Modelling and analyzing adaptive self-assembling 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 is becoming a critical need. The research community has accepted the challenge by proposing approaches of various nature: from software architectures, to programming paradigms, to analysis techniques. Our own contribution in this regard is a conceptual framework for adaptation centered around the stressed role of *control data*. The framework is naturally realized in a reflective logical language like Maude by using the Reflective Russian Dolls model, as we show in this paper. We exploit the recently released statistical model checker PVesta to analyze a prominent example of adaptive system: robot swarms equipped with obstacle-avoidance self-assembly strategies.

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 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 analysis of handcrafted adaptive components are unlikely to pay back, because the results are often scarcely reusable when the components' software is updated or extended with new features.

* Research supported by the European Integrated Project 257414 ASCENS

Given the above premises and our background, we address here some of the above concerns, presenting the methodology that we have devised and applied for prototyping well-engineered self-adaptive components. Our case study consists of modeling and analyzing self-assembly strategies of robots having the goal of crossing a hole while navigating towards a light source, as detailed later. 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 [5], 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 some relevant aspects of our experience when using this tool.

When is a software system adaptive? Self-adaptation is considered 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 for example 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]. Along the years 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 [6], 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 [4], among others.

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 [5], 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 §2.

Is Maude a convenient setting to study self-adaptation? A "convenient" framework must provide a reusable methodology for modelling self-adaptive systems independently from 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 [7] is a good candidate. First, the versatility of rewrite theories provides 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 naturally expressed 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 above sketched architecture, to be further elaborated in §3, facilitates the rapid prototyping of self-adaptive systems, to be simulated, analyzed and vali-

dated exploiting the formal analysis toolset of Maude, including statistical model checking via the PVesta tool [2]. Pragmatically, the possibility to rapidly develop and simulate self-adaptive systems and to compare the behaviour emerging from different adaptation strategies is very important for case studies like the robotic scenario described in the next paragraphs. In fact, such physical devices are very sophisticated to program and their experimentation in real world testing environments involve long time consumption (6 hours or more for a single run) and a limited number of pieces is available (around 25 units) because they are expensive to maintain. In fact, their hardware (both mechanics and electronic) and software are updated very frequently, which makes it harder to build and rely on sophisticated simulators that 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 differ substantially from the simulated runs.

Case study: Self-assembling robot swarms. Self-assembling robotic systems are formed by independent robots that are capable to connect physically when the environment prevents them from reaching their goals individually. 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, because 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. Such features make the self-assembling 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".

The experiments in [16] were conducted on the SWARM-BOT robotic platform [15], whose constituents are called s-bots (see Fig. 5, 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 regularly 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 (e.g. the color green can be used to mark the will to connect to an existing ensemble, and the color red can be used for the will to create a new assembly). The ring can also be grasped by other s-bots thanks to a gripper-based mechanism.

Roughly, the experimented strategies 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 Appendix A), where each s-bot move independently (light blue) until an obstacle is found, in which case tries to

aggregate (light green) to some nearby assembly, if some available, or it becomes the *seed* of a new assembly (light red); (iii) the *preemptive self-assembly* strategy, where the s-bots self-assemble irrespective 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 the assembled robots is coordinated according to a leader-follower architecture from the seed of the assembly.

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`.

Synopsis. In §2 we summarize the conceptual framework for adaptation that we have proposed in [5] and along which we will design adaptive systems in Maude. The general guidelines and principles to be exploited in Maude for modelling self-adaptive systems are described in §3, together with the concrete software architecture used to realize our conceptual framework. In §4 we illustrate the concrete modelling of hole-crossing swarm-bots. The conducted experimentations are described in §5, focusing, for the sake of presentation, on the basic self-assembly response strategy. Some concluding remarks and ongoing research avenues are discussed in §6.

We assume the reader to have some familiarity the Maude framework. For reviewers convenience, we reported in the appendices a detailed description of one self-assembly strategy from [16], and some significant fragments of the source code of our case study.

2 A framework for adaptation

Before describing the way we have modeled and analysed the scenarios presented in the previous section, let us explain some guidelines that we followed when designing the system. The fundamental 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 characterizing features that distinguish such systems from ordinary ("non-adaptive") ones.

We addressed this problem in [5], proposing a simple structural criterion to characterize adaptivity. Oversimplifying a bit, according to a common *black-box* perspective, a software system is often called "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, because according to it almost any software system can be considered self-adaptive. Indeed, any system can likely *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 the input of a data from the user).

Therefore 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 several others. A component is called *adaptable* if it has a clearly identified collection of control data that can be modified at run-time. Thus if the control data are not identified or cannot be modified, the system is not adaptable. 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 a component is *self-adaptive* if it is able to modify its own control data at run-time.

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 that governs the behavior. Consequently, the nature of control data can vary considerably depending both on the degree of adaptivity of the system and on the nature of 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 [5] we discussed how our simple criterion for adaptivity can be applied to several of the reference models cited in the introduction, identifying for each of them what would be a reasonable choice of control data. 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, let us 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 reference model suggests an obvious choice for the control data: these are the data of the managed component which are either sensed by the monitor or modified by the execute phase of the control loop. Thus the control data represent the interface exposed by the managed components through which the control loop can operate, as shown in Fig. 1. Clearly, by our

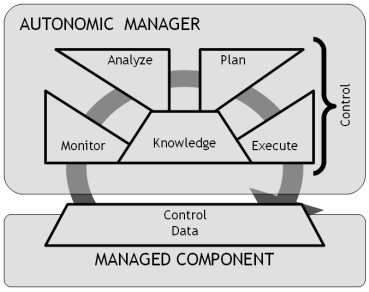

Fig. 1. Control data in MAPE-K.

definitions the managed component is adaptive, and the system made of both the component and the control loop is self-adaptive.

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 also the manager (the control loop) should expose in an interface its control data, which are conceptually part of its knowledge repository. In this way, the approach becomes compositional in a hierarchical way, which allows one to build towers of adaptive components (Fig. 2).

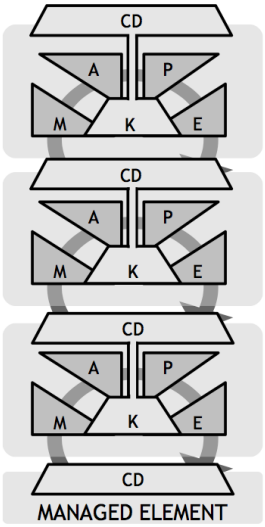

Fig. 2. Tower of adaptation.

3 Adaptivity in Maude

We argue here the suitability of Maude and rewriting logic as language and model for adaptivity (§3.1), describe a generic architecture for developing adaptive components in Maude (§3.2) and show that it conforms to our framework (§3.3).

3.1 Maude, logical reflection and adaptivity

Maude [7] is particularly suitable for the specification of adaptive systems, thanks to the reflective capabilities of rewriting logic. The reflection mechanism of the logic yields what is called the *tower of reflection*. At the ground level, a rewrite theory $\mathcal{R}$ (e.g. software module) allows us 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 us infer the computation at the "meta-level", where theories and terms are meta-represented as terms: $\mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{t}) \to (\overline{\mathcal{R}}, \overline{t'})$. The process can be repeated again and again as $\mathcal{U}$ itself is a rewrite theory. This mechanism is efficiently supported by Maude and has given rise to many interesting meta-programming applications like analysis and transformation tools.

In particular, the reflection mechanism of rewriting logic has been exploited in [14] to formalize a model for distributed object reflection, suitable for the specification of adaptive systems. Such model, suggestively 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. Even at this informal level, it is pretty clear that the RRD model falls within our conceptual framework by identifying as "control data" for each layer the rules of its theory that are possibly modified by the upper layer. Note that, while the tower of reflection relies on a white-box adaptation, the russian dolls approach can deal equally well with black-box components, because wrapped configurations can be managed by message passing. The RRD model has been further exploited for modeling policy-based coordination [21] and for the design of PAGODA, a modular architecture for specifying autonomous systems [22]. The case study presented in the next sections will conform to this model.

3.2 Generic architecture

Intra-layer architecture The structure of each layer is illustrated in Fig. 3. The main constituents are: knowledge (K), effects (E), rules (R) and managed components (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 (i.e. they are the control data).

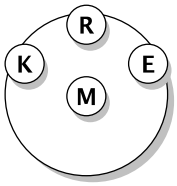

Fig. 3. Intra-layer.

The knowledge represents the information in possess of the component. It can contain data that represent the internal state or assumptions about the component's surrounding environment. For example, the knowledge in our case study includes the status of the gripper, but also the presence of a nearby hole in the ground and the existence of light-emitting artefacts around each robot.

The effects are the actions that the component is willing to perform on its enclosing context. The effects in our example are the actions to be performed on the robot being controlled: e.g. the instruction to move towards some direction, to attach to another component, or to turn on/off some colored led.

The rules determine how the effects are generated in reaction to the knowledge. They constitute the control data of the component. Typically, some rules take care of updating the knowledge of managed components, executing them and collecting (and possibly propagating) their effects. This is the case in which a component acts as a sort of interpreter. In other cases rules can act upon the rules of managed components. These are adaptation rules. In our case study, the rules define, for instance, the assembly strategies.

The managed components are those components in the underlying layer. This part is absent in the innermost layer.

Inter-layer architecture Layers are organized hierarchically. The left part of Fig. 4 shows a three-layered architecture. Each layer contains its own knowledge, effects and rules and, in addition, has the underlying layer as managed component.

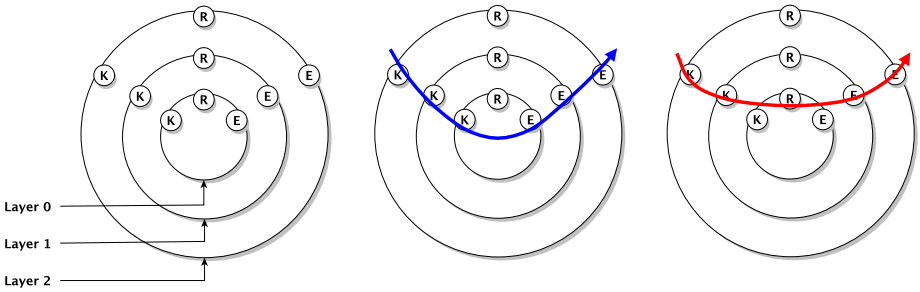

Fig. 4. Inter-layer architecture (left), ordinary flow (center), adaptation flow (right).

The diagram in the middle highlights the control and data flow of ordinary behavior. 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 to modifications.

Instead the diagram depicted on the right of Fig. 4 corresponds to adaptation. The outermost layer acts ordinarily, but in layer 1 an adaptation is triggered. This can be due to some condition on the present knowledge or the status of the managed components. The result is that the rules of layer 1 act (among other things) upon the rules of layer 0 (denoted by the arrow passing through them).

3.3 Architecture as a framework instance

In which sense is the architecture in §3.2 an instance of the framework in §2? Fig. 5 illustrates the answer. The main idea is that our architecture imposes the encapsulation of all components of the tower, apart from the managed element (i.e. the robot itself). This way we obtain several advantages: (i) management becomes hierarchical (e.g. self-management is forbidden); and (ii) the managed element is controlled by the topmost manager component. This does not necessarily mean that the outer layer implements the basic operation logic but, as we see in § 4.1, that this can be demanded to an inner layer, of which the outer can be a monitor, a filter or an interface adaptor (i.e. as a wrapper).

How are the elementary constituents of the framework correlated to the ones of the presented architecture? The rules (R) of the architecture implement the MAPE activities of the framework and are exposed as the control data (CD); the knowledge (K) of the architecture corresponds to the knowledge of the framework (minus the rules); the effects (E) of the architecture correspond essentially to the outgoing control flow in the framework (see the curved arrow in Fig. 5).

4 Architecture and implementation of the case study

This section describes the concrete architecture of our case study (§4.1) and some details of the actual implementation (§4.2).

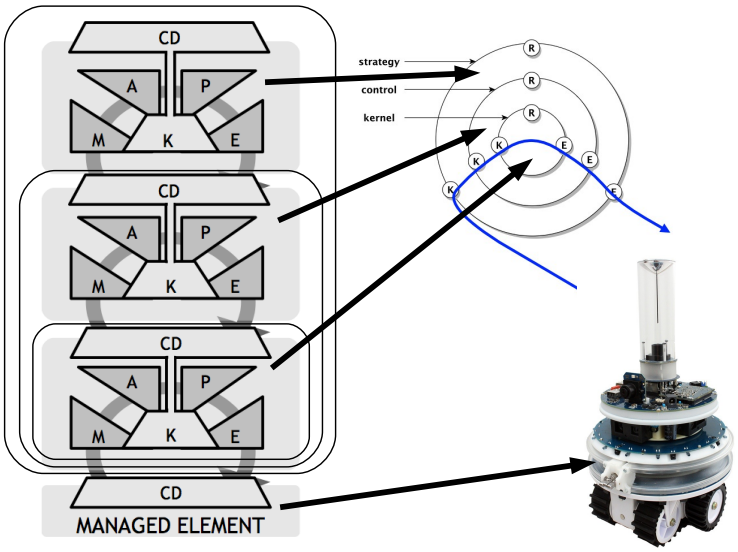

Fig. 5. Architecture as an instance of the framework.

4.1 Architecture of the case study

The layers of the concrete architecture of the case study (cf. Fig. 5, right) essentially capture the informal description of [16]: the high-level state machine of self-assembly strategies, the code executed in each of its states, and the elementary robot functionalities are mapped to separate layers.

Level 0 (managed component). This level[3] models the core functionalities of a robot (see §3 of [16]). Rules are used to model basic movements, color emissions through the LEDs and actioning of the attachment gripper.

Level 1 (basic controller). This level represents the basic controller governing the core functionalities of the robot according to the context. For example, a basic controller may allow to move only along certain directions (e.g. towards particular light sources), or to search for a robot to grab. With respect to the description of §5 and §7 of [16], this layer corresponds to the individual states of the state machines implementing the adaptation strategies.

Level 2 (adaptation). This is the level of the adaptation manager, who decides how to react to changes in the environment by activating the corresponding basic controller. With respect to the description of [16], this layer corresponds to the state machines that model the self-assembly strategies (see Fig. 8 in Appendix A or Fig. 3, 4, 8 and 10 and §5 and §7 of [16]), and in particular takes care of the transitions between its states. In doing so it monitors the environment and its

[3] Which corresponds to PAGODA's hardware abstraction layer [22].

managed component M, and possibly executes an adaptation phase changing the rules of M.

All layers differ in their sets of rules and, of course, in their sets of managed components, but they share the same signature for knowledge and effects. In particular, the knowledge includes predicates about properties of the ground (wall, hole, free) and the presence of robots (their LED emissions) in the surrounding, and the direction of light sources (the goal).

Effects include moving towards one of the four directions, emitting a color towards one of the four directions and trying to grab a robot located at one of the four adjacent cells.

4.2 Implementation details

Components. Our implementation (see §B for an excerpt), similar to the systems described in [14], relies on Maude's object based signature (see chapter 8 of [7]). Without detailing its concrete syntax, we recall that such signature allows us to model concurrent systems as collections (multisets) of *objects* called *configurations*, where objects are defined by an identifier, a class and a set of attributes.

Each layer is implemented as an object having knowledge, effects, rules and managed components as attributes.

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, ontologies or databases) are subject of current investigation but not necessary in the presented case study.

Managed components are just configurations, whose objects implement the underlying layer. In this case study we consider single managed objects with the same identifier of their manager. In terms of [14] we use *homunculus objects*.

Rules are implemented as system modules. That is, every object is equipped with an attribute of type `Module` implementing its behavior. This enormously facilitates their implementation as ordinary Maude specifications and their treatment both for execution (by resorting to rewriting and reachability features of the Maude meta-level) and monitoring and adaptation (by examining and modifying the meta-representation of modules).

A generic rule can be used to *self-execute* an object: an object with rules R proceeds by executing R in its meta-representation. The rules in R can execute the managed components and decide what to do with the outcome, or manipulate the managed components to enact adaptation.

Simulator. Our simulator consists basically of three blocks: the *arena*, the *orchestrator* and the *scheduler*.

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 like the type of the ground or the presence of robots. Only one robot per cell is

allowed. We currently discretized the possible directions towards which a robot can perform an action into up, down, left and right.

The orchestrator synchronizes robots with the arena, determining their knowledge and managing their effects. For instance, it decides if a robot can effectively move towards the direction it is willing to move, or to let a robot sink in a hole.

Finally, the scheduler determines when a robot or the orchestrator can perform an action. It is implemented as an ordinary discrete-event scheduler.

5 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: (§5.1) discrete event simulation; and (§5.2) statistical analysis. 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 rechability analyzer of 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 reasoning about *probabilities* and *quantities* rather than *possibilities*, we have resorted to statistical model checking techniques.

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

5.1 Simulations

Simulations are performed thanks to the discrete-event simulator described in §4.2 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[4], offering the automatic generation of images from states: they have greatly facilitated the debugging of our code.

For example, Fig. 6 illustrates three states of one interesting simulation, in which s-bots execute the *basic self-assembly strategy*. The initial state (left) consists of three s-bots (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 s-bots, i.e. a source of light (the orange circle on the right). After some steps, where the s-bots execute the basic self-assembly strategy, two s-bots finally

[4] `http://www.graphviz.org/`

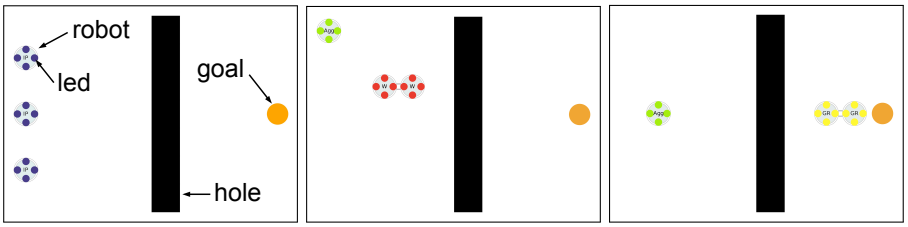

Fig. 6. Three states of a simulation: initial (left), assembly (middle), final (right).

get assembled (middle of Fig. 6). The assembled s-bots can then safely cross the hole and reach the goal (right of Fig. 6), 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 s-bots, 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 s-bots erroneously believing to be part of an assembly, moving into the hole and disappearing. In other simulations we instead noticed pairs of s-bots 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. 8 in Appendix A) 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 led*" (another s-bot willing to be grabbed). Considering the first behavior, an s-bot can change from state `Assembly_Seed` to state `Wait` without another s-bot actually attaching to it. The s-bot then evolves to state `Connected_phototaxis` believing to be assembled with other s-bots. Considering instead the second behaviour, once an s-bot i grabs an s-bot j, i becomes itself "willing to be grabbed" (turning on its red leds) to allow other s-bots 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 s-bots or are they forbidden by real life constraints (e.g. due to the physical structure of the s-bots or to some real-time aspects)? We have still no answer to this question which is subject of current work (in the context of the ASCENS project [3]). Anyway in this phase we observed 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.

5.2 Statistical model checking

A qualitative analysis can provide us proofs that the adaptation strategies can result in some cases to a full success (all s-bots reach the goal) or to a full failure (no s-bots reach the goal). However, in the kind of scenario under study the possibility of full success or failure is typically assumed. The really interesting question is how *probable* are they, respectively?

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 s-bot reaches the goal?*; (P_1) *What is the probability that at least one s-bot reaches it?*; and (P_2) *What is the expected number of s-bots reaching the goal?*.

We have used PVesta [2], a parallel statistical model checker and analysis tool, to perform some comparative analysis. PVesta 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.

A QuaTEx expression is statistically evaluated with respect to two parameters: α and δ. Specifically, expected values are computed from n independent simulations, with n large enough to grant that the size of the $(1-\alpha)100\%$ confidence interval for the expected value is bounded by δ. Intuitively, if a QuaTEx expression is estimated as $\overline{x}$, then, with probability $(1-\alpha)$, the actual expected value belongs to the interval $[(1-\delta)\overline{x}, (1+\delta)\overline{x}]$.

We performed a comparative analysis (wrt. to the above properties) between s-bots equipped differently. We use 4-bots, an abstraction of real s-bots with only 4 LEDs and sensors and capable to move and grip in only 4 directions (up, down, left, right); and 8-bots, which are like 4-bots enriched with LEDs, sensors and movement and gripping capabilities in 4 additional directions (the diagonals). All s-bots execute the same strategy, namely *basic self-assembly response*. The aim was not to derive exact statistical measures but to gain some intuition of the success and performance impact of the different s-bot models. The arena was configured as follows (cf. Fig. 7): an 11×7 grid containing 3 s-bots, the goal (a source of light) and a hole dividing the s-bots from the goal. We remind that an s-bot alone is not able to cross the hole, and hence needs to cooperate (assemble) with other s-bots to cross it.

In general, the 8-bots exhibit a better success rate. Indeed, the only morphology that offers a full success for three 4-bots is a line orthogonal to the hole. In all the other possible morphologies, at least one 4-bot falls in the hole or remains isolated in the left part of the arena.

More precisely, the analysis of P_0 on the 4-bots provides values around 0.5 (i.e. about half of the cases ends up without any 4-bot reaching the goal). For 8-bots this value is approximately half as much.

Regarding the success of at least one s-bot (P_1), the 8-bots exhibit again a much better rate (around 0.8) than the 4-bots (around 0.5).

Finally, the expected number of successful 4-bots (P_2) is about less than one, while for the 8-bots case it is around one and a half.

The obtained results are very reasonable. As we argue above, the less topological constraints are imposed on the s-bots the more flexible their behavior is.

In additional experiments we are trying to accurately estimate the above mentioned (and other) properties, extending our comparative analysis to consider other s-bot features and strategies, and validate our results with the ones reported in [16].

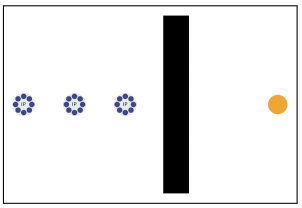

Fig. 7. An initial state.

6 Conclusion

The contributions of our paper are: (i) a description (§ 4) of how to realize in Maude our recently proposed approach to adaptive systems [5] in a simple and natural way; and (ii) a description (§ 5) of how to exploit some Maude tools 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, 3]). 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 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 experimentation we have conducted demonstrates that: (i) Maude is well-suited for prototyping self-assembly systems in early development phases; (ii) simulation can be useful to discover and resolve small ambiguities and bugs in self-assembly strategies; and (iii) statistical model checking can provide rough estimation 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 real challenging question to be tackled is *can we exploit the proposed architecture to implement 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 (e.g. narrowing [8]). 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 challenge of building predictive adaptive systems.

References

1. Agha, G.A., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. In: Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages (QAPL'05). ENTCS, vol. 153 (2), pp. 213–239 (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. LNCS, vol. 6859, pp. 386–392. Springer (2011), `http://musabalturki.net/pvesta`
3. Autonomic Service Component Ensembles (ASCENS), `http://www.ascens-ist.eu`
4. 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. pp. 1029–1033. ACM (2009)
5. Bruni, R., Corradini, A., Lluch Lafuente, A., Gadducci, F., Vandin, A.: A conceptual framework for adaptation. In: Proceedings of 15th the International Conference on Fundamental Aspects of Software Engineering (FASE'12). LNCS, Springer (to appear), `http://eprints.imtlucca.it/1011/`
6. 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)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)
8. Durán, F., Eker, S., Escobar, S., Meseguer, J., Talcott, C.L.: Variants, unification, narrowing, and symbolic reachability in Maude 2.6. In: Schmidt-Schauß, M. (ed.) RTA. LIPIcs, vol. 10, pp. 31–40. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)
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.) Hybrid Systems: Computation and Control, LNCS, vol. 3927, pp. 460–475. Springer Berlin / Heidelberg (2006)
14. Meseguer, J., Talcott, C.: Semantic models for distributed object reflection. In: Magnusson, B. (ed.) Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'2), 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. Auton. 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.) Proceedings of the 1st International Workshop on Self-Adaptive Software (IWSAS'00). 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) (2009)
19. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: Etessami, K., Rajamani, S.K. (eds.) Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05). 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: QEST. pp. 251–252. IEEE Computer Society (2005)
21. Talcott, C.L.: Coordination models based on a formal model of distributed object reflection. ENTCS 150(1), 143–157 (2006)
22. Talcott, C.L.: Policy-based coordination in pagoda: A case study. In: Combined Proceedings of the Second International Workshop on Coordination and Organization (CoOrg 2006) and the Second International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord 2006). Electronic Notes in Theoretical Computer Science, vol. 181, pp. 97 – 112 (2007), `http://pagoda.csl.sri.com/`
23. Weyns, D., Malek, S., Andersson, J.: Forms: a formal reference model for self-adaptation. In: Proceedings of the 7th international conference on Autonomic computing. pp. 205–214. ICAC '10, ACM, New York, NY, USA (2010), `http://doi.acm.org/10.1145/1809049.1809078`
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. pp. 371–380. ACM (2006)

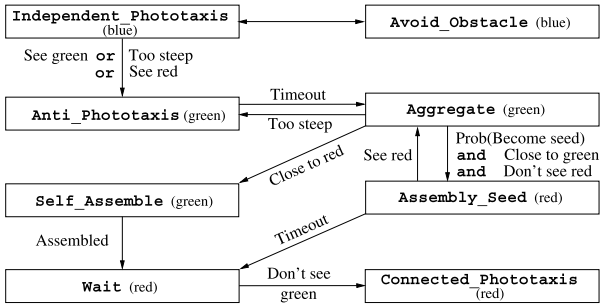

Fig. 8. Excerpt of the basic self-assembly response strategy (borrowed from [16]).

A Basic self-assembly response strategy

We provide here a detailed description of the self-assembly strategy [16].

In § 1 we hinted at various self-assembly strategies for groups of autonomous robots. Then in § 5 we discussed and analysed the "self-assembly response strategy", whose finite state machine is depicted in Fig. 8. Each state contains its name and the color of the leds turned on in that state, while transitions are labelled with their firing condition. Actually Fig. 8 represents an instantiation of the strategy for the hill crossing task, hence the condition "too steep" should be replaced by "see hole".

This controller is executed independently in each individual s-bot (a concrete one in [16], or a software abstractions 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 eventual obstacles (e.g. walls or other robots).

If an s-bot detects a hole (through its infrared ground sensors), or sees a green or red s-bot, then it switches to state `Anti_Phototaxis`, i.e. it illuminates its green LEDs and retreats away from the hole.

After the expiration of a timeout, the s-bot passes in state `Aggregate`: it randomly moves searching for (preferably) a red 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`. In the case in which it instead 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 in state `Wait`, unless it sees another red s-bot, in which case it reverts to state `Aggregate`.

If a s-bot in state `Aggregate` sees a red s-bot, then it switches in state `Self_Assemble` and tries to grab it. Once assembled, the s-bot passes in state `Wait` and turns on its red LEDs. Once no green s-bots are visible, assembled "waiting" s-bots switch to state `Connected_Phototaxis` and navigate to the light source.

B Source code

We provide here some significant fragments of our implementation.

B.1 Adaptive components

We first show how s-bots are represented as Maude terms of sort `Object`:

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

As depicted in Fig. 3, each s-bot has four (main) attributes: `K` (the Knowledge), `R` (the Rules), `E` (the Effects) and `M` (the Managed component). In particular, this last one is itself a term of sort `Object` (and hence a component), allowing to implement the three layered architecture depicted in Fig. 4 and 5. The attribute `M` of the outermost component (an object with class `AC2`) contains an object with class `AC1`, which in turn has an object with class `AC0` in its attributes.

The knowledge, currently represented as a plain set of facts, models the awareness that a component has of the environment. Effects are similar to the knowledge, as they are represented as a plain set of predicates. What differentiates these two attributes is their role respect to the outer layers: if the knowledge can be seen as a sort of input mechanism with the outer component (or the environment for the outermost component), which can filter and elaborate the informations coming from the environment, the Effects can be seen as a sort of output mechanism. For this reason, the effects collect informations about the status of the component (e.g. the color of the LEDs), and about the actions it wants to do, like moving or gripping in a certain direction.

Finally, the rules contain a term of sort `Module`, specifying the behaviour of (the code executed by) the component. The rules are exposed to the outer component, which can hence easily "adapt" the behaviour of the managed component by modifying them.

B.2 Rules

In order to give an idea about how the flow of execution and information depicted in Fig. 4 are actually implemented, we now exemplify a rule for each of the three layers.

Layer 0 The inner components are characterized by the class identifier `AC0`. As explained in §4.1, this layer models the core functionalities of a robot. Hence rules are used to model basic movements, color emissions and gripper's action. The following is a rule through which the component computes the set of directions towards which it can move, that is directions corresponding to neighbour cells without obstacles (walls or other s-bots):

```
rl < oid : AC0 | K: k, E: e                              , A1 >
=> < oid : AC0 | K: k, E: e canMoveTo(possibleMoves(K)) , A1 > .
```

Layer 1 Objects with class `AC1` are components of layer 1. As explained in §4.1, this level corresponds to the single states of the state machine depicted in Fig. 8 of Appendix A. The following is a rule allowing to move towards the light's direction only, implementing part of the logic of state `Independent_Phototaxis`:

```
crl < oid1 : AC1 | K: k1, E: e1        ,
       M: < oid0 : AC0 | K: k0 , E: e0, R: m0, A0  >, A1 >
 => < oid1 : AC1 | K: k1, E: e1 go(dir),
       M: < oid0 : AC0 | K: k0b, E: e0, R: m0, A0b >, A1 >
 if (< oid0 : AC0 | K: k0b, E: e0 canMoveTo(freeDirs), A0b >)
 := reach(< oid0 : AC0 | K: updateK-1To0(K1,K0), E: e0, A0 >, m0)
 /\ possibleDirs := intersection(dirs , oppposite(dirsToLight(K1)))
 /\ possibleDirs =/= empty
 /\ dir := uniformlyChooseDir(possibleDirs, | possibleDirs |) .
```

Layer 2 As explained in §4.1, this is the level of the adaptation manager, which decides how to react to changes in the environment by activating the corresponding basic controller. Considering the self-assembly strategy presented in [16], and whose state machine is depicted in Fig. 8 of Appendix A, if components of layer 1 correspond to the single states of the state machine, this layer's components correspond instead to the state machine itself, and in particular take care of the transitions between its states. In doing so they monitor the environment and their managed components `M`, and eventually executes an adaptation phase changing the rules of `M`.

The following is a rule of this level. The presence of a message with content `generateNextEffect` allows components of layer `AC2` to perform a step of evolution. Following the MAPE-K architecture, it uses the operation `adaptationPhase` to monitor and analyze the current state of the managed component. The operation reduces in the managed component resulting after the eventual phase of adaptation. The managed component is then executed, and its effects are propagated to the ones of level 2.

```
crl { gt | SL } (oid2 <- generateNextEffect)
    { < cellId : Cell | content: < oid2 : AC2 | K: k2, E: e2,
         M: < oid1 : AC1 | K: k1, E: e1, R: m1, A1 >, A2 >, AttrsCell >
      remainingGrid }
   =>
    insert({ gt | SL },
           [gt + time(effect), ORCHESTRATOR <- execute(oid2,effect)])
    { < cellId : Cell | content: < oid2 : AC2 | K: k2A, E: e2A,
          M: < oid1 : AC1 | K: k1b, E: e1A, R: M1A, A1b >, A2A >
        AttrsCell >
      remainingGrid }
  if  --- compute adaptation phase
    < oid2 : AC2 | K: k2A, E: e2A,
        M: < oid1 : AC1 | K: k1A, E: e1A, R: m1A, A1A >, A2A > :=
    adaptationPhase(< oid2 : AC2 | K: k2, E: e2,
        M: < oid1 : AC1 | K: k1, E: E1, R: m1, A1 >, A2 >)
      --- execute the adapted managed component
  /\ < oid1 : AC1 | K: k1b, E: (E1A effect), A1b > := reach(
     < oid1 : AC1 | K: updateK-2To1(k2A,k1A), E: e1A, A1A  >, m1A) .
```

We do not show it here, but the effects of the components of level `AC2` are handled by another entity, the `Orchestrator` having the responsibility of making interact the s-bots with the grid.

B.3 QuaTEx

We conclude this appendix with an example of quantified temporal expressions, and in particular the one to estimate the expected number of s-bots reaching the goal.

First of all we defined the state predicate `completed : Configuration -> Float`, reducing to 1.0 for terminal (absorbing) states, and to 0.0 otherwise. A terminal stae 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 s-bots that currently reached the goal.

Then we defined the equations necessary to PVesta to access such predicates

```
eq val(0,C) = completed(C) .
eq val(1,C) = countRobotInGoal(C) .
```

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() ] ;
```

Such expression tells to PVesta to run the simulation until a terminal state is reached, and then to return the number of robots in goal in the obtained terminal state.