a plan to perform some tasks.

This component might have a manager which devises new plans according to changes in the context or in the component's goals. But this planning component might itself be adaptive, where some component controls and adapts its planning strategy, for instance determining the new strategy on the basis of a tradeoff between optimality of the plans and computational cost. In this case the manager itself (the control loop) should expose its control data (conceptually part of its knowledge repository) in its interface. In this way, the approach becomes compositional in a layered way, which allows one to build towers of adaptive components (Fig. 3) as we do in §5 and §6 for robot prototypes.

4 Adaptivity in Maude

We argue here the suitability of Maude and rewriting logic as a language and a model for adaptivity (§4.1), we describe a generic architecture for developing adaptive components in Maude (§4.2) and we show that it conforms to well-assessed conceptual models for adaptivity, including our framework (§4.3).

4.1 Maude, Logical Reflection and Adaptivity

As argued in [14], Rewriting Logic (RL) is well-suited for the specification of adaptive systems, thanks to its reflective capabilities. The reflection mechanism yields what is called the *tower of reflection*. At the ground level, a rewrite theory $\mathcal{R}$ (e.g. a software module) allows to infer a computation step $\mathcal{R} \vdash t \to t'$ from a term t (e.g. a program state) to a term t'. A universal theory $\mathcal{U}$ lets infer the computation $\mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{t}) \to (\overline{\mathcal{R}}, \overline{t'})$ at the "meta-level" where theories and terms are meta-represented as terms. The process can be repeated as $\mathcal{U}$ itself is a rewrite theory. This mechanism is efficiently supported by Maude and fostered many meta-programming applications like analysis and transformation tools. Since a theory can be represented by a term, it is also possible to specify *adaptation rules* that change the (meta-representation of the) theory, as in $r \vdash (\overline{\mathcal{R}}, \overline{t}) \to (\overline{\mathcal{R}'}, \overline{t'})$, so that the reduction continues with a different set of rules $\mathcal{R}'$.

The reflection mechanism of RL has been exploited in [14] to formalize a model for distributed object reflection, suitable for the specification of adaptive systems. Such model, called Reflective Russian Dolls (RRD), has a structure of layered configurations of objects, where each layer can control the execution of objects in the lower layer by accessing and executing the rules in their theories, possibly after modifying them, e.g. by injecting some specific adaptation logic in the wrapped components. It is worth stressing that logical reflection is only one possible way in which a layer can control the execution of objects of the lower level: objects within a layer interact via message passing, thus objects of the higher layer might intercept messages of the lower level, influencing their behaviour. But even if the resulting model is still very expressive, some form of reflection seems to be very convenient, if not necessary, to implement adaptivity. This is clearly stated in [14] and at a more general level in [3], where *(computational) reflection is promoted as a necessary criterion for any self-adaptive software system.*

The RRD model has been exploited for modeling policy-based coordination [21] and for the design of PAGODA, a modular architecture for specifying autonomous systems [22].

4.2 Generic Architecture

This section describes how we specialize the RRD architecture for modeling adaptive components. We focus on the structure of the layers and on the interactions among them, abstracting from the details of our case study, discussed in §5.

Intra-layer architecture Each layer is a component having the structure illustrated in Fig. 4. Its main constituents are: *knowledge* (K), *effects* (E), *rules* (R) and *managed component* (M). Some of them are intentionally on the boundary of the component, since they are part of its interface: knowledge and effects act respectively as input and output interfaces, while rules correspond to the component's control interface. Therefore we will consider the rules R as the *control data* of a layer.

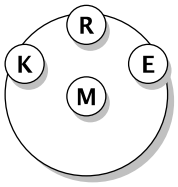

Fig. 4. Intra-layer.

The managed component is a lower-level layer having the same structure: clearly, this part is absent in the innermost layer. The knowledge represents the information available in the layer. It can contain data that represent the internal state or assumptions about the component's surrounding environment. The effects are the actions that the component is willing to perform on its enclosing context. The rules determine which effects are generated on the basis of the knowledge and of the interaction with the managed component. Typical rules update the knowledge of the managed component, execute it and collect its effects. In this case the layer acts as a sort of interpreter. In other cases rules can act upon the rules of the managed component, modifying them: since such rules are control data, the rules modifying them are *adaptation rules* according to §3.

Inter-layer architecture Layers are organized hierarchically: each one contains its knowledge, effects, rules and, in addition, the managed underlying layer (see the leftmost diagram of Fig. 5). The outermost layer interacts with the environment: its knowledge represents the perception that the adaptive component has of the environment, while its effects represent the actions actually performed by the component. Each layer elaborates its knowledge and propagates it to the lower one, if any. In general, while descending the hierarchy, the knowledge becomes simpler, and the generated effects more basilar. Similarly to layered operating systems, each layer builds on simpler functionalities of the lower one to compute more complex operations.

The diagram in the middle of Fig. 5 shows the control and data flow of ordinary behavior (without adaptations). Knowledge is propagated down to the core (layer 0) and the effects are collected up to the skin (layer 2). This flow of information is governed by the rules. Knowledge and effects are subject

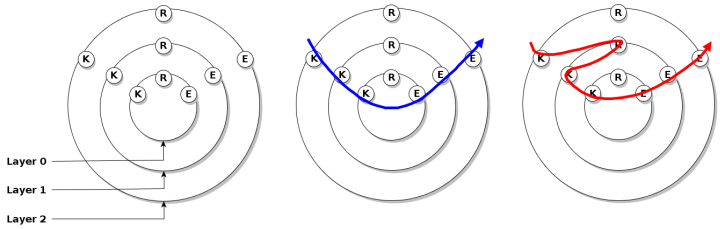

Fig. 5. Inter-layer architecture (left), ordinary flow (center), adaptation flow (right).

to modifications before each propagation. For example, layer 2 may decide to propagate to layer 1 only part of the knowledge perceived from the environment, possibly after pre-processing it. Symmetrically, layer 1 may decide to filter part of the effects generated by layer 0 before the propagation to layer 2, for example discarding all those violating some given constraints.

The rightmost diagram of Fig. 5 corresponds to a phase of adaptation. Here the outermost layer triggers an adaptation at layer 1. This can be due to some conditions on the knowledge of layer 2 or to the status of the managed component (layer 1). The result is that the rules of layer 2 change (among other things) the rules of layer 1 (as shown by the arrow crossing the corresponding R attribute).

4.3 Generic Architecture and Adaptation Frameworks

Let us relate the generic architecture just presented with some general frameworks used for modeling adaptive systems. As suggested in §3, we identified explicitly the control data of each layer, namely, its set of rules: this will allow us to distinguish the adaptation behaviour from the standard computations of the system.

Our architecture is a simplified version of the RRD of [14], because each layer is a single object rather than a proper configuration. The interaction between a layer and its managed component is realized both with logical reflection and with access to shared data (knowledge and effects). Further, there is a clear correspondence between the reflective tower of the RRD model and the adaptation tower discussed in §3, as depicted in Fig. 6, showing that the rules of each layer implement the MAPE control loop on the lower layer. Moreover, the generic architecture imposes the encapsulation of all components of the tower, apart from the robot itself. This offers several advantages: (i) management is hierarchical (e.g. self- or mutually-managing layers are excluded); and (ii) at each level in the hierarchy the adaptation logic of the underlying layer is designed separately from the execution of basic functionalities, that are delegated to lower layers.

5 Concrete Architecture and Case Study Implementation

This section instantiates the generic architecture shown in §4.2 to our case study (§5.1), and presents some relevant details of its implementation (§5.2). We will call s-bots simply robots in the following.

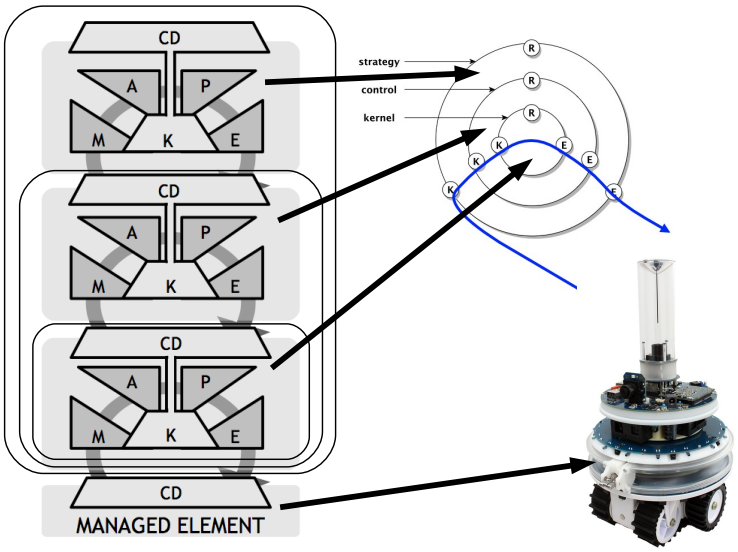

Fig. 6. Architecture as an instance of the framework.

5.1 Architecture of the Case Study

The three layers of the concrete architecture of the case study (cf. Fig. 6, top-right) essentially capture the informal description of [16], in the following sense.

Layer 0 (kernel). This layer models the core functionalities of a robot (see [16, §3]). The rules implement basic movements and actioning of the gripper. Layer 0 corresponds to what some authors call *hardware abstraction layer* (see e.g. [22]).

Layer 1 (basic control). This layer represents the basic controller managing the core functionalities of the robot according to the context. The controller may allow to move only in some directions (e.g. towards a light source) or to search for a robot to grab. This layer corresponds to the individual states of state machines modeling the self-assembly strategies, like the one of Fig. 1 (see [16, §5 and §7]).

Layer 2 (adaptation). This is the layer of the adaptation manager, which reacts to changes in the environment activating the proper basic controller. In our case study, this layer corresponds to the entire state machine modelling the self-assembly strategy of Fig. 1 and, in particular, it takes care of the transitions between its states. This is done by constantly monitoring the environment and the managed component M, and by executing an adaptation phase when needed, which means changing the rules of M. A few other self-assembly strategies are discussed in [16]: they can be implemented by changing the rules of this layer.

The three layers differ in their sets of rules and, of course, in the managed component, but they share part of the signature for knowledge and effects. In particular, knowledge includes predicates about properties of the ground (wall, hole, free), the presence of robots in the surrounding (their LED emissions), and

the direction of the light source (the goal). Effects include moving or emitting a color towards a direction, and trying to grab a robot located in an adjacent cell.

Knowledge and effects are currently implemented as plain sets of predicates. More sophisticated forms of knowledge representation based on some inference mechanism (like PROLOG specifications, epistemic logics, ontologies or constraints) may be possible but are not necessary in the presented case study.

Simulator. The execution environment of the robots is realized by a simulator which consists of three parts: the *orchestrator*, the *scheduler* and the *arena*.

The orchestrator takes care of the actual execution of the actions required to manage the effects generated by (the outermost layer of) a robot. For instance, it decides if a robot can actually move towards the direction it is willing to move.

The scheduler, implemented as an ordinary discrete-event scheduler, activates the scheduled events, allowing a robot or the orchestrator to perform its next action. Intuitively, the emission of an effect e by the outermost layer of a component c causes the scheduling of the event "execute effect e on c" for the orchestrator. Symmetrically, the handling by the orchestrator of an effect previously generated by a component c induces the scheduling of an event "generate next effect" for c.

Finally, the arena defines the scenario where robots run. We abstracted arenas in discrete grids, very much like a chessboard. Each grid's cell has different attributes regarding for example the presence of holes or light sources. A cell may also contain in its attributes (at most) one robot, meaning that the robot is in that position of the arena. Each robot can move or perform an action in eight possible directions (up, down, left, right and the four diagonals).

5.2 Implementation Details

On the structure of adaptive components. Our implementation, similarly to the systems described in [14], relies on Maude's object-like signature (see [8, Chapter 8]). Such signature allows to model concurrent systems as *configurations* (collections) of *objects*, where each object has an identifier, a class and a set of attributes. Intuitively, `< oid : cid | attr1, attr2 >` is an object with identifier `oid`, class `cid` and two attributes `attr1`, `attr2`.

Each layer is implemented as an object with attributes for knowledge (`K`), effects (`E`), rules (`R`) and managed component (`M`): the first two are plain sets of predicates, the third one is a meta-representation of a Maude module, and the fourth one is an object. Three classes are introduced for the different layers, namely `AC0`, `AC1` and `AC2`. For design choice, the objects implementing the layers of a robot have the same identifier: in terms of [14] we use *homunculus objects*.

Therefore a sample robot can have the following overall structure:

```
< c(0) : AC2 | K: gripper(open) on(right,none) towards(right,light) ...,
               E: emitt(up,Green) go(right) ...,
               R: mod_is_sorts_._____endm,
               M: < c(0) : AC1 | K: ..., E: ..., R: ...,
                                 M: < c(0):AC0 | K:..., E:..., R:...> > >
```

On the structure of the simulator. The arena is implemented as a multiset of objects of class `Cell`. A cell may contain in the attributes an object of class `AC2` representing a robot, and the orchestrator implements the move of a robot by changing the cell in which it is stored. This way the robots have no information about the global environment or their current position, but only about the contiguous cells and the direction to take to reach the goal.

The cell encapsulating a robot actually acts as a fourth layer over the object of class `AC2`. In fact, it is responsible of updating its knowledge, of taking care of its effects (e.g. the cell must expose the status of robot's LEDs), and of handling the interactions between the robot and the scheduler.

Rules of adaptive components. The behaviour of each layer is specified by the rules contained in its attribute `R`, which is a term of sort `Module` consisting of a meta-representation of a Maude module. This solution facilitates the implementation of the behaviour of components as ordinary Maude specifications and their treatment for execution (by resorting to meta-level's rewriting features), monitoring and adaptation (by examining and modifying the meta-representation of modules). In fact, on the one hand a generic meta-rule can be used to *self-execute* an object: the object with rules R proceeds by executing R in its meta-representation. On the other hand, rules are exposed to the outer component, which can execute or manipulate the inner one, and analyse the obtained outcome.

In order to give an idea on how the flows of execution and information of Fig. 5 are actually implemented, we present one sample rule for each of the three layers. For the sake of presentation we abstract from irrelevant details.

Layer 0. This layer implements the core functionalities of robots. For example, the following rule computes the set of directions towards which a robot can move:

```
rl [admissibleMovements] :
   < oid : AC0 | K: oneStep k0, E: e0                          , A0 >
=> < oid : AC0 | K:         k0, E: e0 canMoveTo(freeDirs(k0)), A0 > .
```

A rule, like `admissibleMovements`, can be applied to a Maude term t if its left-hand side (LHS) (here the object `< oid : AC0 | ... >` preceding `=>`) matches a subterm of t with some matching substitution σ, and in this case the application consists of replacing the matched sub-term with the term obtained by applying σ to the right-hand side, i.e. the object following `=>`. We shall also use Maude equations: they have higher priority than rules, meaning that rules are applied only to terms in normal form with respect to the equations.

Rule `admissibleMovements` rewrites an `AC0` object to itself, enriching its effects with the term obtained by simplifying equationally `canMoveTo(freeDirs(k0))`. Notice that the constant `oneStep` is consumed by the application of the rule: intuitively, it is a token used to inhibit further applications of the rule, obtaining a one step rewriting. The equations will reduce `freeDirs(k0)` to the set of directions containing each `dir` appearing in a fact `on(dir,content)` of `k0` such that `content`

does not contain obstacles. Operator `canMoveTo` instead is a constructor, hence it cannot be further reduced.

Layer 1. Objects of class `AC1` correspond to components of layer 1, implementing individual states of the state machine of Fig. 1. Rules of this layer can execute the component of the lower level providing additional knowledge, and can elaborate the resulting effects. The following rule implements (part of) the logic of state `Independent_Phototaxis`, computing the desired direction towards which to move.

```
crl [IP-main]: < oid:AC1 | K: oneStep k1, E: e1        ,
                           M: < oid:AC0 | K: k0 , E: e0, R: m0, A0  >, A1 >
 =>            < oid:AC1 | K:         k1, E: e1 go(dir),
                           M: < oid:AC0 | K: k0b, E: e0, R: m0, A0b >, A1 >
 if < oid : AC0 | K: k0b, E: e0 canMoveTo(freeDirs), A0b > :=
    execute(< oid : AC0 | K: oneStep update1To0(k1,k0), E: e0, A0 >, m0)
 /\ preferredDirs := intersection(freeDirs, dirsToLight(k1))
 /\ dir := uniformlyChooseDir(preferredDirs, freeDirs) .
```

This is a conditional rule, as evident from the keyword `crl` and the `if` clause following the RHS. Thus, it can be applied to a matched sub-term only if its (firing) condition is satisfied under the matching. In this case the condition is the conjunction (`/\`) of three sub-conditions, each consisting of a sort of assignment. The sub-conditions are evaluated sequentially, and the LHS of symbol `:=` will be bound in the rest of the rule to the term obtained by reducing its RHS.

`execute(obj,m)` makes use of Maude's meta-level functionalities to execute object `obj` via the rules meta-represented in `m`. More precisely, in rule `IP-main`, the operator `execute` will apply a single rule of module `m0` to the managed component `< oid : AC0 ... >`, after having updated its knowledge. In fact the operation `update1To0(k1,k0)` implements a (controlled) propagation of the knowledge from layer 1 to layer 0, filtering `k1` before updating `k0` (e.g. information about the surrounding cells is propagated, but information about the goal is discarded).

The assignment of the first sub-condition also binds `freeDirs` to the directions towards which the managed component can move. This is used in the second sub-condition to compute the intersection between the directions in `freeDirs` and those towards the light, evaluated reducing `dirsToLight(k1)`. The resulting set of directions is bound to `preferredDirs`. Finally, in the third sub-condition `dir` is bound to a direction randomly chosen from `preferredDirs`, or from `freeDirs` if the first set is empty. Comparing the LHS and the RHS, one sees that the overall effect of rule `IP-main` is the production of a new effect at layer 1, `go(dir)`, and the update of the knowledge of the managed component of layer 0.

Notice that the rules of layer 0 (`m0`) are not affected by the rule: in fact in our implementation rules of layer 1 never trigger an adaptation phase on layer 0. This is just a design choice, as clearly our architecture does not forbid it.

Layer 2. A component of this layer corresponds to the entire state machine of Fig. 1. It monitors the environment, and at each step it triggers a reduction of the managed component of layer 1; if necessary, it also enforces a transition from

the current state to a new one of the state machine by performing an adaptation phase, i.e. by changing the rules of the managed component.

The following is the main rule governing this layer:

```
crl [adaptAndExecute]:
    < oid : AC2 | K: nextEffect k2 , E: e2                              ,
                  M: < oid : AC1 | K: k1 , E: e1 , R: m1 , A1  >, A2  >
 => < oid : AC2 | K:            k2A, E: e2A schedule(event(oid,effect)),
                  M: < oid : AC1 | K: k1b, E: e1A, R: m1A, A1b >, A2A >
 if < oid : AC2 | K: k2A, E: e2A,
                  M: < oid : AC1 | K: k1A, E: e1A, R: m1A, A1A >, A2A > :=
    computeAdaptationPhase( < oid : AC2 | K: k2 , E: e2 ,
                  M: < oid : AC1 | K: k1 , E: e1 , R: m1 , A1  >, A2  > )
/\  < oid : AC1 | K: k1b, E: e1A effect, A1b > := execute(
    < oid : AC1 | K: oneStep update2To1(k2A,k1A), E: e1A, A1A >, m1A ) .
```

The rule is triggered by the token `nextEffect`, generated by the orchestrator, and propagated by the cell containing the robot. The execution of the rule consists of an adaptation phase followed by an execution phase, both on the managed component. The two phases are triggered by the two sub-conditions of the rule.

The adaptation phase is computed by the operation `computeAdaptationPhase`, using the knowledge of layer 2 (`k2`) to enact a state transition, if necessary. Among those defining the operation, the equation below encodes the transition of Fig. 1 from state `Aggregate` to state `Self_Assemble`, labeled with `Close to red`:

```
ceq [AggToSA]: computeAdaptationPhase(
     < oid2 : AC2 | K: state(Aggregate)    k2, E: e2          ,
                    M: < oid1 : AC1 | R: m1 , E: e1  , A1 > , A2 >)
  =  < oid2 : AC2 | K: state(SelfAssemble) k2, E: emitt(green),
                    M: < oid1 : AC1 | R: m1b, E: none, A1 > , A2 >
 if seeEffect(led(red),k2)
 /\ m1b := upModule('AC1-SELF_ASSEMBLE,false) .
```

The conditional equation states that if a robot in state `Aggregate` sees in its neighborhood a robot with red LEDs on, then it must pass to state `Self_Assemble` and turn on its green LEDs. Also the rules of the managed component are changed: the new module `m1b` is obtained with the operation `upModule`, producing the meta-representation of the Maude module passed as first parameter.

We specified one equation for each transition of Fig. 1, plus the following one where `owise` is a special attribute that tells the interpreter to apply the equation only if none of the others is applicable:

```
eq [idle]: computeAdaptationPhase(obj) = obj [ owise ] .
```

Once the adaptation phase is concluded, the second sub-condition of rule `adaptAndExecute` takes care of the one step execution of the (possibly adapted) managed component, using operation `execute`. Finally, the effects generated by layer 1 are wrapped in the constructors `event` and `schedule`, and are added to the effects of layer 2, so that the cell containing it will propagate it to the scheduler.

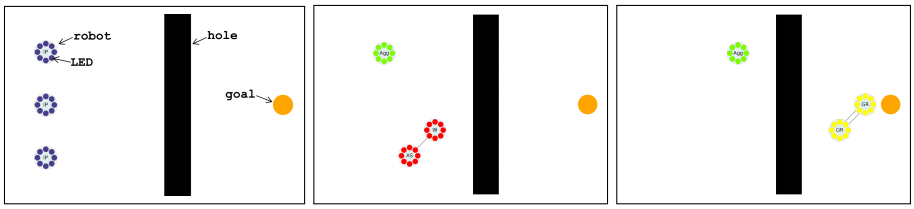

Fig. 7. Three states of a simulation: initial (left), assembly (middle), final (right).

6 Analysis of Adaptation Strategies

This section describes some of the analysis activities carried out with our implementation, available at `http://sysma.lab.imtlucca.it/tools/ensembles` together with some additional material such as animated simulations.

The analysis has been carried out in two phases: (§6.1) discrete event simulation; and (§6.2) statistical model checking. The rationale is the following.

In the early development phases we have mainly concentrated on performing single simulations that have been informally analyzed by observing the behavior of the assemblies in the automatically generated animations. A couple of trial-and-error iterations (where the model was fixed whenever some anomalous behavior was spotted) were enough for the model to acquire sufficient maturity to undergo a more rigorous analysis in terms of model checking.

Ordinary model checking is possible in the Maude framework (via Maude's reachability analysis capabilities, or LTL model checker) but suffers from the state explosion problem and is limited to small scenarios and to *qualitative* properties. To tackle larger scenarios, and to gain more insight into the probabilistic model by reasoning about *probabilities* and *quantities* rather than *possibilities*, we resorted to statistical model checking techniques.

We now provide the details of these analysis phases, centered around one crucial question: *How many robots reach the goal by crossing the hole?*

6.1 Simulations

Simulations are performed thanks to the discrete-event simulator mentioned in §5.2 along the lines of the ones reported in [1, 2, 20]. Valuable help has been obtained implementing an exporter from Maude `Configuration` terms to DOT graphs[3], offering the automatic generation of images from states: they have greatly facilitated the debugging of our code.

For example, Fig. 7 illustrates three states of one interesting simulation in which robots execute the *basic self-assembly response strategy*. The initial state (left) consists of three robots (grey circles with small dots on their perimeter) in their initial state (emitting blue light), a wide hole (the black rectangle) and the goal of the robots, i.e. a source of light (the orange circle on the right). After some steps, where the robots execute the self-assembly strategy, two robots finally get

[3] DOT is a well-established graph description language (`http://www.graphviz.org/`).

assembled (middle of Fig. 7). The assembled robots can then safely cross the hole and reach the goal (right of Fig. 7), while the not assembled one remains abandoned in the left part of the arena.

While performing such simulations with different scenarios, varying the location of the goal and number and distribution of the robots, and with different parameters for duration of timeouts and actions, we observed several *bizarre* behaviors. For instance, in various simulations we observed some not-assembled robots erroneously believing to be part of an assembly, moving into the hole and disappearing. In other simulations we instead noticed pairs of robots grabbing each other. These observations triggered the following questions: *Is there an error in our implementation? Is there an error in the strategies defined in* [16]?

Examining carefully the description of the strategy, we discovered that the two behaviors are indeed not explicitly disallowed in [16] and originated by the two transitions (see Fig. 1 in §2) outgoing from the state `Assembly_Seed` (willing to be grabbed). The first transition leads to state `Wait`, triggered by the expiration of a timeout, while the second one leads to state `Aggregate` (willing to grab), triggered by the event `See red` (i.e. another robot willing to be grabbed). Considering the first behavior, a robot can change from state `Assembly_Seed` to state `Wait` even if no other robot is attached to it. The robot then evolves to state `Connected_phototaxis` believing to be assembled with other robots. Considering instead the second behaviour, once a robot i grabs a robot j, i becomes itself "willing to be grabbed" (turning on its red LEDs) to allow other robots to connect to the assembly. Now, it is clear that if j is grabbed while being in state `Assembly_Seed`, then its transition towards state `Aggregate` is allowed, leading to the second bizarre behaviour. Interestingly enough, we hence notice that the two bizarre behaviors strongly depend on the duration of the timeout: a short one favors the first behaviour, while a long one favors the second one.

Are these behaviors actually possible in real robots or are they forbidden by real life constraints (e.g. due to the physical structure of the robots or to some real-time aspects)? The answer to this question is being investigated within the ASCENS project [4]. However, our experience makes it evident that the self-assembly strategies described in [16] might be adequate for s-bots but not in general for self-assembly settings where other constraints might apply. Fortunately, both bizarre behaviors can be fixed easily by adding further conditions to the two mentioned transitions of the adaptation strategy. In particular, the transition from `Assembly_Seed` to `Aggregate` requires a further condition to ensure that the robot has been gripped. Conversely, the transition from `Assembly_Seed` to state `Aggregate` requires exactly the contrary, i.e. the robot must not be gripped.

6.2 Statistical Model Checking

A qualitative analysis can prove that an assembly strategy can result in different degrees of success, from full success (all robots reach the goal) to full failure (no robot reaches the goal). However, in the kind of scenario under study different levels of success are typically of interest. The really interesting question is how

likely are they? Moreover, another interesting measure could be the *expected number* of robots reaching the goal.

An analysis based on statistical model checking (see e.g. [19, 20, 2]) is more appropriate in these cases. Such techniques do not yield the absolute confidence of qualitative model checking but allow to analyze (up to some statistical errors and at different levels of confidence) larger scenarios and to deal with the stochastic nature of probabilistic systems.

We consider the following properties: (P_0) *What is the probability that no robot reaches the goal?*; (P_1) *What is the probability that at least one robot reaches it?*; and (P_2) *What is the expected number of robots reaching the goal?*.

We have used PVesta [2], a parallel statistical model checker, to perform some comparative analysis. The tool performs a statistical evaluation (Monte Carlo based) of properties expressed in the transient fragments of PCTL and CSL, and of quantitative temporal expressions (QuaTEx)[1], allowing to query about expected values of real-typed expressions of a probabilistic model.

We have performed a comparative analysis (with respect to the above properties) between two different strategies: namely the original *basic self-assembly response* and the variant that fixes the bizarre behaviors discussed above. For each experiment, where we fixed 120 as maximum number of sytem steps, all robots execute the same strategy. The aim was to gain some intuition about the success and performance impact of the absence of the bizarre behaviors rather than to derive exact statistical measures. The arena was configured as follows (cf. Fig. 8): an 11×7 grid containing 3 robots, the goal (a source of light) and a hole dividing the robots from the goal. We remind that a robot alone is not able to cross the hole, and hence needs to cooperate (assemble) with other robots to cross it.

Roughly, our variant of the strategy exhibits a better success rate. More precisely, the analysis of P_0 on the original strategy provides 0.48 (i.e. about half of the cases ends up without any robot reaching the goal), while for our variant we obtain 0.36. Regarding P_1, our variant exhibits again a better rate (0.64) than the original one (0.52). Finally, the expected number of successful robots (P_2) is 1.07 in the original case, while in our variant case it is 1.38.

These preliminary data of the statistical analysis seem to confirm our intuition. Forthcoming experiments will consider other robot features and strategies, and will validate our results against the ones reported in [16].

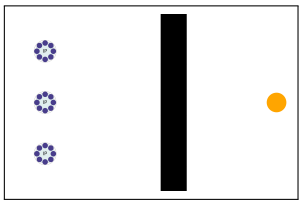

Fig. 8. An initial state.

A Sample QuaTEx Expression. We conclude this section discussing the quantitative temporal expression we defined to estimate the expected number of robots reaching the goal.

QuaTEx [1] is a language to query quantitative aspects of probabilistic systems. Exactly as temporal logics allows to express temporal formulae, QuaTEx allows to write quantitative temporal expressions.

PVesta statistically evaluates quantitative temporal expressions with respect to two parameters: α and δ. Specifically, expected values are computed from n independent simulations, with n large enough to grant that if a QuaTEx expression is estimated as $\overline{x}$, then, with probability $(1-\alpha)$, its actual value belongs to the interval $[(1-\delta)\overline{x}, (1+\delta)\overline{x}]$. For the experiments in §6.2 we fixed $\alpha = \delta = 0.05$.

In the rest of this section we see how the mentioned QuaTEx expression has been defined, and how its value is actually computed for single simulations. We do not detail how PVesta performs one-step executions, since this is out of the scope of this paper. Details can be found in [1].

Before defining our expression it is necessary to define real-typed Maude operations representing the states predicates we are interested in. We defined the state predicate `completed : Configuration -> Float`, reducing to 1.0 for terminal states, and to 0.0 otherwise. A terminal state is a state with no more robots, a state with all the robots in goal, or the state obtained after a given maximum number of steps. We also defined the state predicate `countRobotInGoal : Configuration -> Float`, counting the number of succesful robots.

Then we defined the equations necessary to PVesta to access such predicates (where `C` is a variable with sort `Configuration`): `eq val(0,C) = completed(C)`, and `eq val(1,C) = countRobotInGoal(C)`. Actually QuaTEx's syntax requires to indicate the term "`val(n,s)`" with "`s.rval(n)`", where `n` and `s` are respectively terms with sort `Natural` and `Configuration`.

Finally, the QuaTEx expression to estimate the expected number of robots reaching the goal is easily expressed as

```
count_s-bots_in_goal() = if { s.rval(0) == 1.0 } then s.rval(1)
                         else #count_s-bots_in_goal() fi;
eval E[ count_s-bots_in_goal() ] ;
```

Informally, a QuaTEx expression consists in a list of definitions of recursive temporal operators, followed by a query of the expected value of a path expression obtained (arithmetically) combining the temporal operators. Our formula defines the temporal operator `count_s-bots_in_goal()`, which also corresponds to the estimated path expression `eval E[ count_s-bots_in_goal() ]`.

The path expression is evaluated by PVesta in the initial state (`s`) of the system (e.g. the state depicted in Fig. 8). The tool first evaluates the guard of the `if_then_else` statement, i.e. `s.rval(0) == 1.0`. The condition reads as "is the state predicate `rval(0)` equal to 1.0 if evaluated in the state `s`?", and corresponds to "is the current state a final state?". If the guard is evaluated to true, then the path expression is evaluated as `s.rval(1)`, that is in the number of robots that reached the goal in the state `s`. If the guard is evaluated to false, then the path expression is evaluated as `#count_s-bots_in_goal()`, read "evaluate `count_s-bots_in_goal()` in the state obtained after one step of execution". The symbol `#`, named "next", is in fact a primitive temporal operator.

To conclude, the evaluation of the QuaTEx expression consists in performing step-wise system simulations, and is evaluated as the (mean of the) number of robots that reached the goal in the terminal states of each simulation.

7 Conclusion

The contributions of our paper are: (i) a description (§4, §5) of how to realize in Maude our recently proposed approach to adaptive systems [6] in a simple and natural way; and (ii) a description (§6) of how to exploit the Maude toolset for the analysis of our models, and PVesta [2] in particular.

Our work is inspired by early approaches to coordination and adaptation based on distributed object reflection [14, 21] and research efforts to apply formal analysis onto such kind of systems (e.g. [13]), with a particular focus on adaptive systems (e.g. [22, 4]). Among those, the PAGODA project [22] is the closest in spirit and shape. Our work is original in its clear and neat representation and role of *control data* in the architecture, and in the fact that this is, as far as we know, the first analysis of self-assembly strategies based on statistical model checking.

The case study of self-assembly strategies for robot swarms [16] has contributed to assess our approach. Overall, the conducted experimentation demonstrates that Maude is well-suited for prototyping self-assembly systems in early development phases, and that the associated simulation can be useful to discover and resolve small ambiguities and bugs in self-assembly strategies. Furthermore, statistical model checking can provide preliminary estimations of success rate, that can be used to compare different strategies and also to validate/confute/refine analogous measures provided by other tools or in real world experiments.

We plan to further develop our work by considering other case studies, more realistic abstractions and more modular implementations. However, the key challenging question we want to tackle is: *can we exploit the proposed architecture to design smarter adaptation strategies or to facilitate their analysis?* We envision several interesting paths in this regard. First, we are investigating how logical reflection can be exploited at each layer of the architecture, for instance to equip components with dynamic planning capabilities based on symbolic reachability techniques. Second, we are developing a compositional reasoning technique that exploits the hierarchical structure of the layered architecture.

All in all, we believe that our work is a promising step towards the non-trivial challenges of building predictive adaptive systems, and to analyze them.

Acknowledgements. We are grateful to the anonymous reviewers for their fruitful criticism and to the organizers of the *AWASS 2012 summer school* for the opportunity to mentor a case study based on the experience of this paper.

References

1. Agha, G.A., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. In: Cerone, A., Wiklicky, H. (eds.) QAPL 2005. ENTCS, vol. 153(2), pp. 213–239. Elsevier (2006)
2. AlTurki, M., Meseguer, J.: PVeStA: A parallel statistical model checking and quantitative analysis tool. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 386–392. Springer (2011)
3. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Reflecting on self-adaptive software systems. In: SEAMS 2009. pp. 38–47. IEEE (2009)

4. Autonomic Service Component Ensembles (ASCENS), http://www.ascens-ist.eu
5. Broy, M., Leuxner, C., Sitou, W., Spanfelner, B., Winter, S.: Formalizing the notion of adaptive system behavior. In: Shin, S.Y., Ossowski, S. (eds.) SAC 2009. pp. 1029–1033. ACM (2009)
6. Bruni, R., Corradini, A., Lluch Lafuente, A., Gadducci, F., Vandin, A.: A conceptual framework for adaptation. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 240–254. Springer (2012)
7. Cabri, G., Puviani, M., Zambonelli, F.: Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles. In: Smari, W.W., Fox, G.C. (eds.) CTS 2011. pp. 508–515. IEEE Computer Society (2011)
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)
9. Horn, P.: Autonomic Computing: IBM's perspective on the State of Information Technology (2001)
10. IBM Corporation: An Architectural Blueprint for Autonomic Computing (2006)
11. Karsai, G., Sztipanovits, J.: A model-based approach to self-adaptive software. Intelligent Systems and their Applications 14(3), 46–53 (1999)
12. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
13. Meseguer, J., Sharykin, R.: Specification and analysis of distributed object-based stochastic hybrid systems. In: Hespanha, J., Tiwari, A. (eds.) HSCC 2006, LNCS, vol. 3927, pp. 460–475. Springer (2006)
14. Meseguer, J., Talcott, C.: Semantic models for distributed object reflection. In: Magnusson, B. (ed.) ECOOP 2002, LNCS, vol. 2374, pp. 1–36. Springer (2002)
15. Mondada, F., Pettinaro, G.C., Guignard, A., Kwee, I.W., Floreano, D., Deneubourg, J.L., Nolfi, S., Gambardella, L.M., Dorigo, M.: Swarm-bot: A new distributed robotic concept. Autonomous Robots 17(2-3), 193–221 (2004)
16. O'Grady, R., Groß, R., Christensen, A.L., Dorigo, M.: Self-assembly strategies in a group of autonomous mobile robots. Autonomous Robots 28(4), 439–455 (2010)
17. Pavlovic, D.: Towards semantics of self-adaptive software. In: Robertson, P., Shrobe, H.E., Laddaga, R. (eds.) IWSAS 2000. LNCS, vol. 1936, pp. 65–74. Springer (2000)
18. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems 4(2), 1–42 (2009)
19. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 266–280. Springer (2005)
20. Sen, K., Viswanathan, M., Agha, G.A.: Vesta: A statistical model-checker and analyzer for probabilistic systems. In: Baier, C., Chiola, G., Smirni, E. (eds.) QEST 2005. pp. 251–252. IEEE Computer Society (2005)
21. Talcott, C.L.: Coordination models based on a formal model of distributed object reflection. In: Brim, L., Linden, I. (eds.) MTCoord 2005. ENTCS, vol. 150(1), pp. 143–157. Elsevier (2006)
22. Talcott, C.L.: Policy-based coordination in PAGODA: A case study. In: Boella, G., Dastani, M., Omicini, A., van der Torre, L.W., Cerna, I., Linden, I. (eds.) CoOrg 2006 & MTCoord 2006. ENTCS, vol. 181, pp. 97–112. Elsevier (2007)
23. Weyns, D., Malek, S., Andersson, J.: FORMS: a formal reference model for self-adaptation. In: Figueiredo, R., Kiciman, E. (eds.) ICAC 2010. pp. 205–214. ACM (2010)
24. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) ICSE 2006. pp. 371–380. ACM (2006)