# Modelling and Analyzing Adaptive Self-Assembly Strategies with Maude

Roberto Bruni, Andrea Corradini, Fabio Gadducci

*Dipartimento di Informatica, Università di Pisa, Italy*
*largo Pontecorvo 3c, 56127 Pisa, Italia*
{bruni,andrea,gadducci}@di.unipi.it

Alberto Lluch Lafuente, Andrea Vandin

*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, validate and analyse a prominent example of adaptive system: robot swarms equipped with self-assembly strategies. The analysis exploits the statistical model checker PVeStA.

*Keywords:*
Adaptation, Autonomic Computing, Self-assembly, Ensembles, Maude, Reflective Russian Dolls, Statistical Model Checking, PVeStA

## 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

Research supported by the European Integrated Project 257414 ASCENS.

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 black and white distinction between failure and success, because the non-deterministic 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.

We address here some of the above concerns, presenting the methodology we have devised for prototyping well-engineered self-adaptive components. Such methodology consists of a generic hierarchical and reflective architecture (§4.2) to be instantiated for specific systems or scenarios (§5). Our main case study consists of modelling (§6), debugging (§7.1), and analyzing and comparing (§7.2) self-assembly strategies of robots cooperating for different purposes, including *morphogenesis* (where robots assemble to form predefined shapes §6.1), *obstacle avoidance* (e.g. hole-crossing or hill-climbing while navigating towards a light source §6.2), and *collective healing* (where some robots cooperate to fix other broken robots §6.3).

We specified such robotic scenarios with PMaude [1] (a probabilistic extension of Maude [2]), exploiting on one hand the Reflective Russian Dolls (RRD) model [3] and on the other hand the conceptual framework for adaptation that we proposed in [4], which provides simple but precise guidelines for a clean structuring of self-adaptive systems. Some of the modelled and analyzed self-assembly strategies have been taken from the literature (e.g. from [5]), and the comparison of their performances was carried out with the parallel statistical model checker PVeStA [6].

*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, self-assembly and self-healing, as discussed e.g. in [7]). 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 [8]. 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, 7, 10], FORMS [11], the adaptation patterns of [12], and the already mentioned RRD [3]), to approaches based on model-based development [13] or model transformation [14], to theoretical frameworks based on category theory [15] or stream-based systems [16]. A detailed discussion of some of these and other related works is provided in §8.

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 [4], 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 is modified at run time. We discuss further this topic in §3.

*Is Maude a convenient setting to study self-adaptation?* A "convenient" framework should 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 such systems. There are several reasons why we think that Maude [2] is a good candidate. First, the versatility of rewrite theories can offer the right level of abstraction for addressing the specification, modelling and analysis of self-adaptive systems and their environment within one single coherent framework. Second, since Maude is a rule-based approach, the control data can be expressed naturally as a subset 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 with the RRD model [17]. Third, the conceptual framework for adaptation described in [4], to be further elaborated in §4, facilitates early and rapid prototyping of self-adaptive systems. 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 parallel statistical model checker PVeStA [6] is useful, because it allows to conduct quantitative analysis, parametric with respect to the desired level of statistical confidence.

Pragmatically, the possibility to rapidly develop and simulate self-adaptive systems, and to quantitatively analyze their behaviours at the early stages of software development is very important for case studies like the robotic scenarios 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 (six hours or more for each run). Additionally, only a limited number of robots is typically available (e.g. in [5] only 6 out of the 25 existing s-bots were used) 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, even if performed on a quite abstract representation of the real system, can at least speed-up testing and debugging, and could dispense the programmers from coding lowest-performance strategies.

*Contribution and Synopsis.* In §2 we present the robotic scenario and the self-assembly strategy that will be used as main case study and running example along the paper. In §3 we summarize the conceptual framework for adaptation that we proposed in [4], and that we exploited in the design of the self-assembly strategies in order to stress their adaptive features. The general guidelines and principles used in Maude for modelling self-adaptive systems (including logical reflection) are briefly described in §4, together with the conceptual, hierarchical architecture based on the Reflective Russian Dolls model that we adopted.

This architecture is instantiated to an implementation of the self-assembly strategy of the case study in §5, based on *MESSI* (Maude Ensemble Strategies Simulator and Inquirer) [18]. We first describe conceptually the role of each of the three layers we use, and of the execution environment. Next we provide details of the Maude implementation, including samples of code of the various layers, and stressing the modularity of the methodology. Such modularity is fully exploited in §6, where we validate our approach by presenting four more strategies for three distinct scenarios, heavily reusing components developed for the running example. Next in §7 we describe how to exploit our approach for the analysis of self-assembly strategies. In particular, we first show how it is possible to debug self-assembly strategies resorting to single simulations (and to the automatically generated animations), and we show how to evaluate the performance of strategies via statistical model checking. Interestingly, in §7.1 we discuss how we exploited the debugging capabilities to find some *bizarre behaviours* in the strategy of [5]. Instead, in §7.2 we show how to exploit statistical model checking to ensure that we get rid of such behaviours in our variants of the strategy, and to compare the performances of some of the implemented strategies. In §8 we discuss some approaches that have influenced and inspired our work. This section covers most of the distinguishing features of our approach, serving as an argument to convince the reader about its suitability. Finally, in §9 we present some concluding remarks and we hint at ongoing research avenues.

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

A preliminary version of the present paper was published as [19]. The initial work presented there evolved into the platform *MESSI* [18], designed for early prototyping of self-assembly strategies, their debugging and analysis. The modularity, applicability and accessibility of the framework was improved by providing a library of generic controllers implementing basic behaviours, like "*move towards light*", or "*assembly to the source of an admissible color emission*". Intuitively, self-assembly strategies can be defined easily just by specifying transitions (adaptations) between states executing different controllers, and this is exploited here to present additional strategies w.r.t. [19]. In particular, this is reflected by the totally revised §5, and by the novel §6. Moreover, in [19] we discussed only some preliminary analysis result of a single strategy, while in the revised §7 we show how to exploit our approach to properly analize and compare the performances of several self-assembly strategies. Finally, also the discussion of related work in §8 is original to this contribution.

## 2. Case Study: Self-Assembling Robot Swarms

Self-assembling robotic systems are formed by independent robots capable to connect physically to form *assemblies* (or *ensembles* in our terminology) 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-assembling robot swarm a challenging scenario to engineer.

In [5], 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 steepness of the hill, the width of the hole, or the location of the robot to be rescued, the robots must self-assemble because unable to complete their tasks 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". Additionally, in order to demonstrate the versatility of our approach, in §6 we present strategies for other scenarios.

The experiments described in [5] were conducted on the SWARM-BOT robotic platform [20], 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 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 intention to create a new ensemble. The ring can also be grasped by other s-bots thanks to a gripper-based mechanism. From [5], we know that the s-bots have a maximal speed of 30 cm/s and a diameter of 12 cm. Moreover, each robot is able to perceive (with an acceptable accuracy) six different color emissions: red, green, blue, magenta, yellow and cyan.

Roughly, the several strategies presented in [5] are: (i) the *independent execution* strategy, where s-bots move independently from one another 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 is available, or it becomes the *seed* of a new assembly (red light); (iii) the *preemptive self-assembly* strategy, where the s-bots assemble together independently of the environment and not by emergency as in the basic self-assembly response; (iv) the *connected coordination* strategy, where the s-bots are pre-assembled in a line, and their orientation with respect to the obstacle (hole/hill) is coordinated according to a leader-following architecture.

The experiments reported in [5] concern different strategies in different scenarios (with holes of different size and random initial positions of the s-bots) and were 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 [5] are linked from the web page of the MESSI framework [18].

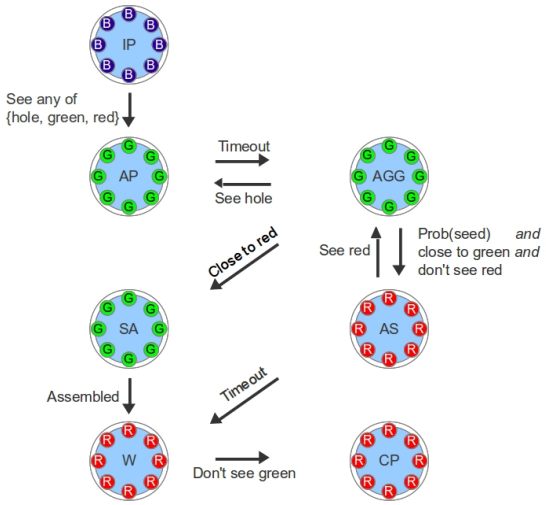


Figure 1: Basic self-assembly response strategy (as proposed in [5]).

*Basic self-assembly response strategy.* We describe here the *basic self-assembly strategy* of [5], on which we will focus as running case study in the rest of the paper.

A finite state machine representing the strategy is depicted in Fig. 1. States are depicted as bird-eye views of an s-bot (the outer circle represents the ring, the inner one represents the body, while the eight small circles on the border represent the LEDs), where we indicate the name of the state (e.g. `IP` or `AS`), and the color of each LED (the letter inside each LED is the initial of its color). Transitions are labelled with their firing condition.

This finite state machine is executed independently in each s-bot (a concrete realization in [5], or a software abstraction in this work). In the starting state `IP` (`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 s-bots). 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 `AP` (`Anti_Phototaxis`), i.e. it turns on its green LEDs and retreats away from the direction of the light.

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

Even if not depicted, we consider a further transition in which s-bots move into state `GoalReached` with yellow LEDs whenever they reach the goal.

## 3. A White-Box Conceptual Framework for Adaptation

Before describing how we modeled and analysed self-assembly scenarios like the one we just presented, let us explain some guidelines that we followed when designing them. 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, as a first step we found it necessary to understand *"when is a software system adaptive"*, by identifying the features distinguishing such systems from ordinary ("non-adaptive") ones, since, unfortunately, there is no general agreement on the notion of adaptivity in general or in software systems, and no general consensus is perceived around a foundational model for adaptivity. We addressed this problem in [4], proposing a simple structural criterion to characterize adaptivity.

According to widely accepted informal definitions, a software system is 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: 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 data from the user). Moreover, this definition is often approached from a *black-box* (i.e. behavioral, observational) perspective. As a paradigmatic example related to our case study, some authors [21] consider that *"obstacle avoidance may count as adaptive behaviour if [...] obstacles appear rarely. [...] If the normal environment is [...] obstacle-rich, then avoidance becomes [...] normal behaviour rather than an adaptation"*. In sum, under such perspective obstacle avoidance can be a form of adaptation in some contexts but not in others. Indeed, such perspective focuses on the point of view of an observer and does not care about the internal mechanisms by which the adaptive behavior is achieved. We argue that such perspective is of little use for design purposes where modularization and reuse are critical aspects.

In order to have a separation of concerns which facilitates distinguishing situations where the modification of behaviour is part of the application logic from those where they realize the adaptation logic, we 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, *variations* in context-oriented programming [22], *policies* in policy-driven languages (e.g. [23]), *aspects* in aspect-oriented languages (e.g. [24]), and even entire programs, in models of computation exhibiting higher-order or reflective features (e.g. [17, 25]).

In [4] we discussed how our simple criterion for adaptivity can be applied to several of the reference models we mentioned in the introduction. For each of them we identified what would be a reasonable choice of control data, so that our notion of adaptation ("modification of control data") coincides with that of the authors. 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 (Fig. 2). Clearly, by our definitions the managed component is adaptive, and the system made of both component and control loop is self-adaptive.

The construction can be iterated, as the control loop itself could be adaptive. Think for example 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. 5, left) as we do in the next sections for robot prototypes.

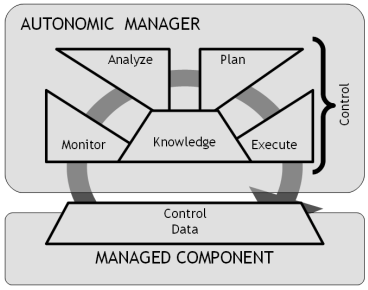


Figure 2: Control data in MAPE.

## 4. Adaptivity in Maude

We start this section arguing again the suitability of Maude and rewriting logic as a language and a model for adaptivity (§4.1), in more detail with respect to the discussion offered in §1. Next, 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 generic framework of [4] based on control data (§4.3).

### *4.1. Maude, Logical Reflection and Adaptivity*

As argued in [17], Rewriting Logic (RL) [26] is well-suited for the specification of adaptive systems, thanks to its reflective capabilities. Indeed, reflection is widely accepted as one of the key instruments to realize self-adaptive software systems [27] since it provides basic and flexible mechanisms for introspection and meta-programming (see also the discussion in §8). The reflection mechanism of rewriting logic yields what is called a *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 [17] to formalize a model for distributed object reflection, suitable for the specification of adaptive systems (see the discussion in §8). 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 (as e.g. in [28]). 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 [17] and at a more general level in [29, 27], where (computational) reflection is promoted as a necessary feature for any self-adaptive software system.

### *4.2. Generic Architecture*

This section describes how we specialize the RRD architecture for modelling adaptive components (similar approaches and alternatives are discussed in §8). We focus on the structure of the layers and on the interactions among them, abstracting from the considered case study.

*4.2.1. Intra-layer architecture*

Each layer is a component having the structure illustrated in Fig. 3. 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. In fact we will consider the rules $R$ as the *control data* of a layer.

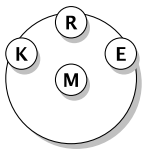

Figure 3: Intra-layer.

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 of the component 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 our conceptual framework (see §3).

*4.2.2. Inter-layer architecture*

Layers are organized hierarchically: each one contains its knowledge, effects, rules and, in addition, the managed underlying layer. An example with three layers is depicted in the leftmost diagram of Fig. 4. Of course, the architecture does not impose any number of layers. 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 elementary. 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. 4 shows the control and data flow of ordinary behaviors (without adaptations). Knowledge is propagated down to the core (layer 0) and the effects are collected up to the surface (layer 2). This flow of information is governed by the rules of each layer. Knowledge and effects are subject 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. 4 corresponds to a phase of adaptation. Here the outermost layer triggers an adaptation at layer 1. This can be due to some

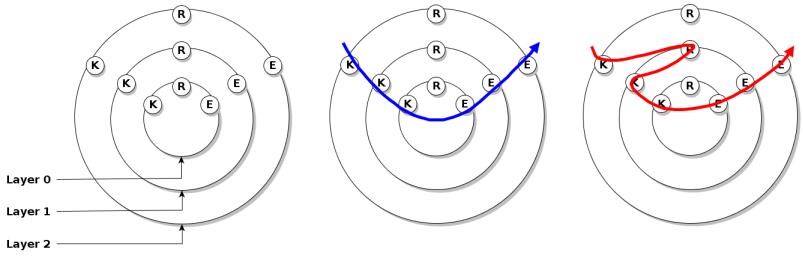


Figure 4: Inter-layer architecture (left), ordinary flow (center), adaptation flow (right).

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 multi-layered architecture just presented with some general frameworks used for modelling 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 behaviours from the standard computations of the system.

Our architecture is a simplified version of the RRD of [17] where each layer is a single object rather than a collection of objects. 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. 5, showing that the rules of each layer implement the MAPE-K control loop on the lower layer.

The generic architecture imposes the encapsulation of all components of the tower. 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. In the next section we discuss in particular how we exploited this last point to facilitate the development of self-assembly strategies by resorting to predefined behaviours to be (re)used as building blocks.

## 5. Concrete Architecture and Case Study Implementation

This section instantiates in Maude the generic architecture shown in §4.2 to scenarios involving s-bots (§5.1), presenting also some significant implementative details (§5.2). We consider here as running case study the basic self-assembly response strategy discussed in §2 (see Fig. 1), while in §6 we see how other scenarios and strategies can be easily designed and implemented.

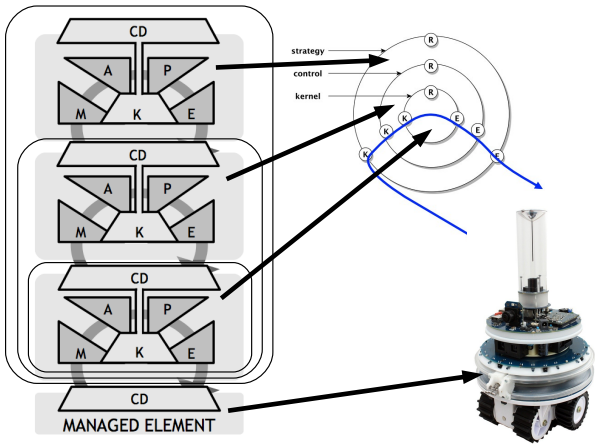


Figure 5: Architecture as an instance of the framework.

*5.1. Instantiation of the Methodology to the Robotic Case Study*

The concrete architecture of our case study has three layers (see. Fig. 5, top-right). Indeed, even if the informal description of [5] does not explicitly mention those layers, the way the s-bot behavior and strategies are described suggests the following three layers.

*Layer 0 (kernel).* This layer models the core functionalities of an s-bot (see [5, §3]). The rules implement basic movements and actioning of the gripper, and are hence given once, independently on the modelled strategy or scenario. Layer 0 corresponds to what some authors call *hardware abstraction layer* (see e.g. [28]).

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

*Layer 2 (adaptation).* This is the layer of the adaptation manager, which reacts to changes in the environment by activating a suitable basic controller. Intuitively, this layer corresponds to an entire state machine modelling a self-assembly strategy (e.g. Fig. 1), and it takes care of the transitions between its states. This is done by constantly monitoring the environment and the managed component $M$ (layer 1), and by executing adaptation phases when needed, which means changing the rules of $M$. More details are provided in §5.2. Other strategies can be implemented by modifying this layer only, as we will show in §6 for variants of the basic self-assembly response strategy, and for strategies to handle other scenarios.

| Name | Brief description |
|---|---|
| IDLE | An idle s-bot: it neither moves, nor grabs. |
| MOVE_TOWARDS_LIGHT | The s-bot moves towards the light source if possible, otherwise it stays idle if hindered. |
| MOVE_PREFERABLY_ TOWARDS_LIGHT | The s-bot moves towards the light source if possible, otherwise it moves in one of the directions without obstacles (e.g. walls, s-bots) |
| MOVE_AWAY_FROM_LIGHT | The s-bot retreats away from the light by randomly choosing one of the free directions opposite to light. |
| MOVE_IN_ANY_DIRECTION | The s-bot navigates in any randomly chosen free direction. |
| GRAB_ADMISSIBLE_LED | The s-bot does not move, but grabs a LED with a grippable color as specified in its knowledge if near enough. If more than one grippable LED is perceived, then the one to be grabbed is randomly chosen. |
| OUTFLANK_EFFECT | The s-bot navigates, if possible, in a randomly chosen free direction not opposite to the ones from where it perceives a given effect, as specified in its knowledge. |

Table 1: Some predefined basic controllers for layer 1

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 s-bots in the surrounding (their LED emissions), and the direction of the light source (the goal). Generable effects include requests of movements or grabbing of s-bots in adjacent cells (handled by the execution environment), as well as color emissions towards a direction (i.e. the color emitted by a LED).

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.

Of course, other layers could be added to the s-bot architecture. For example, a third layer could be encharged of a *meta-adaptation* logic, which triggers an adaptation of layer 2 by changing the adaptation strategy.

*Predefined basic controllers for layer 1.* We provide a library of predefined basic controllers, implementing basic behaviours for layer 1, summarized in Table 1. They will be used as building blocks in the definition of the strategies. The `IDLE` controller implements the trivial behaviour of "idle" s-bots, which do not perform any action, and is typical of waiting states. Controller `MOVE_TOWARDS_LIGHT` implements the behaviour "move towards a goal light source", where an s-bot navigates (if not hindered) towards the direction of the light source. The third basic controller (`MOVE_PREFERABLY_TOWARDS_LIGHT`) is similar to the previous one, but if the directions towards the light source contain obstacles, then movements in other directions are allowed to try to avoid such obstacles. The fourth controller (`MOVE_AWAY_FROM_LIGHT`) allows to retract away from the light source (i.e. to move in directions opposite to the ones from which the light source is perceived), while `MOVE_IN_ANY_DIRECTION` allows to randomly move in any direction not containing obstacles. The behaviour "grip a LED of a given color" (controller `GRAB_ADMISSIBLE_LED`) prohibits movements but allows to grip other s-bots (or

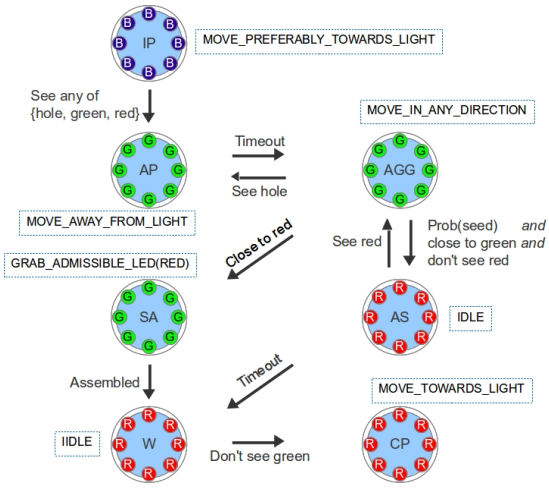


Figure 6: Basic self-assembly response strategy implemented with predefined controllers.

other sources of color emissions) if near enough, in order to form ensembles. The set of *grippable colors* has to be explicitly specified in the knowledge of the s-bot (e.g. by providing `grippableLeds(red)` we allow s-bots to grip red LED emissions).

The last controller `OUTFLANK_EFFECT` implements a behaviour that allows to *outflank* a given color emission, i.e. to *move without departing from the perceived effect*. It can be used in scenarios where constraints are imposed on the morphology of the ensemble, like for two of the strategies presented in §6, where we are interested in creating line-shaped ensembles. In these cases, a suitable color can be used to inform an s-bot that it is near to the ensemble, but it has to reach the tail of the line before connecting to it. Hence this controller can be used to drive an s-bot towards a given part of an ensemble. As for the case of the grippable colors, the set of effects to be outflanked has to be specified in the knowledge of the s-bot (e.g. by providing `outflank(magenta)` we allow s-bots to outflank magenta LED emissions).

The controllers just described were used to implement the states of the state machines modelling the self-assembly strategies. For example, the state `IP` (`Independent_Phototaxis`) of Fig. 1 is governed by the basic controller `MOVE_PREFERABLY_TOWARDS_LIGHT`, while state `SA` (`Self_Assemble`) is governed by `GRAB_ADMISSIBLE_LED`. More in detail, the correspondence between the states of the basic self-assembly response strategy of Fig. 1 and our basic controllers (Table 1) is depicted in Fig. 6. Further details are provided in §5.2.

In §6 we will show how the available controllers facilitate the implementation of other strategies, both for the same hole-crossing scenario and for others. In fact it will be sufficient to specify the transitions among states according to the strategy's logic.

*Execution environment.* The execution environment of the s-bots is realized by a discrete-event 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 an s-bot. For instance, it decides if an s-bot can actually move towards the direction it is willing to move (indicated by the effects emitted by the outermost layer of a component).

The scheduler, implemented as an ordinary discrete-event scheduler, activates the scheduled events, allowing an s-bot 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 $c$ at time $t$" 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 at time $t'$" for $c$.

Finally, the arena defines the scenario where s-bots 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 s-bot, meaning that the s-bot is in that position of the arena. Each s-bot can move to an adjacent cell, or perform an action (i.e. grip) in eight possible directions (up, down, left, right and the four diagonals).

### *5.2. Implementation Details*

In this section we detail the implementation of *MESSI* (Maude Ensemble Strategies Simulator and Inquirer) [18], a Maude instantiation of our methodology which allows: (*i*) to model self-assembly strategies, (*ii*) to debug them via animations produced from discrete-event simulations, and (*iii*) to statistically estimate their quantitative properties relying on the parallel statistical model checker PVeStA.

#### *5.2.1. On the structure of adaptive components*

Our implementation, similarly to the systems described in [17], relies on Maude's object-like signature (see [2, 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 an s-bot have the same identifier: in terms of [17] we use *homunculus objects*.

Therefore, a sample s-bot can have the overall structure of Listing 1.

Listing 1: A sample s-bot specification

```
1  < c(0) : AC2 | K: gripper(open) on(right,none) towards(right,light) ...,
2                 E: emitt(up,Green) go(right) ...,
3                 R: mod_is_sorts_._____endm,
4                 M: < c(0) : AC1 | K: ..., E: ..., R: ...,
5                                   M: < c(0) : AC0 | K:..., E:..., R:... >
6                      >
7   >
```

*5.2.2. On the structure of the simulator*

The arena is implemented as a multi-set of objects of class `Cell`, each of which may contain in the attributes (at most) one object of class `AC2` representing an s-bot. The orchestrator implements the movement of an s-bot by passing it from the cell in which it is stored to one of the eight adjacent cells. This way the s-bots 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.

Intuitively, the cell encapsulating an s-bot may be seen as a layer wrapping objects 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 s-bot's LEDs), and of handling the interactions between the s-bot and the scheduler.

*5.2.3. Rules of adaptive components*

The behaviour of each layer is specified by the rules contained in its `R` attribute, 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 rewriting features), monitoring and adaptation (by examining and modifying the meta-representation of modules). In fact, on one hand a basic controlling strategy can be implemented by a single generic meta-rule to *self-execute* an object: the object with rules $R$ proceeds by executing $R$ in its meta-representation. On the other hand, more sophisticated control strategies can be realized: since rules are exposed to the outer component, these can execute or manipulate them, and analyse the outcome. This way, the wrapping component can act as a planner or decision-taker.

In order to give an idea on how the flows of execution and information of Fig. 4 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 s-bots. For example, the rule of Listing 2 computes the set of directions towards which an s-bot can move.

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 $\sigma$, and in this case the application consists of replacing the matched sub-term with the term obtained

Listing 2: A rule of layer 0 to compute the set of free directions

```
1  rl [admissibleMovements] :
2     < oid : AC0 | K: 1Step k0, E: e0                          , A0 >
3  => < oid : AC0 | K:       k0, E: e0 canMoveTo(freeDirs(k0)), A0 > .
```

Listing 3: A rule of layer 1 to compute a direction towards which to move

```
1  crl [MovePreferablyTowardsLight]:
2      < oid : AC1 | K: 1Step k1, E: e1        ,
3                    M: < oid : AC0 | K: k0 , E: e0, R: m0, A0  >, A1 >
4   => < oid : AC1 | K:       k1, E: e1 go(dir),
5                    M: < oid : AC0 | K: k0b, E: e0, R: m0, A0b >, A1 >
6  if < oid : AC0 | K: k0b, E: e0 canMoveTo(freeDirs), A0b > := execute(
7     < oid : AC0 | K: 1Step update1To0(k1,k0), E: e0, A0 >, m0)
8  /\ preferredDirs := intersection(freeDirs, dirsToLight(k1))
9  /\ dir := uniformlyChooseDir(preferredDirs, freeDirs) .
```

by applying $\sigma$ 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` allows to rewrite an object of class `AC0` to itself, enriching its effects with the equational evaluation of `canMoveTo(freeDirs(k0))`. Notice that the constant `1Step` 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 reduction. 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. Hence, `freeDirs` is reduced to the set of directions towards which the s-bot can move, i.e. those not containing obstacles. Operator `canMoveTo` instead is a constructor, hence it cannot be reduced.

*Layer 1.* Objects of class `AC1` correspond to components of layer 1, implementing the basic controllers of Table 1 which correspond to the individual states of the state machine of Fig. 1. Rules of this layer can execute the component of the lower level (an object of class `AC0`) providing additional knowledge to it, and then elaborating the resulting effects. The rule of Listing 3 implements (part of) the logic of state `IP`, computing the desired direction towards which to move.

Listing 3 shows a conditional rule, as evident from the keyword `crl` and the `if` clause following the RHS in line 6. 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.

The operator `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 `MovePreferablyTowardsLight`, the operator `execute` will apply a single

Listing 4: A rule of layer 2 to compute adaptation and execution phases of layer 1

```
1  crl [adaptAndExecute]:
2     < oid : AC2 | K: nextEffect k2 , E: e2,
3                   M: < oid : AC1 | K: k1 , E: e1 , R: m1 , A1  >, A2  >
4  => < oid : AC2 | K:             k2A, E: e2A schedule(event(oid,effect)),
5                   M: < oid : AC1 | K: k1b, E: e1A, R: m1A, A1b >, A2A >
6  if < oid : AC2 | K: k2A, E: e2A,
7           M: < oid : AC1 | K: k1A, E: e1A, R: m1A, A1A >, A2A > := adapt(
8     < oid : AC2 | K: k2 , E: e2 ,
9           M: < oid : AC1 | K: k1 , E: e1 , R: m1 , A1  >, A2  > )
10 /\ < oid : AC1 | K: k1b, E: e1A effect, A1b > := execute(
11    < oid : AC1 | K: 1Step update2To1(k2A,k1A), E: e1A, A1A >, m1A ) .
```

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` (for example, 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 `MovePreferablyTowardsLight` 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 stored in the attributes 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. The idea is that the hardware abstraction layer remains constant since in the considered scenarios the hardware of the s-bots does not change.

*Layer 2.* A component of this layer corresponds to an entire state machine of a self-assembly strategy (e.g. Fig. 1). It monitors the environment, and triggers single transitions of the managed component (layer 1), like movements and gripper actions. Transitions of the managed component are performed by executing it with the rules stored in its attribute `R`.

If necessary, this layer also enforces adaptation phases, that is transitions from the current state of the (finite state machine of the) self-assembly strategy to a new one, by changing the rules of the managed component, and the colors of the LEDs. As discussed later, by *changing the rules of the managed component* we mean that the layer 2 substitutes the attribute `R` of layer 1 with a new one.

Listing 4 contains the main rule governing this layer. The rule is triggered by the token `nextEffect` (line 2), generated by the scheduler and propagated to the s-bot by the cell containing it. The execution of the rule consists of

Listing 5: One of the equations implementing the strategy of Fig. 6 (AGG → SA)

```
1  ceq [AggToSA]:
2    adapt(< oid2 : AC2 | K: state(AGG) k2, E: e2                ,
3                         M: < oid1 : AC1 | R: m1 , E: e1   , A1 > , A2 >)
4  = adapt(< oid2 : AC2 | K: state(SA)  k2, E: setAllLEDs(green),
5                         M: < oid1 : AC1 | R: m1b, E: none, A1 > , A2 >)
6   if seeEffect(led(red),k2)
7   /\ m1b  := upModule('GRAB_ADMISSIBLE_LED,false) .
```

Listing 6: The reduction of adapt in case no adaptation is needed

```
1  eq [noAdaptationNeeded]: adapt(obj) = obj [ owise ] .
```

an adaptation phase (lines 6-9) followed by an execution phase (lines 10-11), 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 `adapt`, using the knowledge of layer 2 (`k2`) to enact a transition to a new state of the strategy, if necessary. Among the ones defining the operation, the equation exemplified in Listing 5 encodes the transition of Fig. 1 from state `Aggregate` to state `Self_Assemble`, labeled with `Close to red`.

The conditional equation states that if an s-bot in state `Aggregate` sees in its neighborhood an s-bot 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 whose name is passed as first parameter. In this case the relevant module is called `GRAB_ADMISSIBLE_LED`, and it contains the rules defining the controller for layer 1 with the same name, which is used in state `Self_Assemble` as shown in Fig. 6. Notice also how, in line 4, we intentionally keep the operation `adapt`. This allows to consecutively compute more than one adaptation phase if necessary. Termination is of course guaranteed.

It is now easy to understand that in order to implement a self-assembly strategy we only have to specify an equation like the one in Listing 5 for every transition in the corresponding state machine, plus the one in Listing 6 for the case in which no adaptation is required, where `owise` is a special attribute that tells the Maude interpreter to apply the equation only if none of the others can be applied.

It is worth to remark that this solution works for those strategies that are free of non-determinism. In fact the absence of non-determinism allows us to implement adaptation strategy as a function (`adapt`), defined by a set of confluent and terminating equations. The general case of non-deterministic strategies can be easily handled by lifting the function `adapt` to *sets* of possible future states and then implementing a function that takes the decision by solving the non-determinism on the basis of suitable criteria (e.g. randomly or using

Listing 7: Some of the predefined basic controllers of Table 1

```
1  mod IDLE is
2    pr AC1 .
3  endm
4
5  mod MOVE_PREFERABLY_TOWARDS_LIGHT is
6    pr AC1 .
7
8    var k1 k0 k0b e1 e0 e0b : Config . var m0 : Module .
9    vars A0 A0b A1 : AttributeSet . var oid : Oid .
10   var freeDirs preferredDirs : Set{Direction} . var dir : Direction .
11
12   crl [MovePreferablyTowardsLight]:
13      < oid : AC1 | K: 1Step k1, E: e1        ,
14                    M: < oid : AC0 | K: k0 , E: e0, R: m0, A0  >, A1 >
15   => < oid : AC1 | K:       k1, E: e1 go(dir),
16                    M: < oid : AC0 | K: k0b, E: e0, R: m0, A0b >, A1 >
17   if < oid : AC0 | K: k0b, E: e0 canMoveTo(freeDirs), A0b > := execute(
18      < oid : AC0 | K: 1Step update1To0(k1,k0), E: e0, A0 >, m0)
19   /\ preferredDirs := intersection(freeDirs, dirsToLight(k1))
20   /\ dir := uniformlyChooseDir(preferredDirs, freeDirs) .
21 endm
22
23 mod GRAB_ADMISSIBLE_LED is
24   pr AC1 .
25
26   var k1 k0 k0b e1 e0 e0b : Config . var m0 : Module .
27   vars A0 A0b A1 : AttributeSet . var oid : Oid .
28   var dirsWithEmiss : Set{Direction} . var dir : Direction .
29
30   crl [gripColorEmission] :
31      < oid : AC1 | K: 1Step k1, E: e1,
32                    M: < oid : AC0 | K: k0 , E: e0 , R: m0, A0  > , A1 >
33   => < oid : AC1 | K:       k1, E: insertEffect(attach(dir),e1),
34                    M: < oid : AC0 | K: k0b, E: e0b, R: m0, A0b > , A1 >
35   if < oid : AC0 | K: k0b, E: e0b  canAttachTo(dirsWithEmiss), A0b > :=
          execute(< oid0 : AC0 | K: 1Step update1To0(k1,k0), E: e0, A0  > , m0)
36   /\ dirsWithEmiss =/= empty
37   /\ dir := uniformlyChooseDir(dirsWithEmiss, | dirsWithEmiss |) .
38 endm
```

preference functions).

Coming back to the rule of Listing 4, 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 the s-bot will propagate it to the scheduler.

*5.2.4. Implementation of the predefined basic controllers for layer 1*

In this section we discuss in more detail the implementation of our predefined basic controllers summarized in Table 1. Each of these controllers has been defined in its own Maude module (e.g. the controller `GRAB_ADMISSIBLE_LED` is defined in a module `mod GRAB_ADMISSIBLE_LED is ... endm`).

Listing 7 exemplifies the implementation of some of the controllers. In lines `1-3` we find the simplest one, `IDLE`, which as expected has a trivial code. Lines `5`

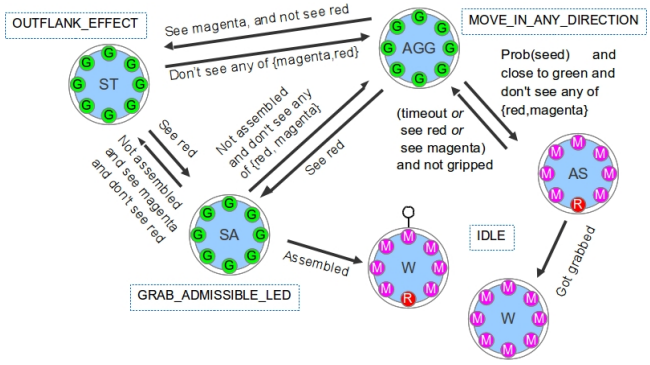


Figure 7: An assembly strategy to form ensembles shaped as lines.

to 21 show the code of controller `MOVE_PREFERABLY_TOWARDS_LIGHT`: we already discussed its rule (also shown in Listing 3), but the code shows the syntax of Maude modules and the needed variable declarations. Finally, lines 23 to 37 show the code of the controller `GRAB_ADMISSIBLE_LED`. This controller is typical of states in which the s-bot does not perform movements, but tries to grip other s-bots. In particular, in line 35 we first execute the component of layer 0, providing to it the `1Step` token, and updating its knowledge with the (filtered part of the) one of layer 1. In this way we obtain `dirsWithEmiss`, that is the set of directions corresponding to the surrounding cells from which it is possible to sense a given color emission (in this case red, which is specified in the knowledge of layer 0). Then, if this set is not empty (line 36), in line 37 we randomly select one of the directions. Finally, in line 33 this direction is inserted in the effects of layer 1, effect which will be propagated up to the scheduler by layer 2 and the cell containing the s-bot.

## 6. Methodology Validation: Design of Self-Assembly Strategies

In this section we demonstrate the wide applicability and accessibility of our general architecture, and its instantiation MESSI, by providing some examples of implemented self-assembly strategies for several scenarios. The aim of this section is not to propose new self-assembly strategies, but rather to show the versatility of our framework and methodology. In particular, we show how it is possible to exploit the predefined basic controllers of Table 1 in the definition of self-assembly strategies: since the basic behaviours are already implemented, only the state transitions (i.e. adaptations) of the strategies have to be defined.

We will first present the strategy depicted in Fig. 7, having the goal of forming line-shaped ensembles (a typical example of *morphogenesis problems* [30]). Next we will consider two variations of the strategy for the hole-crossing scenario, and a strategy for collective healing scenarios, where groups of s-bots cooperate to fix broken ones.

*6.1. A Self-Assembly Strategy for Line-Forming Scenarios*

There exists a wide literature regarding morphogenesis (e.g. [30, 31, 32, 33, 34, 35, 36]), i.e. where robots (autonomously) self-assemble in given morphologies in order to solve specific tasks.

In this section we illustrate a simple self-assembly strategy, depicted in Fig. 7, to accomplish the task of forming line-shaped ensembles. Intuitively, in this scenario there are no obstacles around, and three colors are used to guide the composition of ensembles: the green color stands for "*the s-bot wants to grip another s-bot*", red stands for "*the s-bot wants to be grabbed*" and magenta stands for "*the s-bot is part of an ensemble, but other s-bots should not connect to this ensemble here, as they should search for the tail of the line*".

The s-bots are initialized in state `Aggregate`, randomly moving in any direction (controller `MOVE_IN_ANY_DIRECTION`). If a red color emission is sensed, then the s-bot grabs it by changing its status to `Self_Assemble` with controller `GRAB_ADMISSIBLE_LED` and keeping the LEDs green. If the gripping action succeeds, the s-bot has to wait for other s-bots to grab it, hence it changes its status to `Wait` and sets the `IDLE` controller. Interestingly, only the LED opposite to the direction of the gripper is set to red, while the others are set to magenta: this will force the generation of ensembles shaped as lines. Noteworthy, if an s-bot in state `Aggregate` does not see any red color emission, but sees a magenta one, then it knows that it is near to an ensemble and that it has to search for the right point to grip it (the tail of the line). This behaviour is offered by the basic controller `OUTFLANK_EFFECT`: it allows to move in randomly chosen directions, as long as the movement allows to remain near to the ensemble (i.e. a magenta color emission). Looking at Fig. 7, the transition outgoing from state `Aggregate`, and labeled with "`See magenta, and not see red`" goes in state `Search_Tail`, with controller `OUTFLANK_EFFECT`.

If an s-bot in state `Self_assembly` does not perceive any assemblies anymore (i.e. neither red nor magenta color emissions), then the gripping action cannot be accomplished, and the robot reverts in state `Aggregate`. If instead a magenta color emission is still perceived (but not a red one), then the s-bot moves in state `Search_Tail` to search for the tail of the assembly.

Finally, the last outgoing transition from `Aggregate` regards the case in which the s-bot does not perceive any ensemble (red or magenta color emissions), while it perceives single s-bots (green color emissions). Then, with probability `Prob(seed)` the s-bot becomes the seed of a new ensemble, and waits for other s-bots to grip it: it changes its state to `Assembly_Seed` with controller `IDLE`. Noteworthy, it sets a randomly selected LED to red (which other s-bots are allowed to grab) while the others are setted to magenta. The strategy has a last transition from state `Assembly_Seed` to state `Aggregate` with firing condition "`(timeout or see red or see magenta) and not gripped`". The idea is that, potentially, any s-bot could become the seed of its own ensemble, a case which is clearly to be avoided. Then, in order to limit the number of ensembles, an s-bot in state `Assembly_Seed` can revert to state `Aggregate` if it understands that the choice of generating a new ensemble was unfortunate. Namely, this

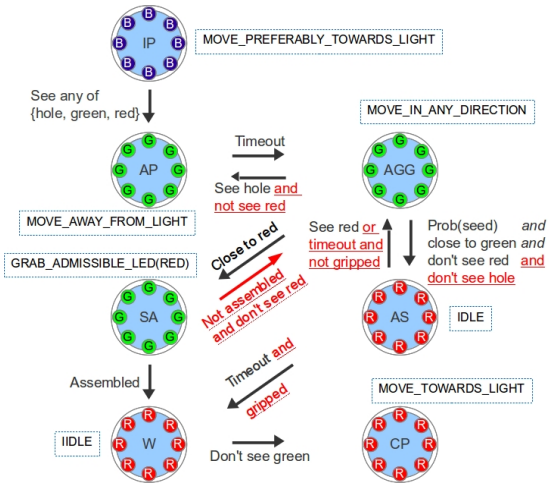


Figure 8: A variant of the basic self-assembly response strategy to deal with *bizarre behaviours*.

transition is allowed when the s-bot has not been connected, and either a fixed amount of time elapsed (`timeout`), or another ensemble is perceived (`see red or see magenta`).

### *6.2. Self-Assembly Strategies for Hole-Crossing Scenarios*

In this section we discuss two variants of the basic self-assembly response strategy of Fig. 6. The first one, depicted in Fig. 8, is just a mild variation where we enrich some of the firing conditions of its transitions, and add a new transition from state `Self_Assemble` to state `Aggregate`. The differences with respect to the original strategy are highlighted with red color and underscore. Intuitively, this variant solves some bizarre behaviours that we found in the basic self-assembly response strategy. A brief description of this variant and the motivations for its proposal are given in §7.1.

The strategy of Fig. 9 is instead a major variation. It is obtained composing our variant of the basic self-assembly response strategy (Fig. 8) with the one depicted in Fig. 7 for the line-forming scenario: s-bots self-assemble in lines to cross the hole. As argued in [5], the idea is that if s-bots compose in lines, then there are more chances to succeed in crossing obstacles (i.e. holes or hills). This is a quite intuitive reasoning which we evaluate in §7.2 by comparing the performances of the three self-assembly strategies (Fig. 6, 8, 9) for the hole-crossing scenario.

As for the case of the line-forming strategy (Fig. 7), the red color stands for *I want to be gripped*, green stands for *I want to grip an ensemble*, and magenta stands for *I am part of an ensemble, but this is not the right position to grip it*.

More in detail, the first two transitions (i.e. from state `IP` to `AP`, and from `AP` to `AGG`) are taken from the strategy of Fig. 8, the only difference is that

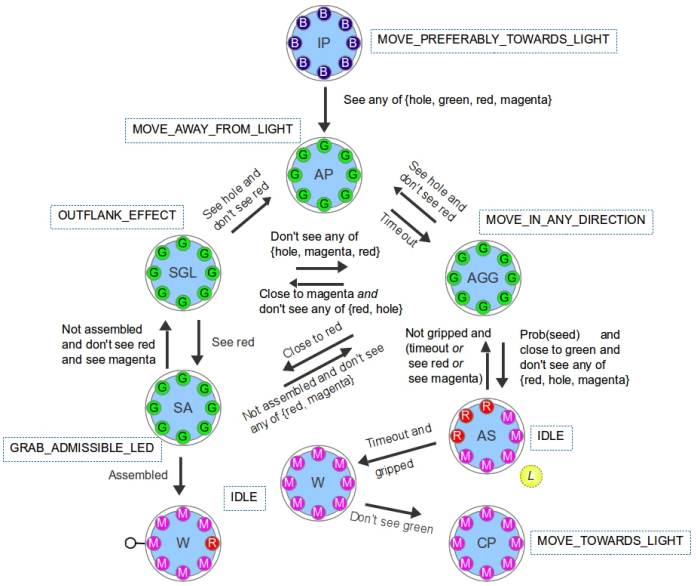


Figure 9: A variant of the basic self-assembly response strategy where s-bots assemble in lines.

now a further LED emission is considered (magenta): initially, an s-bot navigates independently towards the goal light source avoiding obstacles (state `Independent_Phototaxis`, controller `MOVE_PREFERABLY_TOWARDS_LIGHT`), and changes to state `Anti_Phototaxis` with controller `MOVE_AWAY_FROM_LIGHT` if it perceives a hole or a LED emission of color green, red, or magenta. After the expiration of a timeout, the s-bot changes its state to `Aggregate` with controller `MOVE_IN_ANY_DIRECTION` to search for grippable s-bots. Three cases can then arise.

In the first case a red LED emission is perceived, then the s-bot changes to state `Self_Assembly` with controller `GRAB_ADMISSIBLE_LED` to grip the perceived assembly. Once the gripping action completes, the s-bot changes its state to `Wait` with controller `IDLE`, turning all its LEDs to magenta, except for the one opposite to the gripper, which represents the new tail of the line, and that is set to red to allow for other s-bots to grab it.

In the second case a magenta LED emission is perceived, and neither a red LED nor a hole are perceived. In this case the state of the s-bot is changed to `Search_Grippable_LED`, with controller `OUTFLANK_EFFECT`. Intuitively, the s-bot has perceived an assembly (i.e. the magenta emission), however it has to search for the correct point of the assembly to grip it, i.e. a red LED emission denoting the tail of the assembly. This is exactly the behaviour offered by the controller `OUTFLANK_EFFECT`. Once a red LED emission is perceived, then, as in the previous case, the s-bot changes to state `Self_Assembly`. If instead a hole is perceived, then, in order to avoid to fall in it, the state of the s-bot is changed back to `Anti_Phototaxis`. Finally, if for some reason the assembly is

Listing 8: Managing of LEDs in the transition from `Aggregate` to `Assembly_Seed` of the line-forming cross hole strategy

```
1  E: setLEDs(oppositeDirections(dirsToLight(K2)),red)
2     setOtherLEDs(oppositeDirections(dirsToLight(K2)),magenta)
```

not perceived anymore, then the state of the s-bot is reverted to `Aggregate`.

In the third case, neither assemblies nor holes are perceived, but only single s-bots (green color emissions). As for the basic self-assembly response strategy, with probability `Prob(seed)` the s-bot becomes the seed of a new assembly (i.e. the head of the line) by changing its state in `Assembly_Seed`, with controller `IDLE`. However, differently from the basic self-assembly response strategy, by resorting to the color magenta the seed s-bot influences the shape of the future assembly allowing to be gripped only in the directions opposite to the ones of the light source. This is depicted in state `AS` of Fig. 9: the LEDs corresponding to the directions opposite to the target light source are changed to red, while the others to magenta. Without giving too many details, the LEDs are setted to red and magenta with the equation of `adapt` (similarly to the one given in Listing 5) corresponding to the transition from state `Aggregate` to state `Assembly_Seed` as schematized in Listing 8. Now, after the expiration of a timeout, if the s-bot has been gripped it changes to state `Wait` with controller `IDLE`, and sets all its LEDs to magenta. Next, if no green emissions are perceived it changes in state `Connected_Phototaxis` and collectively navigates (with its assembly) towards the light source (controller `MOVE_TOWARDS_LIGHT`). As done in the basic self-assembly response strategy, in the case in which no s-bot grips an s-bot in state `Assembly_Seed` before the expiration of a timeout, or if other assemblies are perceived (a red or magenta emission), then it means that the choice of becoming the seed of a new assembly was unfortunate, and hence the s-bot changes back to state `Aggregate`.

### *6.3. A Self-Assembly Strategy for Collective Healing Scenarios*

We discuss here a *collective healing* scenario, where groups of agents cooperate to repair malfunctioning ones. The strategy of Fig. 10 is part of the outcome of the work done by a group of three students during the *International AWARENESS summer school on Self-Awareness in Autonomic Computing Systems (AWASS 2012)* [2], under the supervision of the fifth author. It is worth noticing that none of the students were expert in robotics, self-assembly behaviours or swarm computing. This shows the easiness of use of our framework.

The scenario was inspired by one of the invited talks of the summer school [37], that described how an organism reacts to a tumor cell: it gets surrounded by many cells, building a sort of shield around it, aimed at healing it. Similarly, we thought of a scenario where there is a virus (i.e. a malfunctioning battery

[2] `http://www.aware-project.eu/awareness-training/awass-2012/`

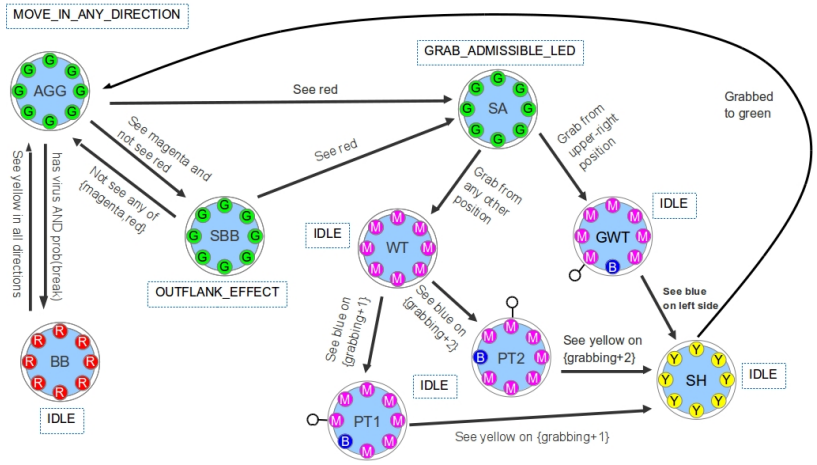


Figure 10: A self-assembly strategy for collective healing scenarios.

cell). With a certain probability, an s-bot with a malfunctioning battery may be unable to have enough energy to move. Then, in order to repair it, eight s-bots have to surround him and recharge it. As soon as the s-bot is charged, the s-bots disassemble. However, recharging the malfunctioning battery may damage other batteries (from the s-bots that performed the recharge operation).

For easiness of presentation, rather than discussing in detail the strategy, we exemplify it commenting a sample execution. Fig. 11 shows six relevant states of a discrete-event simulation of eleven s-bots executing the collective healing strategy of Fig. 10. Namely, top-left depicts a broken s-bot, i.e. the one in the middle with label `BB` (`BrokenBot`) with red LEDs and controller `IDLE`, while the other s-bots are randomly moving in the arena (controller `MOVE_IN_ANY_DIRECTION`) in state `Aggregate`. After a while, the s-bots notice the broken s-bot, change their status to `Self_Assembly` with controller `GRAB_ADMISSIBLE_LED` and start to assembly to it in order to completely surround it (Fig. 11, top-middle).

Like for the line-forming scenario, the color magenta is used to drive the generation of the assembly: as soon as an s-bot assembles a broken one, it changes its LEDs to magenta, in order to signal the presence of an ensemble and sets its controller to `IDLE`. When an s-bot in status `Aggregate` perceives a magenta LED emission, it knows that it is near to a broken s-bot, and hence changes its status to `Search_Broken_Bot` and search the broken s-bot thanks to the controller `OUTFLANK_EFFECT`.

Notice how, by design choice, an assembled s-bot is either in state `GWT` (`Generate_Wait_Token`) if it assembled the broken s-bot from the top-right position, or in state WT (`Wait_Token`). In fact, the idea is that we require to start healing an s-bot only when the ensemble has been completed, for example this may be reasonable if the broken s-bot can be healed only when all the eight s-bots help it, and each s-bot has to minimize the healing time, since it

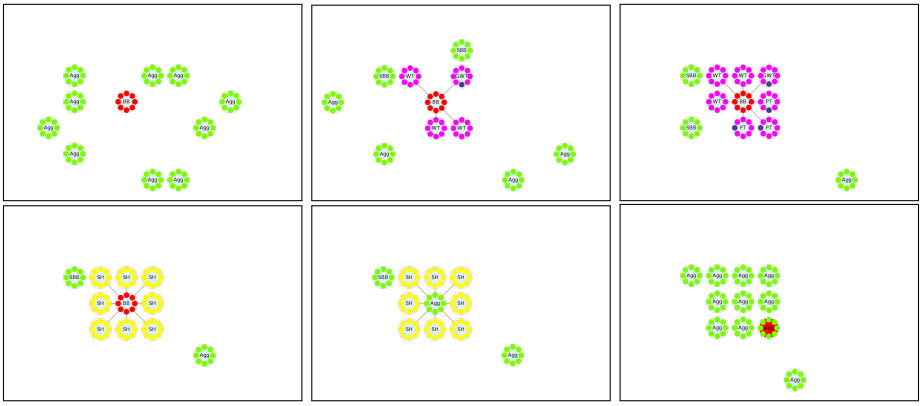

Figure 11: Six states of a simulation of the collective healing scenario.

may be highly energy consuming. For this reason we developed a simple token-based decentralized protocol to know when a broken s-bot has been completely surrounded: the s-bot in state `GWT` generates a token (a blue LED), which is then propagated clockwise (Fig. 11, top-right) by the s-bots in state `Wait_Token` which change to status `Propagate_Token` (either `PT1` or `PT2` depending on the direction from which the token is received). Clearly, once the s-bot in state `GWT` receives the token back, it knows that the ensemble is complete, and it can hence begin to irradiate with yellow light the broken s-bot (status `Self_Healing`). The irradiation is perceived by its neighbour, which consequently changes to status `Self_Healing` and turns on its yellow LEDs, and so on (Fig. 11, bottom-left).

As soon as the broken s-bot perceives a yellow light emission from all the eight directions, it knows it has been charged and changes back to status `Aggregate` (Fig. 11, bottom-middle). Finally, the s-bots disassemble from the healed s-bot. Eventually, the battery of some of these s-bots may be damaged, depicted as an s-bot with red body (Fig. 11, bottom-right). After a while, the malfunctioning battery will break the affected s-bot, and the strategy will start again.

The complete video of the described simulation is available in the MESSI website [18].

## 7. Methodology Validation: Analysis of Self-Assembly Strategies

In this section we describe the analysis activities that we performed exploiting our framework *MESSI* [18]. The analysis of self-assembly strategies were carried out in two phases: (§7.1) discrete event simulations; and (§7.2) statistical model checking. The rationale is the following.

Considering our experience in modelling the basic self-assembly response strategy, in the early development phases we were mainly concentrated on performing single simulations that were informally analyzed by observing the behavior of the assemblies in the automatically generated animations. This can be considered as a sort of debugging phase.

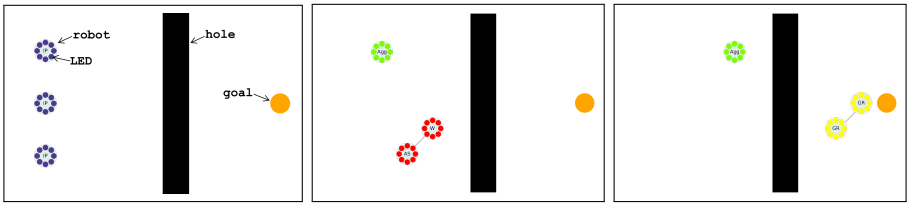


Figure 12: Three states of a simulation: initial (left), assembly (middle), final (right).

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 it suffers from the state-space explosion problem, and it is limited to small scenarios and to *qualitative* properties. To tackle larger scenarios, and to gain more insights into the probabilistic model by reasoning about *probabilities* and *quantities* rather than *possibilities*, we resorted to quantitative model checking techniques, and in particular to statistical model checking.

We now provide the details of these analysis phases, considering some of the self-assembly strategies discussed in this paper.

### *7.1. Simulations*

Simulations are performed thanks to the discrete-event simulator mentioned in §5 along the lines reported in [1, 6, 38]. Valuable help has been obtained implementing an exporter from Maude `Configuration` terms to DOT graphs [39] offering the automatic generation of images from states, and of animations from images: they have greatly facilitated the debugging of our code.

For example, Fig. 12 illustrates three states of a simulation where s-bots execute the *basic self-assembly response 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 light source (the orange circle to the right). After some steps, where the s-bots execute the self-assembly strategy, two s-bots finally get assembled (middle of Fig. 12). The assembled s-bots can then safely cross the hole and reach the goal (right of Fig. 12), while the third 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 strategy defined in* [5]?

Examining carefully the description of the strategy, we discovered that the two behaviors are indeed not explicitly disallowed in [5] and that they are originated by the two transitions (see Fig. 1) 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 s-bot willing to be grabbed). But an s-bot can change from state `Assembly_Seed` to state `Wait` even if no other s-bot is attached to it; in this case it can evolve to state `Connected_Phototaxis` and thus move towards the goal without trying to avoid the hole. Considering the other transition, 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. But now $j$ can pass from state `Assembly_Seed` to state `Aggregate`, where it will try to grab $i$. Interestingly, we noticed 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.

Performing simulations we noticed other bizarre behaviors not forbidden by the basic self-assembly response strategy, like for example when an s-bot remains stuck in state `Self_Assemble`. This is due to the fact that no recovery mechanism is provided for the cases in which an s-bot in state `Self_Assemble` fails to assemble with another s-bot, for example because the s-bot to be gripped is not perceived anymore (because it moved away).

*Are these behaviors actually possible for 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)?* Our experience makes it evident that the effectiveness of the self-assembly strategies described in [5] can also depend on the physical and mechanical properties of s-bots, and therefore these strategies might not be adequate in general for self-assembling settings where other constraints apply. Luckily, the three bizarre behaviors can be fixed easily by adding a new transition from state `Self_Assemble` to state `Aggregate`, and further conditions to the two mentioned transitions of the strategy. In particular, the transition from `Assembly_Seed` to `Wait` requires a further condition to ensure that the s-bot has been gripped, while the transition from `Assembly_Seed` to state `Aggregate` requires that the s-bot is not gripped. Finally, a new transition from state `Self_Assemble` to state `Aggregate` has to be added in order to solve the third bizarre behaviour. This variant of the strategy, actually with a few more arrangements, is depicted in Fig. 8.

### *7.2. Statistical Model Checking*

A qualitative analysis can prove that an assembly strategy can result in different degrees of success, from full success (e.g. all s-bots reach the goal, a line is always formed, or all the broken s-bots are fixed) to full failure (e.g. no s-bot reaches the goal). However, in the kind of scenarios considered, the really interesting questions are *how likely is each possible result?* Or *what is the expected number of a given measure?* For example the number of s-bots reaching the goal, or the number of composed lines.

A quantitative analysis is more appropriate in these cases. We resort to statistical model checking (see e.g. [40, 38, 6]), where probabilities and quantities are estimated. Such techniques do not yield the absolute confidence of qualitative or probabilistic 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. In fact, in statistical model checking we do not explore the whole state-space of a system, but we perform $n$ independent simulations, with $n$ large enough to statistically estimate quantitative properties.

We exploit PVeStA [6], a parallel statistical model checker. 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. As usual, in order to obtain meaningful estimations, we made our models totally probabilistic, meaning that we get rid of unquantified nondeterminism.

In the rest of this section we first describe QuaTEx, the language used to express quantitative properties evaluated by PVeStA (§7.2.1). Then we provide some information about the machine where we performed the experiments and the value of some parameters (§7.2.2), and finally we discuss some of the analysis results that we obtained, focusing on the hole-crossing scenario (§7.2.3).

*7.2.1. A sample QuaTEx expression*

Before detailing the analysis of some of the self-assembly strategies treated in this paper it is worth to introduce QuaTEx [1], the language that we used to express quantitative properties, and to discuss how PVeStA interprets it.

QuaTEx is a language to query quantitative aspects of probabilistic systems. Exactly like temporal logics allow to express temporal formulae, QuaTEx allows to express *quantitative temporal expressions*.

PVeStA statistically evaluates quantitative temporal expressions with respect to two parameters: $\alpha$ and $\delta$. 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 $[(1 - \delta)\overline{x}, (1 + \delta)\overline{x}]$.

As an example of quantitative temporal expression, we discuss now expression $Q_2$, analyzed later in §7.2.3 to estimate the expected number of s-bots reaching the goal in the hole-crossing scenario. In particular, we show how it has been defined, and how its value is actually computed for single simulations. We do not detail how PVeStA performs one-step system 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 state 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 s-bots, a state with all the s-bots 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 successful s-bots.

Listing 9: The QuaTEx expression to estimated the number of s-bots in goal

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

Then we defined the equations needed by 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` denoting the number of the predicate and the state on which to evaluate it.

Finally, the QuaTEx expression to estimate the expected number of s-bots reaching the goal is easily expressed as in Listing 9.

Informally, a QuaTEx expression consists of a list of definitions of recursive temporal operators, followed by a query of the expected value of a path expression obtained combining arithmetically 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. one of the states depicted in Fig. 13). 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 s-bots 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 one-step temporal operator.

To conclude, the evaluation of the QuaTEx expression consists of performing step-wise system simulations, and the result is the mean of the numbers of s-bots that reached the goal in the terminal states of each simulation.

*7.2.2. Assumptions, parameters and hardware specifications.*

In this section we provide the specifications of the machine where we performed the experiments, and we discuss some parameters and assumptions that we used.

All the experiments have been performed on a Linux machine with 64 GB of memory and 48 cores with clock speed 2.00 GHz. As previously mentioned, statistical model checkers (as it is the case of PVeStA) estimate quantitative properties of a probabilistic system modulo a confidence interval, specified by two parameters: $\alpha$ and $\delta$. Intuitively, there is a trade off between the estimation accuracy and the time required to compute it: the coarser is the confidence interval, the less accurate is the estimation, and faster is its estimation. In all

our experiments we fixed 0.05 for both $\alpha$ and $\delta$, meaning that with probability 95%, the actual value of a property estimated as $\overline{x}$ belongs to $[0.95 * \overline{x}, 1.05 * \overline{x}]$ (i.e. $\overline{x} \pm 5\%$). To reach such level of confidence PVeStA ran about one thousand simulations in average for an average total run time of about four hours.

We recall from §2 that s-bots have a diameter of 12 cm, and a maximal speed of 30 cm/s. As discussed in §5, we abstracted arenas to discrete grids (like chessboards). Each cell can contain at most one s-bot, and it is in fact dimensioned as the diameter of an s-bot (i.e. $12 \times 12$ cm$^2$). Our representation of s-bot's actions and perceptions is influenced by this abstraction: an s-bot can perform one-step movements to one of the eight cells surrounding it. We fixed 0.6 seconds as the time necessary to move to an adjacent cell. Similarly, an s-bot can grip other s-bots in one of the eight cells surrounding it. In particular, we decided to abstract from the time necessary to rotate the gripper, and we set to 2 seconds the time necessary to grip an s-bot. In the same way, s-bots perceptions are limited to the surrounding cells: an s-bot perceives only the LEDs emissions of the s-bots in the eight surrounding cells. The only global information perceived by an s-bot is the direction towards the light source.

*7.2.3. Analysis of self-assembly strategies for the hole-crossing scenario*

We performed some comparative analysis among the self-assembly strategies discussed in this paper for the hole-crossing scenario, namely the original *basic self-assembly response* of Fig. 6, our variant in Fig. 8 to deal with the bizarre behaviors discussed in §7.1, and the strategy of Fig. 9, where s-bots assemble in lines to cross the hole.

The aim of these experiments was twofold: on the one hand we wanted to demonstrate that the strategies of Fig. 8 and 9 don't generate the bizarre behaviors which, as discussed in §7.1, arise in (our implementation of) the basic self-assembly response strategy. In particular, we focused on the behaviour in which two s-bots grab each other. For this reason we defined the QuaTEx expression $Q_0$ : "*What is the probability that at least two s-bots grab each other in an execution?*".

On the other hand, we wanted to compare the performances of the three strategies in terms of success rate, in order to study how the absence of bizarre behaviours and the kind of shapes of the assemblies influence them. For this reason we defined the two QuaTEx expressions $Q_1$ : "*What is the probability that at least one s-bot reaches the goal?*", and $Q_2$ : "*What is the expected number of s-bots reaching the goal?*".

For each strategy we estimated the three properties for three configurations: 3 s-bots in a $11 \times 7$ grid, 6 s-bots in a $12 \times 8$ grid, and 9 s-bots in a $14 \times 10$ grid. The initial states of the three configurations are depicted in Fig.13. Other than the s-bots, in the initial states there were the goal (a source of yellow light) and a hole dividing the s-bots from the goal. Moreover, for the case with 3 s-bots we fixed 200 as maximal number of system steps, while we fixed 400 for the others. Finally, for all the experiments we fixed `Prob(seed)` to 0.7 (i.e. the probability for an s-bot in state `Aggregate` to become the seed of a new assembly), three

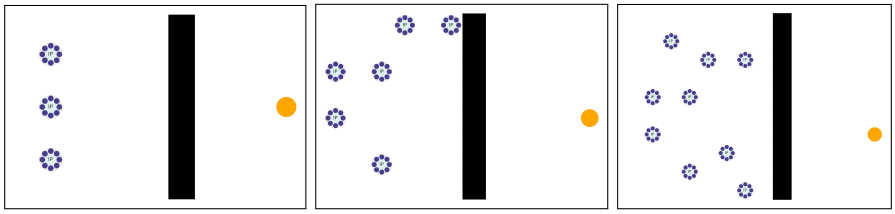

Figure 13: Three initial states for the hole-crossing scenario with 3 (left), 6 (middle) and 9 (right) s-bots.

| Property | Scenario | BSRS | BSRS$^+$ | BSRS$^+$LINE | BSRS$^+$(NON-VERTICAL) LINE |
|---|---|---|---|---|---|
| $Q_0$ | 3-BOTS | 0.64 | 0.00 | 0.00 | 0.00 |
| | 6-BOTS | 0.99 | 0.00 | 0.00 | 0.00 |
| | 9-BOTS | 0.99 | 0.00 | 0.00 | 0.00 |
| $Q_1$ | 3-BOTS | 0.57 | 0.61 | 0.53 | 0.87 |
| | 6-BOTS | 0.87 | 0.96 | 0.86 | 0.98 |
| | 9-BOTS | 0.96 | 0.96 | 0.96 | 1.00 |
| $Q_2$ | 3-BOTS | 1.21 | 1.15 | 1.07 | 1.83 |
| | 6-BOTS | 3.04 | 3.36 | 3.12 | 4.87 |
| | 9-BOTS | 4.91 | 5.34 | 4.37 | 6.78 |

Table 2: The result of the model checking procedure.

times the time to grip an s-bot as timeout for state `AssemblySeed`, and the time necessary to perform a movement as timeout for state `Anti_Phototaxis`.

Summarizing the results presented below, the double gripping behaviour is absent in our two strategies, while it arises in a significant number of executions for the basic self-assembly response strategy. Moreover, our variant of the basic self-assembly strategy (Fig. 8) exhibits the best success rate both in terms of probability that at least one s-bot reaches the goal, that in terms of expected number of succeeding s-bots, while the cross hole line-forming strategy of Fig. 9 does not have the expected performances.

Table 2 contains the results of our analysis. For each property ($Q_0$, $Q_1$ and $Q_2$) and for each initial configuration of Fig. 13, we show the estimation of a property for each of the three self-assembly strategies:"BSRS" stands for basic self-assembly response strategy (Fig. 6), "BSRS$^+$" stands for our variant of Fig. 8, and "BSRS$^+$LINE" is the strategy of Fig. 9. The last column "BSRS$^+$(NON-VERTICAL)LINE" refers to a variant of the strategy of Fig. 9 which we discuss later. We first focus on the columns BSRS, BSRS$^+$ and BSRS$^+$LINE.

Considering property $Q_0$, the analysis shows that the bizarre behavior does not arise in any configuration for our strategies, while it arises almost always in the original strategy.

Considering property $Q_1$, we see that if the s-bots execute our variant of the basic self-assembly strategy, then there is a higher probability that at least one of them reaches the goal for the 3- and 6-BOTS cases. For the 9-BOTS case, instead, the probabilities are almost the same for any strategy: clearly, as there are more s-bots, the probabilities that at least one reaches the goal are pushed

Listing 10: Managing of LEDs in the transition from `Aggregate` to `Assembly_Seed` to avoid vertical lines

```
1  E: setLEDs(oppositeDirections(dirsToLight(K2)) \ up \ down,red)
2     setOtherLEDs(oppositeDirections(dirsToLight(K2)) \ up \ down,magenta)
```

closer to one.

Finally, considering $Q_2$, the strategy BSRS$^+$ has in general the best performances, except than for the case with three robots in which it is slightly worse than the strategy BSRS.

Noteworthy, we expected BSRS$^+$LINE to be the best strategy, while it has the worst performances. We suspected that the strategy has low performances due to the fact that often the formed lines are parallel to the hole, so that all the s-bots of the line successively fall in it. Note that the light is perceived from one to three directions: for example, if the light is in a position on the right and below with respect to the s-bot, then it will be perceived from `right`, `down-right` and `down`. In order to confirm this hypothesis we modified the strategy to avoid lines parallel to the hole. In the considered configurations we only have vertical holes, hence it is sufficient to forbid the seed of the line (an s-bot in state `Assembly_Seed`) to turn red the LEDs in direction up and down. Intuitively, during the transition from `Aggregate` to `Assembly_Seed` the LEDs are not managed as schematized in Listing 8, but as in Listing 10.

The evaluation of the performances of this new strategy, depicted in the column BSRS$^+$(NON-VERTICAL)LINE of Table 2, confirms our hypothesis, as they are much better than the ones of BSRS$^+$LINE, and also of the other two cases. In particular, the probability that at least an s-bot reaches the goal is 0.87 for the scenario with three s-bots, and almost 1 for the scenarios with six and nine s-bots. The expected number of s-bots reaching the goal is instead almost 2, 5 and 7, respectively, for the scenario with three, six and nine s-bots.

As previously stated, our aim is not that of designing new self-assembly strategies, but rather that of showing how these can be defined and evaluated following our approach. Interestingly, what we found is that the extra time spent in forming lines pays back if we manage to obtain lines non parallel to the hole.

## 8. Related Work

We discuss in this section some works that have influenced and inspired us, together with further related works. The discussion serves also as an argument for supporting the suitability of our approach, by covering most of its distinguishing features. In particular, we consider approaches built around the notion of Russian Dolls and other computational models of adaptive systems based on hierarchical structures or on computational reflection, and verification based on statistical model checking, with a focus on the analysis of probabilistic rewrite theories. Summarizing, 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.

*Russian Dolls and other hierarchical models of adaptive systems.* Our work is inspired by early approaches to coordination and adaptation based on distributed object reflection [17, 3] and research efforts to apply formal analysis onto such kind of systems (see the discussions below on this topic), with a particular focus on adaptive systems.

Among those, PAGODA (Policy And GOal based Distributed Autonomy) [28] is the closest in spirit and shape. PAGODA is a modular architecture for specifying and prototyping systems of autonomous cooperating agents, and it builds upon the Russian Dolls model. A difference with respect to our work is that in PAGODA adaptation (called *coordination*) is mainly realized by intercepting and manipulating messages rather than on the meta-programming mechanisms based on reflection adopted in our approach. In addition, contrary to our approach, PAGODA imposes a concrete two layered global architecture (system and nodes). Also the concrete node architecture is different from ours. In particular, nodes are composed of a knowledge base, a reasoner, a monitor, a learner, and a hardware abstraction layer. Our generic architecture does not impose such a structure which could be certainly adopted as a concrete instance.

Another approach similar in spirit to PAGODA is PobSAM (Policy-based Self-Adaptive Model) [23], a formal framework for modelling and analyzing self-adaptive systems which relies on policies as high-level mechanism to realize adaptive behaviors. PobSAM combines the actor model of coordination [41] with process algebra machinery and shares our white-box spirit of separating application and adaptation concerns. As in PAGODA the global architecture of the system has two layers only, composed of *managed actors*, which implement the functional behavior of the system, and *autonomic manager (meta-)actors*, which control the behavior of managed actors by enforcing policies (rules that determine under which condition actors must or must not do a certain action). The configuration of managers is determined by their sets of policies which can vary (i.e. adapt) dynamically. The currently active set of policies represents the control data in this approach. Adaptation is indeed the switch between active policies. So, besides the differences in the architectural aspects (see the above discussion of PAGODA), PobSAM and our approach share the idea of having rules as a high-level object of adaptation (i.e. control data in our terminology).

Close to our work is also the *composite actor model* of [42] which combines the Russian Dolls and actor models to specify hierarchically composed entities in a disciplined way by imposing constraints on the contents of some attributes of actors, and on their interaction.

Besides Russian Dolls, many modelling frameworks for adaptive systems are based on hierarchical structures. In particular, as in our case study, very often an adaptive system is seen as a system composed by a base-level system with a fixed collection of possible behaviors (or behavioral models), and adaptation consists of passing from one behavior to another. We mention among others the two-layered Petri nets of [13], *Adaptive Featured Transition Systems* (A-

FTS) [43], *S[B] systems* [44], approaches based on Mode automata [45, 46, 47], the *transitional adaptation lattices* of [48], the *Synchronous Adaptive Systems* (SAS) of MARS [49, 50], *Adaptable Transition Systems* (ATSs) [51], *n-plex adaptive programs* [52] and the state space *zones* of [53].

The most relevant difference with respect to our approach is that such models are typically limited to two layers and, in some cases, the system is seen as monolithic rather than component-based. Our approach does not impose a two-layered asymmetric structure: layers can be composed at will, possibly forming towers of adaptation [4] in the spirit of the MAPE-K reference architecture. In addition, each component of an adaptive system (be it a manager, a managed component, or both) is represented with the same mathematical object allowing us to reuse the same techniques at each layer.

*Reflection-based adaptation.* Computational reflection is widely accepted as one of the key instruments to model and build self-adaptive systems [27]. Indeed, computational paradigms equipped with reflective, meta-level or higher-order features, allow one to represent programs as first-class citizens. In these cases adaptivity emerges, according to our conceptual framework, if the program in execution or part of it is represented in the control data of the system, and it is modified during execution causing changes of behaviour. Prominent examples of such formalisms are process calculi with higher-order or meta-level aspects (e.g. HO $\pi$-calculus [54], MetaKlaim [55]), higher-order variants of Petri nets and Graph Grammars, the logical reflection of rewriting logic, Logic Programming, and programming languages like LISP, Java, C#, Perl and several others. Systems implemented in these paradigms can realize adaptation within themselves (self-adaptivity), but in general the program under execution can be modified also by a different entity, like an autonomic manager implemented in a different language, or in the same language but running in a separate thread. Of course, computational reflection assumes different forms and, despite of being a very convenient mechanism, it is not strictly necessary: any programming language can be used to build a self-adaptive system (see e.g. the discussion of [56]).

As an illustrative example where reflection plays a major role in the development of adaptive systems we mention the FOrmal Reference Model for Self-adaptation (FORMS) [29, 11, 25]. Reflection in FORMS implies the presence, besides of base-level components and computations, of meta-level subsystems and meta-computations that act on a meta-model. Meta-computations can inspect and modify the meta-model that is causally connected to the base-level system, so that changes in one are reflected in the other. This implies some major differences with our approach which lacks of an explicit notion of meta-model in the FORMS sense, and whose hierarchical architecture would forbid a base layer to modify directly the upper layer (a hot-linking mechanism would be needed to keep synchronized both layers).

The authors argue that most methodologies and frameworks proposed for the design and development of self-adaptive systems rely on some form of reflection, even if this is not always made explicit. FORMS provides basic modelling

primitives and relationships among them, suitable for the design of self-adaptive systems. Such primitives allow one to make explicit the presence of reflective (meta-level) subsystems, computations and models. The FORMS modelling primitives can be instantiated and composed in a variety of ways. For example, the authors of [11] provide one example that conforms to the MAPE-K reference model and another one that follows an application-specific design. In general, in any such reflective system the control data can be identified at the boundaries between the meta-level and the base-level components.

Another illustrative example that is worth mentioning belongs to the field of process algebras and consists of studying the suitability of the tuple-space coordination model of KLAIM [57] as a convenient mechanism for modelling self-adaptive systems [58]. The authors describe how to adopt in KLAIM three paradigms for adaptation: two that focus on the language-level, namely, context-oriented programming [59] and dynamic aspect-oriented programming [24], and one that focuses on the architectural-level (i.e. MAPE-K). The main idea in all the cases is to rely on the use of *process tuples*, that is tuples (the equivalent of messages in the tuple-space paradigm) that denote entire processes. These process tuples can be sent by manager components (*locations* in KLAIM) to managed components, which can then install them via the *eval* primitive of KLAIM. In few words, adaptation is achieved by means of code mobility and code injection. This is a point in common with our approach even if we don't consider a particular way of encapsulating code (behavior) such as aspects or variations (respectively used in the aspect- and context-oriented paradigms). Interestingly enough, KLAIM and its toolset are used in [60] in order to specify and analyse collective robotic systems. In particular, and differently from our approach, they focus on collective transport problems. As in our case, the analysis is based on statistical model checking. Letting apart the differences, this work shows the raising interest of the robotics community on the use of high-level, abstract formal modelling and verification techniques for rapid-prototyping purposes (see also the use of the probabilistic model checker PRISM in the methodology of [61]).

*Modelling and analysing probabilistic systems with Maude.* Moving to the concrete setting of Maude, several works (e.g. [42, 62, 63, 64, 65, 66]) can be found in the literature where probabilistic systems specified by probabilistic rewrite theories in PMaude [1] are analyzed using probabilistic analysis methods like statistical model checking. Examples range from hybrid systems to probabilistic load balancing policies, to service stability protocols. Among others, we highlight the already mentioned composite actors of [42] and the probabilistic strategy language of [66]. The authors of [66] face a typical problem that arises when modelling and analyzing probabilistic systems: one needs to ensure that the underlying system is free of nondeterminism in order to make it amenable for statistical model checking. Typical solutions are based on imposing certain conditions and rule formats, or by resorting to ad-hoc solutions like minimizing the probability of concurrent events. For example, in the above discussed composite actor models [42] this problem is solved by a transformation which,

provided a system specification satisfying some well-formedness requirements, exploits the idea of using a (top-level) scheduler of messages, to impose an ordering of consumption of messages, as proposed in [1, 63]. To cope with the hierarchical structure (the scheduler and the messages to be scheduled do not necessarily reside in the same level) messages have to be moved across layers when scheduled and descheduled. Our approach is similar, as we also resort to a top-level scheduler of messages. However, rather than moving messages across layers, we let them be consumed by the cells containing robots, which reside at the same level of the scheduler. Then messages are transformed in tokens for the outermost layer of a robot, and are used by the outer layers to perform one-step executions of the managed components (i.e. inner layers). Instead, the probabilistic strategy language of [66] offers an elegant and flexible solution that allows to keep the nondeterminism in the system, which is solved at the level of the strategy. Roughly, the idea is to assign probabilities to nondeterministic transitions by assigning weights to rule matches (based on the rule name, the context of application, and the substitution) that are later normalized into values in the interval $[0, 1]$.

## 9. Conclusion

*Contribution.* The main contributions of our paper are: (i) a description (§4–§5) of how to realize in Maude our recently proposed approach to adaptive systems [4] in a simple and natural way; and (ii) the validation of our methodology and of *MESSI*, its instantiation in Maude, for the early prototyping of adaptive systems (§6) and for their analysis (§7) exploiting the Maude toolset.

The distinguishing features of our approach are: (i) hierarchical structure to modularize the design; (ii) high-level, rule-based self-adaptation based on computational reflection; and (iii) quantitative analysis based on statistical model checking. The suitability of such features has been discussed throughout the paper and is witnessed by prominent examples from the literature on self-adaptive systems (as discussed in §8).

*Originality.* We have discussed our sources of inspiration in §8, the main ones being early approaches to coordination and adaptation based on distributed object reflection [3, 17, 28]. 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.

*Validation.* The case study of self-assembly strategies for robot swarms [5] has contributed to assess our approach and framework. Overall, the many implemented strategies (§6) witness the wide applicability and accessibility of our approach, thanks to the modularity provided by our hierarchical structuring. Moreover, the conducted experimentations demonstrate that it is well-suited for prototyping self-assembly systems in early development stages, and that the capability of performing and visualizing simulations can be useful to discover

and resolve small ambiguities and bugs of self-assembly strategies. Furthermore, statistical model checking allows us to estimate quantitative properties of self-assembly strategies, enabling to compare them at prototypal development stage.

*Future work.* We plan to further develop our work by considering other case studies and more realistic abstractions. However, the key challenging question we want to tackle is: *can we exploit the proposed architecture to facilitate also the analysis of adaptation strategies other than just their design?* 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 plan to develop a compositional reasoning technique that exploits the hierarchical structure of the layered architecture. For this purpose we plan to draw our inspiration from some works that focus on the distinction of particular properties of adaptive systems (see e.g. the *local*, *global* and *adaptation* properties of [52, 67, 68] and the *weak* and *strong* adaptability of [44]), variants of temporal logics tailored for expressing properties of adaptive systems (e.g. AdaCTL [43], $m$LTL [47], and A-LTL [69]), and modular reasoning techniques (e.g. based on *transitional adaptation lattices* [48] or *n-plex adaptive programs* [52]).

**Acknowledgements.** We are grateful to the organizers of the *AWASS 2012 summer school*[3] for the opportunity to mentor a case study based on the experience of this paper, and to the mentored students Herwig Guggi, Ilias Gerostathopoulos and Rosario Surace for their work.

## References

[1] G. A. Agha, J. Meseguer, K. Sen, PMaude: Rewrite-based specification language for probabilistic object systems, in: A. Cerone, H. Wiklicky (Eds.), QAPL 2005, Vol. 153(2) of ENTCS, Elsevier, 2006, pp. 213–239.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott, All About Maude, Vol. 4350 of LNCS, Springer, 2007.

[3] C. L. Talcott, Coordination models based on a formal model of distributed object reflection, in: L. Brim, I. Linden (Eds.), MTCoord 2005, Vol. 150(1) of ENTCS, Elsevier, 2006, pp. 143–157.

[4] R. Bruni, A. Corradini, A. Lluch Lafuente, F. Gadducci, A. Vandin, A conceptual framework for adaptation, in: J. de Lara, A. Zisman (Eds.), FASE 2012, Vol. 7212 of LNCS, Springer, 2012, pp. 240–254.

[5] R. O'Grady, R. Groß, A. L. Christensen, M. Dorigo, Self-assembly strategies in a group of autonomous mobile robots, Autonomous Robots 28 (4) (2010) 439–455.

[3] `http://www.aware-project.eu/awareness-training/awass-2012/`

[6] M. AlTurki, J. Meseguer, PVeStA: A parallel statistical model checking and quantitative analysis tool, in: Corradini et al. [70], pp. 386–392.

[7] IBM Corporation, An Architectural Blueprint for Autonomic Computing (2006).

[8] M. Salehie, L. Tahvildari, Self-adaptive software: Landscape and research challenges, ACM Transactions on Autonomous and Adaptive Systems 4 (2) (2009) 1–42.

[9] P. Horn, Autonomic Computing: IBM's perspective on the State of Information Technology (2001).

[10] J. O. Kephart, D. M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50.

[11] D. Weyns, S. Malek, J. Andersson, FORMS: a formal reference model for self-adaptation, in: R. Figueiredo, E. Kiciman (Eds.), ICAC 2010, ACM, 2010, pp. 205–214.

[12] G. Cabri, M. Puviani, F. Zambonelli, Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles, in: W. W. Smari, G. C. Fox (Eds.), CTS 2011, IEEE Computer Society, 2011, pp. 508–515.

[13] J. Zhang, B. H. C. Cheng, Model-based development of dynamically adaptive software, in: Osterweil et al. [71], pp. 371–380.

[14] G. Karsai, J. Sztipanovits, A model-based approach to self-adaptive software, Intelligent Systems and their Applications 14 (3) (1999) 46–53.

[15] D. Pavlovic, Towards semantics of self-adaptive software, in: Robertson et al. [72], pp. 65–74.

[16] M. Broy, C. Leuxner, W. Sitou, B. Spanfelner, S. Winter, Formalizing the notion of adaptive system behavior, in: Shin and Ossowski [73], pp. 1029–1033.

[17] J. Meseguer, C. Talcott, Semantic models for distributed object reflection, in: B. Magnusson (Ed.), ECOOP 2002, Vol. 2374 of LNCS, Springer, 2002, pp. 1–36.

[18] Maude Ensemble Strategies Simulator and Inquirer (MESSI) (2012).
URL `http://sysma.lab.imtlucca.it/tools/ensembles/`

[19] R. Bruni, A. Corradini, F. Gadducci, A. Lluch-Lafuente, A. Vandin, Modelling and Analyzing Adaptive Self-assembly Strategies with Maude, in: Durán [74], pp. 118–138.

[20] F. Mondada, G. C. Pettinaro, A. Guignard, I. W. Kwee, D. Floreano, J.-L. Deneubourg, S. Nolfi, L. M. Gambardella, M. Dorigo, Swarm-bot: A new distributed robotic concept, Autonomous Robots 17 (2-3) (2004) 193–221.

[21] I. Harvey, E. A. D. Paolo, R. Wood, M. Quinn, E. Tuci, Evolutionary robotics: A new scientific tool for studying cognition, Artificial Life 11 (1-2) (2005) 79–98.

[22] G. Salvaneschi, C. Ghezzi, M. Pradella, Context-oriented programming: A programming paradigm for autonomic systems, CoRR abs/1105.0069.

[23] N. Khakpour, S. Jalili, C. Talcott, M. Sirjani, M. Mousavi, Formal modeling of evolving self-adaptive systems, Science of Computer Programming 78 (1) (2012) 3 – 26.

[24] P. Greenwood, L. Blair., Using dynamic aspect-oriented programming to implement an autonomic system, in: Proceedings of the 2004 Dynamic Aspects Workshop (DAW04), RIACS, 2004, pp. 76–88.

[25] D. Weyns, S. Malek, J. Andersson, FORMS: Unifying reference model for formal specification of distributed self-adaptive systems, ACM Transactions on Autonomous and Adaptive Systems 7 (1) (2012) 8.

[26] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theoretical Computer Science 96 (1) (1992) 73–155.

[27] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, B. H. C. Cheng, Composing adaptive software, IEEE Computer 37 (7) (2004) 56–64.

[28] C. L. Talcott, Policy-based coordination in PAGODA: A case study, in: G. Boella, M. Dastani, A. Omicini, L. W. van der Torre, I. Cerna, I. Linden (Eds.), CoOrg 2006 & MTCoord 2006, Vol. 181 of ENTCS, Elsevier, 2007, pp. 97–112.

[29] J. Andersson, R. de Lemos, S. Malek, D. Weyns, Reflecting on self-adaptive software systems, in: SEAMS 2009 [75], pp. 38–47.

[30] R. O'Grady, A. L. Christensen, C. Pinciroli, M. Dorigo, Robots autonomously self-assemble into dedicated morphologies to solve different tasks, in: W. van der Hoek, G. A. Kaminka, Y. Lespérance, M. Luck, S. Sen (Eds.), AAMAS 2010, IFAAMAS, 2010, pp. 1517–1518.

[31] Y. Jin, Y. Meng, Morphogenetic robotics: An emerging new field in developmental robotics, IEEE Transactions on Systems, Man, and Cybernetics, Part C 41 (2) (2011) 145–160.

[32] R. O'Grady, A. L. Christensen, M. Dorigo, Swarmorph: Multirobot morphogenesis using directional self-assembly, IEEE Transactions on Robotics 25 (3) (2009) 738–743.

[33] M. Mamei, M. Vasirani, F. Zambonelli, Experiments of morphogenesis in swarms of simple mobile robots, Applied Artificial Intelligence 18 (9-10) (2004) 903–919.

[34] S. C. Goldstein, J. D. Campbell, T. C. Mowry, Programmable matter, IEEE Computer 38 (6) (2005) 99–101.

[35] K. Gilpin, D. Rus, What's in the bag: A distributed approach to 3d shape duplication with modular robots, in: Robotics: Science and Systems, 2012.

[36] M. Yim, W.-M. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, G. Chirikjian, Modular self-reconfigurable robot systems [grand challenges of robotics], Robotics Automation Magazine, IEEE 14 (1) (2007) 43 –52.

[37] J. Timmis, Immunity in Self-Aware Systems, Invited talk at AWASS2012. URL `http://www.aware-project.eu/documents/awass-2012/immunity-in-self-aware-systems-jon-timmis.pdf`

[38] K. Sen, M. Viswanathan, G. A. Agha, Vesta: A statistical model-checker and analyzer for probabilistic systems, in: C. Baier, G. Chiola, E. Smirni (Eds.), QEST 2005, IEEE Computer Society, 2005, pp. 251–252.

[39] GraphViz – Graph Visualization Software.
URL `http://www.graphviz.org/`

[40] K. Sen, M. Viswanathan, G. Agha, On statistical model checking of stochastic systems, in: K. Etessami, S. K. Rajamani (Eds.), CAV 2005, Vol. 3576 of LNCS, Springer, 2005, pp. 266–280.

[41] G. Agha, Actors: a model of concurrent computation in distributed systems, MIT Press, Cambridge, MA, USA, 1986.

[42] J. M. Jonas Eckhardt, Tobias Mühlbauer, M. Wirsing, Statistical model-checking for composite actor systems, in: Preproceedings of the 21st International Workshop on Algebraic Development Techniques (WADT 2012), 2012.

[43] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, Model checking adaptive software with featured transition systems, in: Proceedings of the 4th Workshop on Games for Design, Verification and Synthesis, 2012.

[44] E. Merelli, N. Paoletti, L. Tesei, A multi-level model for self-adaptive systems, in: Proceedings of the 11th International Workshop on Foundations of Coordination Languages and Self Adaptation (FOCLASA 2012), 2012.

[45] F. Maraninchi, Y. Rémond, Mode-automata: About modes and states for reactive systems, in: C. Hankin (Ed.), ESOP, Vol. 1381 of Lecture Notes in Computer Science, Springer, 1998, pp. 185–199.

[46] F. Maraninchi, Y. Rémond, Mode-automata: a new domain-specific construct for the development of safe critical systems, Sci. Comput. Program. 46 (3) (2003) 219–254.

[47] Y. Zhao, D. Ma, J. Li, Z. Li, Model checking of adaptive programs with mode-extended linear temporal logic, Engineering of Autonomic and Autonomous Systems, IEEE International Workshop on 0 (2011) 40–48.

[48] K. N. Biyani, S. S. Kulkarni, Assurance of dynamic adaptation in distributed systems, J. Parallel Distrib. Comput. 68 (8) (2008) 1097–1112.

[49] R. Adler, I. Schaefer, T. Schüle, E. Vecchié, From model-based design to formal verification of adaptive embedded systems, in: M. Butler, M. G. Hinchey, M. M. Larrondo-Petrie (Eds.), ICFEM, Vol. 4789 of Lecture Notes in Computer Science, Springer, 2007, pp. 76–95.

[50] I. Schaefer, A. Poetzsch-Heffter, Using abstraction in modular verification of synchronous adaptive systems, in: S. Autexier, S. Merz, L. W. N. van der Torre, R. Wilhelm, P. Wolper (Eds.), Trustworthy Software, Vol. 3 of OASICS, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[51] R. Bruni, A. Corradini, F. Gadducci, A. L. Lafuente, A. Vandin, Adaptable transition systems, in: Preproceedings of the 21st International Workshop on Algebraic Development Techniques (WADT 2012), 2012.

[52] J. Zhang, H. Goldsby, B. H. C. Cheng, Modular verification of dynamically adaptive systems, in: K. J. Sullivan, A. Moreira, C. Schwanninger, J. Gray (Eds.), AOSD, ACM, 2009, pp. 161–172.

[53] U. Iftikhar, D. Weyns, A case study on formal verification of self-adaptive behaviors in a decentralized system, in: Proceedings of the 11th International Workshop on Foundations of Coordination Languages and Self Adaptation (FOCLASA 2012), 2012.

[54] D. Sangiorgi, Expressing mobility in process algebras: First-order and higher-order paradigms, Ph.D. thesis, University of Edinburgh (1992).

[55] G. L. Ferrari, E. Moggi, R. Pugliese, Metaklaim: a type safe multi-stage language for global computing, Mathematical Structures in Computer Science 14 (3) (2004) 367–395.

[56] C. Ghezzi, M. Pradella, G. Salvaneschi, An evaluation of the adaptation capabilities in programming languages, in: H. Giese, B. H. Cheng (Eds.), SEAMS 2011, ACM, 2011, pp. 50–59.

[57] R. De Nicola, G. L. Ferrari, R. Pugliese, Klaim: A kernel language for agents interaction and mobility, IEEE Trans. Software Eng. 24 (5) (1998) 315–330.

[58] E. Gjondrekaj, M. Loreti, R. Pugliese, F. Tiezzi, Modeling adaptation with a tuple-based coordination language, in: S. Ossowski, P. Lecca (Eds.), SAC, ACM, 2012, pp. 1522–1527.

[59] R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-oriented programming, Journal of Object Technology 7 (3) (2008) 125–151.

[60] E. Gjondrekaj, M. Loreti, R. Pugliese, F. Tiezzi, C. Pinciroli, M. Brambilla, M. Birattari, M. Dorigo, Towards a formal verification methodology for collective robotic systems, in: T. Aoki, K. Taguchi (Eds.), ICFEM, Vol. 7635 of Lecture Notes in Computer Science, Springer, 2012, pp. 54–70.

[61] M. Brambilla, C. Pinciroli, M. Birattari, M. Dorigo, Property-driven design for swarm robotics, in: W. van der Hoek, L. Padgham, V. Conitzer, M. Winikoff (Eds.), AAMAS, IFAAMAS, 2012, pp. 139–146.

[62] J. Meseguer, R. Sharykin, Specification and analysis of distributed object-based stochastic hybrid systems, in: J. Hespanha, A. Tiwari (Eds.), HSCC 2006, Vol. 3927 of LNCS, Springer, 2006, pp. 460–475.

[63] M. AlTurki, J. Meseguer, C. A. Gunter, Probabilistic Modeling and Analysis of DoS Protection for the ASV Protocol, Electr. Notes Theor. Comput. Sci. 234 (2009) 3–18.

[64] M. Wirsing, J. Eckhardt, T. Mühlbauer, J. Meseguer, Design and Analysis of Cloud-Based Architectures with KLAIM and Maude, in: Durán [74], pp. 54–82.

[65] J. Eckhardt, T. Mühlbauer, M. AlTurki, J. Meseguer, M. Wirsing, Stable availability under denial of service attacks through formal patterns, in: J. de Lara, A. Zisman (Eds.), FASE, Vol. 7212 of Lecture Notes in Computer Science, Springer, 2012, pp. 78–93.

[66] L. Bentea, P. C. Ölveczky, A probabilistic strategy language for probabilistic rewrite theories, in: Preproceedings of the 21st International Workshop on Algebraic Development Techniques (WADT 2012), 2012.

[67] Y. Zhao, D. Ma, J. Li, Z. Li, Model checking of adaptive programs with mode-extended linear temporal logic, in: Engineering of Autonomic and Autonomous Systems (EASe), 2011 8th IEEE International Conference and Workshops on, IEEE Computer Society, 2011, pp. 40–48.

[68] S. S. Kulkarni, K. N. Biyani, Correctness of component-based adaptation, in: I. Crnkovic, J. A. Stafford, H. W. Schmidt, K. C. Wallnau (Eds.), CBSE, Vol. 3054 of Lecture Notes in Computer Science, Springer, 2004, pp. 48–58.

[69] J. Zhang, B. H. C. Cheng, Using temporal logic to specify adaptive program semantics, Journal of Systems and Software 79 (10) (2006) 1361–1369.

[70] A. Corradini, B. Klin, C. Cîrstea (Eds.), Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings, Vol. 6859 of LNCS, Springer, 2011.

[71] L. J. Osterweil, H. D. Rombach, M. L. Soffa (Eds.), 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006, ACM, 2006.

[72] P. Robertson, H. E. Shrobe, R. Laddaga (Eds.), Self-Adaptive Software, First International Workshop, IWSAS 2000, Oxford, UK, April 17-19, 2000, Revised Papers, Vol. 1936 of LNCS, Springer, 2001.

[73] S. Y. Shin, S. Ossowski (Eds.), Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009, ACM, 2009.

[74] F. Durán (Ed.), Rewriting Logic and Its Applications - 9th International Workshop, WRLA 2012, Held as a Satellite Event of ETAPS, Tallinn, Estonia, March 24-25, 2012, Revised Selected Papers, Vol. 7571 of Lecture Notes in Computer Science, Springer, 2012.

[75] 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2009, Vancouver, BC, Canada, May 18-19, 2009, IEEE, 2009.