Modelling and Analyzing Adaptive Self-Assembly Strategies with Maude

Roberto Bruni, Andrea Corradini, Fabio Gadducci

Dipartimento di Informatica, Università di Pisa, Italy
{bruni,andrea,gadducci}@di.unipi.it

Alberto Lluch Lafuente

IMT Institute for Advanced Studies Lucca, Italy
{alberto.lluch}@imtlucca.it

Andrea Vandin

Department of Electronic and Computer Science, University of Southampton, UK,
IMT Institute for Advanced Studies Lucca, Italy
{a.vandin}@soton.ac.uk

Abstract

Building adaptive systems with predictable emergent behavior is a difficult 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 and analysis techniques. Our white-box conceptual approach to adaptive systems based on the notion of *control data* promotes a clear distinction between the application and the adaptation logic. In this paper we propose a concrete instance of our approach based on (i) a neat identification of control data; (ii) a hierarchical architecture that provides the basic structure to separate the adaptation and application logics; (iii) computational reflection as the main mechanism to realize the adaptation logic; (iv) probabilistic rule-based specifications and quantitative verification techniques to specify and analyze the adaptation logic. We show that our solution can be naturally realized in Maude, a Rewriting Logic based framework, and illustrate our approach by specifying, validating and analysing a prominent example of adaptive systems: robot swarms equipped with self-assembly strategies.

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

Research partly supported by the European IP 257414 ASCENS, the European STReP 600708 QUANTICOL, and the Italian PRIN 2010LHT4KM CINA.

1. Introduction

One of todays' grand challenges in Computer Science is to engineer autonomic systems. First, autonomic components run in unpredictable environments, and thus must be engineered by relying on the smallest possible amount of assumptions, i.e. as *adaptive* components. Second, there is no crisp distinction between failure and success, because the non-deterministic behaviour of such systems prevents an absolute winning strategy to exist. Third, efforts spent in the accurate analysis of handcrafted adaptive components are unlikely to pay back, because the results are scarcely reusable when components are often updated or extended with new features. It is no surprise that no general formal framework for adaptive systems exists that is widely accepted. Instead, several adaptation models and guidelines occur in the literature that offer ad-hoc solutions, often tailored to specific application domains or programming languages.

In [1] we have proposed a conceptual framework for adaptation that provides simple guidelines for a clean structuring of self-adaptive systems. Here we instantiate our conceptual approach to a concrete architecture whose main characteristics are (i) a white-box approach to adaptation based on the notion of control data [1] that clearly identifies the adaptation logic; (ii) a hierarchical architecture inspired by the MAPE-K reference model [2], that provides the basic structure of the adaptation logic; (iii) computational reflection as the main mechanism to realize the adaptation logic [3]; (iv) probabilistic rule-based specifications and quantitative verification techniques to specify and analyze the adaptation logic.

We show that our proposal can be naturally realized in Maude [4], a framework based on Rewriting Logic [5], since (i) Maude's algebraic approach facilitates the formalization of control data; (ii) hierarchical architectures such as the Reflective Russian Dolls model [6] have been promoted and shown to be suitable to specify adaptive systems in Maude; (iii) Maude efficiently supports computational reflection; and (iv) it is a rule-based language for which probabilistic extensions and analysis techniques are available. Of course, our approach can be realized in any other framework providing the necessary support for (i)–(iv).

More precisely, we describe a methodology to instantiate our generic architecture for prototyping well-engineered self-adaptive components in specific systems or scenarios. Our main case study consists of modelling, debugging, analyzing and comparing self-assembly strategies of robots cooperating for different purposes, including *morphogenesis* (where robots assemble to form predefined shapes) and *obstacle avoidance* (e.g., hole-crossing while navigating towards a light source [7]). This is achieved by exploiting MESSI (*Maude Ensemble Strategies Simulator and Inquirer*), an integrated toolset supporting: (1) the modeling of robotic self-assembly strategies with PMaude [8] (a probabilistic extension of Maude), (2) debugging them via animated simulations, and (3) analysing and comparing their performances using the parallel statistical model checker PVeStA [9]. For a brief and informal presentation of MESSI, including animations and the source code, we refer to the MESSI website [10].

The work reported in this paper is primarily addressed to strategy designers

for swarm-robotics and to Maude researchers. For the latter, we provide a principled development methodology that is based on a state-of-the-art Maude toolset and that can be reused and extended to support the specification and analysis of adaptive systems. For the former audience, the possibility to rapidly develop self-adaptive systems and to quantitatively analyze them at the early stages of development is very important. Indeed, for example, the robots used in the experiments reported in [7] require specialized programming skills and their testing in real world 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 [7] only 6 out of the 25 existing robots were used) due to maintenance costs. Also, their hardware and software are frequently updated, making it harder to build and to maintain sophisticated simulators that can take as input exactly the same code to be run on the robots. Even when this has been attempted, the real world tests can differ substantially from the simulated runs. Thus, early analysis on prototypes, even if performed on a quite abstract representation of the real system, can at least speed-up testing and debugging, and 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 a running example along the paper. In §3 we start by overviewing the inspiring principles of our generic multi-layered approach for adaptive systems (§3.1), then we define our proposal in detail (§3.2), and finally we describe a concrete instance of the generic architecture tailored to the robotic case study (§3.3). This architecture is instantiated to an implementation of the strategies of the case study in §4, based on MESSI, a Maude tool developed to that purpose [10]. The general guidelines and principles used in Maude for modelling self-adaptive systems (including logical reflection) are briefly described in §4.1. In §4.2 we provide details of the Maude implementation, stressing the modularity of the methodology. Such modularity is fully exploited in §5, where we validate our approach by presenting several strategies, heavily reusing components developed for the running example. Next in §6 we describe how to exploit our approach for the analysis of self-assembly strategies. In §7 we discuss related work, and finally in §8 we present some concluding remarks and we hint at ongoing research avenues. Although we assume the reader to have some familiarity with the Maude framework [4], we briefly explain the most relevant Maude technicalities whenever this is needed.

A preliminary version of the present work was presented in [11].

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*) 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. Self-assembly

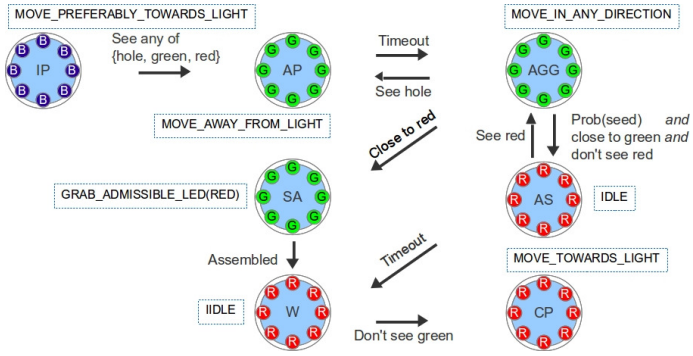

Figure 2: Basic self-assembly response strategy (as proposed in [7]).

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 swarms a challenging scenario to engineer.

In [7] different self-assembly strategies are proposed to carry out tasks that range from hill-crossing and hole-crossing to robot rescue. We focus on the *hole-crossing scenario* as a running case study, where "robots may need to cross a hole while they navigate to a light source" and "a single unit by itself will fall off into the crevice, but if it is a connected body, falling can be prevented".

The experiments described in [7] were conducted on the SWARM-BOT robotic platform [12], whose constituents are called s-bots (see Fig. 1). 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 camera turret with 360 degrees view), and is surrounded by a transparent ring that contains eight RGB colored LEDs distributed uniformly around the ring. The LEDs can serve as a mean of communication with nearby s-bots. For example, the green and red colors can be used to signal the willingness to connect to an existing ensemble or to create a new one, respectively. The ring can also be grasped by other s-bots thanks to a gripper. S-bots have a maximal speed of 30 cm/s and a diameter of 12 cm, and they are able to perceive six different colors: red, green, blue, magenta, yellow and cyan. The experiments reported in [7] concern alternative 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).

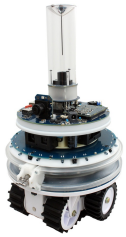

Figure 1: An s-bot.

Running case study: Basic self-assembly response strategy. We focus on the *basic self-assembly response* strategy, where each s-bot moves independently (blue light) until a hole is found, in which case it tries to aggregate (green light) to a

nearby assembly, if some is available, or it becomes the *seed* of a new assembly (red light). A finite state machine representing this strategy is shown in Fig. 2: it is executed independently by each s-bot. States are depicted as bird-eye views of an s-bot (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 (initial letter of the) color of each LED (e.g. `R` for red). The boxed label near each state (like `MOVE_TOWARDS_LIGHT`) is the name of the controller implementing its behaviour, to be detailed in §3.3.1. Transitions are labelled with their firing condition.

In the starting state `IP` (`Independent_Phototaxis`) each s-bot turns on its blue LEDs, and navigates towards the target light source, avoiding the obstacles (e.g. walls or s-bots). If an s-bot detects a hole, 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. If 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 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`, the s-bot waits until a timeout expires and switches to state `W`, unless it sees another red s-bot, in which case it reverts to state `AGG`. Once no green s-bot is visible, assembled "waiting" s-bots switch to state `CP` (`Connected_Phototaxis`) and navigate to the light source. We also consider a state `GR` (`GoalReached`), with yellow LEDs to which the s-bots evolve when they reach the target. It is not shown in Fig. 2 to keep the picture simple.

3. An Architectural Approach to Adaptive Systems

In this section we propose a generic multi-layer architecture for the design of adaptive systems. We start discussing the inspiring principles and the motivations for the design choices in §3.1. Next we describe the architecture in detail in §3.2, and finally we present a concrete instance for the robotic case study in §3.3.

3.1. Inspiring Guidelines and Motivations

The main characteristics of our approach, enumerated as (i)–(iv) in the Introduction, have been inspired by some complementary considerations that we sketch here and elaborate upon in the next subsections: (i) since one of our goals is to highlight the adaptive behaviour of the modelled systems, we commit ourselves to a *white-box* perspective on adaptation, ensuring that the intended adaptive behaviour is explicitly identified in the architecture; (ii) a clear distinction in an adaptive system between the application and the adaptation logics naturally brings to a hierarchical structure, where a base-level adaptive component is controlled by an adaptation manager, as neatly represented in the paradigmatic MAPE-K architecture. By pushing this structuring farther, we envision the possibility of dealing with several levels of adaptivity, leading to a so-called *adaptation tower*; (iii) among the several computational paradigms

that can be used to program adaptive systems, we think that a well-structured use of *computational reflection* is at the same time very powerful and natural to use; (iv) in the realm of modelling and analyzing the specification of adaptive systems, declarative, rule-based styles and quantitative analysis techniques are very natural and widely used in practice.

3.1.1. A White-Box Perspective on Adaptation Based on Control Data

According to largely accepted informal definitions, a software system is "self-adaptive" if *it can modify its behaviour as a reaction to a change in its context of execution*, e.g. in order to better achieve its goals. This definition reflects the point of view of an external observer, and it can be considered a *black-box* (i.e. behavioral, observational) perspective on adaptation. We believe that such a perspective is of little use for *design* purposes, where modularization and reuse are critical aspects, because the black-box view disregards the internal mechanisms by which the adaptive behavior is achieved. As a paradigmatic example related to our case study, some authors 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"* [13]. Therefore under the black-box perspective the same activity (obstacle avoidance) is a form of adaptation in some contexts, but not in others.

In [1] we proposed a *white-box* conceptual framework that requires to identify a set of data, called *control data*, that are supposedly used to control the deviation from the ordinary behavior of a component. Then we define *adaptation* as the *run-time modification of the control data*, and a component is deemed *adaptable* if its control data 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.

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 [14], *policies* in policy-driven languages (e.g. [15]), *aspects* in aspect-oriented languages (e.g. [16]), and even entire programs, in models of computation exhibiting higher-order or reflective features (e.g. [17, 18]).

3.1.2. Hierarchical Structure and the MAPE-K Architecture

Modularity, composability and re-usability are key issues for the success of software models and system architectures. A paradigmatic example is the MAPE-K architecture [2], according to which a self-adaptive system is made of two neatly separated components: a base component implementing the application logic, and a manager component that realizes a control loop that *monitors* the execution through sensors, *analyses* the collected data, *plans* an adaptation strategy,

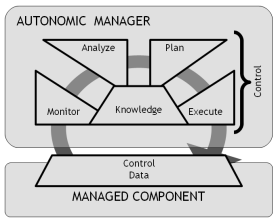

Figure 3: Control data in MAPE.

and finally *executes* the adaptation of the managed component through effectors. All the phases of the control loop access a shared *knowledge* repository. Fig. 3 shows an instance of the MAPE-K architecture, where additionally we highlighted the control data of the managed component: they are identified as the data that are modified by the execute phase of the control loop. When the manager component itself is adaptive, the construction can be iterated, making the architecture compositional in a layered way, which allows one to build towers of adaptive components (see Fig. 6, left).

3.1.3. Computational Reflection

Computational reflection is widely accepted as one of the key instruments to model and build self-adaptive systems (cf. [19, 17, 3]). It is even promoted as a necessary criterion for any self-adaptive software system in [20], where 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 when this is not made explicit.

As explained in [3], computational reflection provides two fundamental mechanisms to realize adaptive systems: *introspection* to let a system observe its own behavior, and *intercession* to let a system modify its own behavior. Indeed, computational reflection can be used to realize (self)-adaptation in a pretty intuitive way. Since reflection allows to manipulate programs as first-class objects, a program can read (i.e., introspect) and modify (i.e., intercede) its own code during execution. Obviously, this powerful mechanism comes at some price. First, it may require an additional computational overhead. In our approach this is not a major concern, since we only consider the phases of modeling and analyzing adaptive systems, and not of deploying them. Second, an unrestricted use of computational reflection could make a system monolithic and extremely difficult to analyze with standard techniques. However, we argue that a localized and structured use of reflection, conforming for example to the MAPE-K architecture, can be very useful for the design of adaptive systems. In fact, the manager component can use reflection to manipulate the code of the base component (which, incidentally, becomes in this case the control data), typically replacing such code with a different one when certain conditions arise. Actually, the manager could even run the base component for a few steps in a sort of simulation mode, before deciding whether to replace it or not: this last scenario is very hard to implement without resorting to reflection.

3.1.4. Rule-based specifications and quantitative analysis

Adaptive systems are very often equipped with strategies based on reactive behaviors and probabilistic algorithms. This is particularly evident in the field of swarm computing due to the successful adoption of popular bio-inspired techniques from the field of Artificial Intelligence, such as ant colony optimization.

In general, reactive behaviors are naturally specified in rule-based languages. More specifically, since our goal is to architecture the specification and rapid prototyping of adaptive systems in general and of swarm robotic strategies in particular, a declarative style of programming is valuable here, as it allows the

designer to concentrate on the high-level (typically reactive) logic of the system behaviour, abstracting from low level (typically procedural) operational details. Furthermore, in the case of swarm robotic strategies both the behaviour of the individual components and the interactions among them are rather simple, even if the emerging behaviour of the swarm can be quite sophisticated. The use of rule-based programming is therefore very adequate (and indeed popular) in this framework, using rules to describe the basic reactions of a component to some elementary stimuli from the environment.

The use of probabilistic algorithms implies that our systems will be intrinsically stochastic. For those systems, it may be neither possible nor useful to prescribe that a certain property is met exactly or that a given goal is achieved completely, but it is more realistic to reason about *to what extent* a property or a goal is reached, after fixing suitable metrics. Therefore *quantitative analysis* appears as the natural way to analyze such systems.

These considerations allow us to identify as a preferred specification formalism a flexible rule-based programming language, equipped with quantitative analysis tools. As anticipated, Maude perfectly meets these requirements.

3.2. A Generic Architecture for Self-Adaptive Systems

We describe here a generic architecture for adaptive components, which combines the key ingredients (i)–(iv) of our approach. We first focus on the structure of layers and their interactions, abstracting from our case study.

3.2.1. Intra-layer architecture

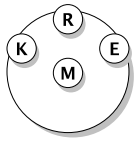

Figure 4: Intra-layer.

Each layer is a component having the structure illustrated in Fig. 4. Its main constituents are: *knowledge* (K), *effects* (E), *rules* (R) and *managed component* (M). Some of them are intentionally on the boundary of the component, since they are part of its interface. The managed component is a lower-level layer having the same structure: clearly, this part is void 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 the local behaviour of the layer, and in particular which effects are generated on the basis of the knowledge and of the interaction with the managed component.

It is important to stress already at this high level of abstraction that we will consider the rules R as the *control data* of a layer. 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 trigger an adaptation.

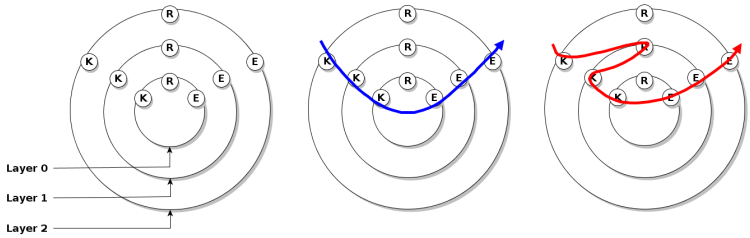

Figure 5: Inter-layer architecture (left), ordinary flow (center), adaptation flow (right).

3.2.2. 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. 5. Fig. 6 shows the correspondence between the layers and the three control loops in a MAPE-K adaptation tower. Of course, the architecture does not impose a specific number of layers, and a new layer can always be added on top of existing ones. The outermost layer interacts with the environment: its knowledge includes information about the environment, as perceived by the component, 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, we may think that while descending the hierarchy, the knowledge becomes simpler, and the generated effects more elementary.

The diagram in the middle of Fig. 5 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.

The rightmost diagram of Fig. 5 corresponds to a phase of adaptation. Here the outermost layer triggers an adaptation at layer 1 by exploiting computational reflection. 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).

3.3. Instantiation of the Architecture to the Robotic Case Study

The concrete architecture of our case study has three layers (see Fig. 6, top-right), which correspond naturally to the way the behaviour of s-bots and the strategies are described in [7]. Moreover, 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*.

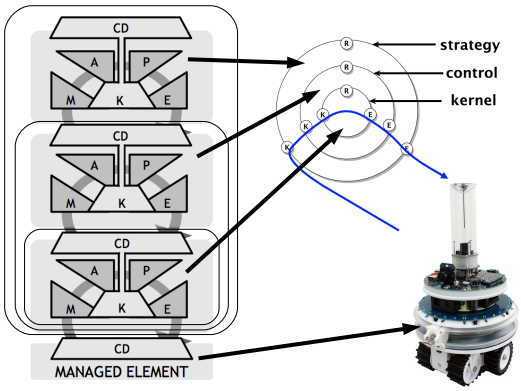

Figure 6: Architecture as an instance of the framework.

Layer 0 (kernel). This layer models the core functionalities of an s-bot (see [7, §3]). The rules implement basic movements and the actioning of the gripper. They are defined once, independently of the strategy or scenario at hand. Layer 0 is similar to what some authors call *hardware abstraction layer* (see e.g. [21]).

Layer 1 (basic control). This layer represents the basic controller managing the s-bot core functionalities, 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 is inhabited by the individual states of state machines modelling the self-assembly strategies, as the one of Fig. 2 (see [7, §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 the entire state machine modelling a self-assembly strategy (e.g. Fig. 2), 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. Different strategies can be implemented by modifying this layer only, as shown in §5.

Implementation details for each layer are outlined in §4.2, together with Maude code snippets. Of course, other layers could be added to the s-bot architecture. For example, a fourth layer could realize a *meta-adaptation* logic, which triggers an adaptation of layer 2 by changing the adaptation strategy.

Orchestrator. 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 the component).

Scheduler. An ordinary discrete-event scheduler activates the scheduled events, allowing an s-bot or the orchestrator to perform its next action. 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 $time + t$", where t can be thought of as the time to execute the effect (e.g. the time required to perform a movement). Such events are handled by the orchestrator. Moreover, for every handled event of a component c the orchestrator also schedules a new event of the form "generate next effect at time $time + t'$" for c.

Arena. The arena defines the scenario where s-bots run. We abstracted the arena into a discrete grid, very much like a chessboard. Each cell of the grid 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 perform an action on (e.g. grip) or move to each of the eight cells surrounding the cells where it resides.

3.3.1. Mapping the Self-Assembly Strategy over the Architecture

Roughly, starting from the finite state machine representing an adaptation strategy, like the basic self-assembly strategy in Fig. 2, we map the automaton to a layer 2 abstraction, its states to layer 1 abstractions and core s-bot functionalities to layer 0 abstractions.

The three layers differ in their sets of rules and in the managed components, 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) are possible but they are not necessary in the presented case study.

Predefined basic controllers for layer 1. We provide a library of predefined basic controllers, implementing basic behaviours for layer 1, summarized in Table 1 and used as building blocks in the definition of the strategies. The `IDLE` controller implements the trivial behaviour of idle s-bots, typical of waiting states. Controller `MOVE_TOWARDS_LIGHT` implements the movement of the s-bot (if not hindered) towards the direction of the light source. 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 the s-bot 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 without obstacles. The behaviour

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 with no obstacle.
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.

"grip a LED of a given color" (controller `GRAB_ADMISSIBLE_LED`) prohibits movements but allows to grip other s-bots (or 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.

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. For example, a suitable color can be used to inform an s-bot that it is near to a line-shaped ensemble, and it has to reach the tail of the line before connecting to it. As for the case of the grippable colors, the set of effects to be outflanked has to be specified in the s-bot's knowledge.

The controllers we just described were used to implement the states of the state machines modelling the self-assembly strategies. The correspondence between the states of the basic self-assembly response strategy and our controllers is depicted in Fig. 2. For example, the state `IP` (`Independent_Phototaxis`) is governed by the basic controller `MOVE_PREFERABLY_TOWARDS_LIGHT`, while state `SA` (`Self_Assemble`) is governed by `GRAB_ADMISSIBLE_LED`.

4. MESSI: Maude Ensemble Strategies Simulator and Inquirer

We discuss here the suitability of Maude [4] as a framework to realize our generic approach (§4.1) and then we describe our implementation (§4.2). Such implementation is called MESSI, and it realizes the architecture of §3.2 and its instantiation for robotic scenarios discussed in §3.3. In this section we focus on our running case study, while other strategies are discussed in §5.

4.1. Rewriting Logic, Maude and the Reflective Russian Dolls

As we have already mentioned in the Introduction, Maude is a convenient setting to realize our solution since it allows us to address its key aspects (i)–(iv). Indeed: (i) Maude's algebraic approach facilitates the formalization of control data; (ii) hierarchical architectures such as the Russian Dolls model have been

promoted and shown to be suitable to specify adaptive systems in Maude; (iii) Maude efficiently supports computational reflection; (iv) Maude is a rule-based language for which probabilistic extensions and analysis techniques are available.

Maude's theoretical foundations are based on *Rewriting Logic* (RL), a powerful computational and logical framework that has been exploited to represent a variety of programming and modeling languages, and where different logics and automated deduction procedures have been represented, mechanized, and reasoned about [5].

A *rewrite theory* $\mathcal{R} = (\Sigma; E; R)$ consists of an equational theory $(\Sigma; E)$ and a set of (possibly conditional) rewrite rules R, where $(\Sigma; E)$ specifies the terms (e.g. the states of a system and their constituents) by means of *operations* and *equations*, and the rules in R specify the dynamics.

Rewriting Logic includes *logical reflection* capabilities. At the ground level, a rewrite theory $\mathcal{R}$ allows to infer a computation step $\mathcal{R} \vdash t \rightarrow t'$ from a term t (representing, for example, a program state) to a term t' (representing the program state after a one-step execution). A universal theory $\mathcal{U}$ may let one infer the computation $\mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{t}) \rightarrow (\overline{\mathcal{R}}, \overline{t'})$ 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, leading to what is called a *tower of reflection*.

Note that since a theory can be represented by a term, it is possible to specify (meta-)rules that change the (meta-representation of the) theory, as $[r] : (\overline{\mathcal{R}}, \overline{t}) \rightarrow (\overline{\mathcal{R}'}, \overline{t'})$, so that reduction continues with a different rewrite theory $\mathcal{R}'$. Pretty obviously, this mechanism enables a powerful form of adaptation, by literally changing the behavior of a system specified as a rewrite theory.

The Maude system [4] provides an implementation of RL, supporting in a systematic and efficient way logical reflection, and providing a rich collection of analysis tools. Roughly, Maude specifications are made of *modules* (which can be functional modules, rewrite theories, or *object-oriented* modules), and powerful module composition operations, including parameterized modules, are supported. With the `META-LEVEL` module Maude provides a library to transform modules to terms with sort `Module` (i.e. modules at the meta-level), so that they can be manipulated as any other term, as well as operations to execute meta-terms (e.g. systems' states brought to the meta-level). Actually, as any term can be brought to the meta-level, this mechanism can be iterated.

The hierarchical structuring of layers can also be specified easily in Maude, by exploiting the object-oriented syntax, explicitly supported by the framework.

It is fair recalling here that 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 also the discussion in §7). The 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. The Maude implementation of our generic architecture can also be considered as an instance of the RRD architecture, where each layer can inject specific adaptation logic in the wrapped components.

A last strong motivation for the use of Maude is the availability of a rich toolset for the analysis of specifications. In particular, given the stochastic nature

```
1  < bot1 : AC2 | K: gripper(open) on(right,none) towards(right,light) ...,
2                 E: emit(up,Green) go(right) ...,
3                 R: mod_is ... endm,
4                 M: < bot1 : AC1 | K: ..., E: ..., R: ...,
5                                   M: < bot1 : AC0 | K:..., E:..., R:... > > >
```

Listing 1: The overall structure of a sample s-bot.

of adaptive systems, we have found very convenient to use the statistical model checker PVeStA [9] to conduct quantitative analysis of our model, parametric with respect to the desired level of statistical confidence, in order to evaluate the effectiveness of the collective robot strategies.

4.2. Implementation Details

We now move our attention to MESSI. In the following, the most relevant notation and features of Maude are explained as they are introduced.

4.2.1. On the structure of adaptive components and of the simulator

We rely on Maude's object-like signature (see [4, Chapter 8]) that allows us to model concurrent systems as collections of objects (*configurations*), each having an identifier, a class and a set of attributes. An object with identifier `oid`, class `cid` and list of attributes `attrs` is written `< oid : cid | attrs >`.

As discussed in §3.3, we use a three-layered architecture to model an s-bot. Three classes are introduced for the different layers, namely `AC0`, `AC1` and `AC2`. Listing 1 depicts the overall structure of an s-bot. 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 containing the specification of the behaviour of the layer, and the fourth one is an object (the inner layer). As a design choice, the objects implementing the layers of an s-bot carry the same identifier (see e.g. the three occurrences of `bot1` in Listing 1, lines `1`, `4` and `5`).

The simulation of s-bots ensembles takes place in a virtual arena. The arena is a set of objects of class `Cell`. Each cell can contain among the attributes (at most) one object of class `AC2` (an s-bot). When needed, the orchestrator moves an s-bot by passing it from the cell containing it to one of the eight adjacent cells. Thus, s-bots have no information about the global environment or their current position, but only about the contiguous cells and the direction towards the target. Intuitively, the cell encapsulating an s-bot may be seen as a further layer wrapping objects of class `AC2`. In fact, it is responsible for updating its knowledge, taking care of its effects (e.g. exposing the status of s-bot's LEDs), and handling the interactions between the s-bot and the scheduler.

4.2.2. Rules of adaptive components

The behaviour of a layer is specified by the rules contained in its `R` attribute. This is a term of sort `Module`, i.e. a meta-representation of a Maude module. Sophisticated control strategies can be realized: the outer component can execute

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

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

or manipulate the rules in `R`, and analyse the outcome. This way, the wrapping component can act as a planner or decision-taker. To illustrate how the flows of execution and information of Fig. 5 discussed in §3.2 are actually implemented, we present one sample rule per layer. For the sake of presentation we abstract from irrelevant details. The full code is available at [10].

Layer 0. This layer implements the core functionalities of s-bots. For example, the rule (keyword `rl`) of Listing 2, named `admissibleMovements`, computes the set of directions an s-bot can move to. A rule can be applied to a Maude term t if its left-hand side (LHS) (here the object in line 2, containing variables `oid`, `k0`, `e0` and `A0`) matches a subterm of t with substitution σ, and the application consists of replacing the matched sub-term with the term obtained by applying σ to the right-hand side, i.e. the object in line 3 that follows the symbol `=>`.

Rule `admissibleMovements` allows to rewrite an object of class `AC0` to itself, enriching its effects with the result of evaluating `canMoveTo(freeDirs(k0))`. Operator `canMoveTo` is a constructor, i.e. it cannot be reduced, acting as a container. Instead `freeDirs(k0)` reduces to the set of directions the s-bot can move to, i.e. those not containing walls or other s-bots. This is realized by analysing the facts of shape `on(dir,content)` contained in the knowledge `k0`. For example, if `on(right,none)` is a fact in `k0`, then `right` is a direction in `freeDirs(k0)`. It is worth noting that the behaviour of `freeDirs()` is specified with Maude *equations* instead of rules: they have higher priority than rules, meaning that rules are applied only to terms in normal form with respect to the equations (two sample equations are discussed later in Listing 5). Note that rule `admissibleMovements` consumes the constant `1Step` from the knowledge of the object: intuitively, it is a token used to inhibit further applications of the rule, obtaining a one-step rewriting. Variable `A0` matches all remaining attributes of object `oid`, and they are left unchanged by the application of the rule.

Layer 1. Objects of class `AC1` implement the basic controllers discussed in § 3.3.1 and correspond to the states of the state machine of Fig. 2. Rules of this layer can execute the component of the lower level (an object of class `AC0`) providing additional knowledge to it, and elaborating its effects. The conditional rule of Listing 3 (keyword `crl`) implements (part of) the logic of state `IP`, computing the direction towards which to move.

The rule can be applied to a matched sub-term only if its (firing) condition (i.e. the `if` clause of lines `6`–`10`) is satisfied under the matching. Maude allows four kinds of conditions: 1) equational conditions, to check equality of two terms t and t'; 2) membership predicates, to check if a term t has sort s; 3) rewrite expressions, to check if a term t can be rewritten to a term t'; and 4) matching

```
1  crl [MovePreferablyTowardsLight]:
2      < oid : AC1 | K: k1 1Step, 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: k0e, E: e0, R: m0, A0e >, A1 >
6  if             < oid : AC0 | K: k0e, E: e0 canMoveTo(freeDirs), A0e >
7     := execute(< oid : AC0 | K: 1Step update1To0(k1,k0), E: e0, R: m0, A0 >)
8  /\ prefDirs := intersection(freeDirs, dirsToLight(k1))
9  /\ dir := if prefDirs =/= empty then uniformlyChooseDir(prefDirs)
10                                 else uniformlyChooseDir(freeDirs) fi .
```

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

conditions, written p `:=` t to check if the pattern p matches the term (obtained by reducing) t. When the match is successful, a matching condition can be regarded as a sort of assignment that binds the variables in p. As a special case, boolean predicates can be used in conditions that are checked for equality with respect to the value true. For the rule of Listing 3, the condition is the conjunction (`/\`) of three matching conditions, that are evaluated sequentially.

The operator `execute` exploits Maude's meta-level functionalities to obtain the meta-representation of one object, and to execute it via the rules meta-represented in its attribute `R: m`. In particular, in line 7 of Listing 3, the operator applies a single rule of `m0` to the managed component `< oid : AC0 ... >`, after having updated its knowledge. In fact, the operation `update1To0(k1,k0)` implements a (controlled) propagation of the knowledge from layer 1 to layer 0, filtering `k1` before updating `k0` (e.g. information about the surrounding cells is propagated, but not the one about the target).

The assignment of the first matching condition (lines 6 and 7) updates knowledge and attributes of the managed component `< oid : AC0 ... >` (cf. variables `k0e` and `A0e`, with `e` a mnemonic suffix for "executed") and binds `freeDirs` to the directions towards which the managed component can move. This is used in the second matching condition (line 8) to compute the intersection between `freeDirs` and the directions towards the light, evaluated reducing `dirsToLight(k1)`. The resulting set of directions is bound to `prefDirs`. Finally, by resorting to the `if_then_else_fi` statement, in the third matching condition (lines 9-10) `dir` is bound to a direction randomly chosen from `prefDirs`, or from `freeDirs` if the former set is empty. Comparing the LHS (lines 2–3) and the RHS (lines 4–5), one sees that the overall effect of the rule is the production of a new effect at layer 1, `go(dir)`, and the update of the knowledge of the 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: 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. 2). It monitors the environment and

```
1  crl [adaptAndExecute]:
2     < oid : AC2 | K: k2 nextEffect, 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: k1e, E: e1a, R: m1a, A1e >, A2a >
6  if          < oid : AC2 | K: k2a, E: e2a,
7                    M: < oid : AC1 | K: k1a, E: e1a, R: m1a, A1a >, A2a >
8    := adapt(< oid : AC2 | K: k2 , E: e2 ,
9                    M: < oid : AC1 | K: k1 , E: e1 , R: m1 , A1  >, A2  >)
10 /\            <oid: AC1 | K: k1e                        , E: e1a effect , A1e>
11   := execute(<oid: AC1 | K: 1Step update2To1(k2a,k1a), E: e1a, R: m1a, A1a>).
```

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

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 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 of this layer. It is triggered by the token `nextEffect` (line 2), generated by the scheduler and propagated to the s-bot by the cell containing it. The rule consists of an adaptation phase (lines 6–9) followed by an execution phase (lines 10–11), both performed on the managed component and both triggered by the two matching conditions of the rule. The variables bound by the two matching conditions have either `a` or `e` as suffix, to recall the phase where they have been introduced (*adapt* or *execute*).

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. Operation `adapt` is specified by a set of equations, two of which are presented in Listing 5. The first one is a *conditional equation* (keyword `ceq`), which encodes the transition of Fig. 2 from state `AGG` to `SA`, labeled with `Close to red`. It states that if an s-bot in state `AGG` sees in its neighborhood an s-bot with red LEDs on, then it must pass to state `SA` and turn on its green LEDs. Also, the rules of the managed component are changed: the new module `m1a` 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 used module is `GRAB_ADMISSIBLE_LED`, and it contains the rules defining the controller for layer 1 with the same name, which is used in state `SA` as shown in Fig. 2. Note also how, in line 4, we invoke again the operation `adapt`. This allows to consecutively compute more than one adaptation phase if necessary.

In order to implement a self-assembly strategy we only need to specify an equation for every transition in the corresponding state machine, plus the default rule `noAdaptationNeeded` of Listing 5 line 9 where `owise` is a special attribute telling the Maude interpreter to apply the equation only if none of the others can be applied. The latter is a plain (non-conditional) equation, as it can be

```
ceq [AggToSA]:
   adapt(< oid2 : AC2 | K: state(AGG) k2, E: e2               ,
                        M: < oid1 : AC1 | R: m1 , E: e1  , A1 > , A2 >)
 = adapt(< oid2 : AC2 | K: state(SA)  k2, E: setAllLEDs(green),
                        M: < oid1 : AC1 | R: m1a, E: none, A1 > , A2 >)
 if seeEffect(led(red),k2)
 /\ m1a  := upModule('GRAB_ADMISSIBLE_LED,false) .

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

Listing 5: One of the equations for the strategy of Fig. 2 (`AGG` → `SA`) plus the default case.

```
mod IDLE is
  pr AC1 .
endm

mod MOVE_PREFERABLY_TOWARDS_LIGHT is
  pr AC1 .

  crl [MovePreferablyTowardsLight]: *** shown in Listing 3
endm

mod GRAB_ADMISSIBLE_LED is
  pr AC1 .

  var k1 k0 k0e e1 e0 e0e : Config . var m0 : Module .
  vars A0 A0e A1 : AttributeSet . var oid : Oid .
  var dirsWithEmiss : Set{Direction} . var dir : Direction .

  crl [gripColorEmission] :
     < oid : AC1 | K: k1 1Step, E: e1                          ,
                   M: < oid : AC0 | K: k0 , E: e0 , R: m0, A0  > , A1 >
  => < oid : AC1 | K: k1       , E: insertEffect(attach(dir),e1),
                   M: < oid : AC0 | K: k0e, E: e0e, R: m0, A0e > , A1 >
  if            <oid : AC0 | K: k0e, E: e0e canAttachTo(dirsWithEmiss), A0e>
     := execute(<oid : AC0 | K: 1Step update1To0(k1,k0), E: e0 , R: m0, A0 >)
  /\ dirsWithEmiss =/= empty
  /\ dir := uniformlyChooseDir(dirsWithEmiss) .
endm
```

Listing 6: Some of the predefined basic controllers discussed in §3.3.1.

seen by the use of keyword `eq`.

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

Coming back to the rule of Listing 4, once the adaptation phase is concluded, the second sub-condition of the rule takes care of the one-step execution of the (possibly adapted) managed component. Finally, the effects generated by layer 1 are wrapped in the constructors `event` and `schedule`, and the resulting term is added to the effects of layer 2, that will be propagated to the scheduler by the encosing cell.

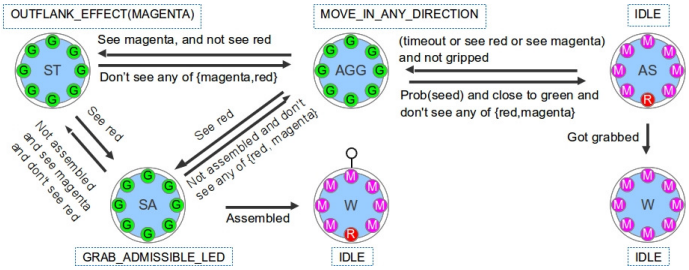

Figure 7: An assembly strategy to form ensembles shaped as lines.

4.2.3. Implementation of the Predefined Basic Controllers for Layer 1

We present here some details of the implementation of the predefined basic controllers discussed in §3.3.1. Each of these controllers is defined by a Maude module with the same name. Listing 6 shows the implementation of three of them. They all import in a protecting way (keyword `pr`) the module `AC1` that contains the definition of the common operations of this layer (like `update1To0` and `insertEffect`).

Lines `1`–`3` show the trivial code of controller `IDLE`. Lines `5-9` sketch the code of controller `MOVE_PREFERABLY_TOWARDS_LIGHT`, whose only rule has been already discussed (see Listing 3). Finally, lines `11-27` show the code of controller `GRAB_ADMISSIBLE_LED`. This controller is used in states in which the s-bot does not move, but only tries to grip other s-bots. In particular, in lines `23`–`24` we first execute the component of layer 0, providing to it the `1Step` token, and updating its knowledge with the one of layer 1. We thus 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 `25`), in line `26` we randomly select one of the directions. Finally, in line `21` this direction is inserted in the effects of layer 1.

5. Design of Self-Assembly Strategies

We now demonstrate the wide applicability of our architecture (and of MESSI) by discussing some implemented strategies. In particular, since the basic behaviours are already implemented by the controllers presented in §3.3.1, for each new strategy only the state transitions have to be defined.

5.1. A Self-Assembly Strategy for Line-Forming Scenarios

We start with the simple self-assembly strategy of Fig. 7 to form line-shaped ensembles. In the scenario there are no obstacles, and three colors are used to guide the assembling: 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 for "*the s-bot is part of an ensemble, but other s-bots should connect elsewhere*". The s-bots

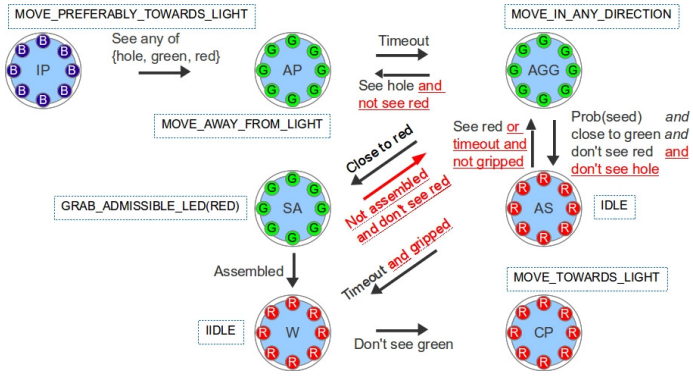

Figure 8: Variant of the basic self-assembly response strategy to deal with *unexpected behaviours*.

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. Only the LED opposite to the direction of the gripper is set to red (the others are set to magenta): this will force the generation of line-shaped ensembles. If an s-bot in state `Aggregate` does not see any red color emission, but sees a magenta one, it knows that it is near to an ensemble and searches for the tail of the line by changing to the basic controller `OUTFLANK_EFFECT`. This case is captured by the transition from state `Aggregate` to state `Search_Tail` labeled with "`See magenta, and not see red`".

If an s-bot in state `Self_assembly` does not perceive any assembly anymore (i.e. neither red nor magenta color emissions), then it reverts to state `Aggregate`. If instead a magenta color emission is still perceived (but not a red one), then the s-bot starts searching for the tail of the assembly (state `Search_Tail`).

The last outgoing transition from `Aggregate` handles the case when the s-bot does not perceive any ensemble, 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`, sets a randomly selected LED to red and the others to magenta. The strategy has a last transition from `Assembly_Seed` to `Aggregate` with firing condition "`(timeout or see red or see magenta) and not gripped`". In order to limit the number of ensembles, an s-bot in state `Assembly_Seed` can revert to state `Aggregate`. This 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 magenta`).

5.2. Self-Assembly Strategies for Hole-Crossing Scenarios

We now discuss two variants of the basic self-assembly response strategy of Fig. 2. 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 `Aggregate`. The differences with respect to the original strategy are highlighted with red color and underlined. This variant solves some unexpected and undesired behaviours of the original strategy, which were discovered during our experiments. A brief description of this variant and the motivations for its proposal are given in §6.1.

The strategy of Fig. 9 is instead a major variation. It is obtained by composing the above strategy (Fig. 8) with the one depicted in Fig. 7 for the line-forming scenario. As argued in [7], the idea is that if s-bots compose in lines, then there are more chances to succeed in crossing holes. In §6.2 we evaluate this intuitive reasoning by comparing the performances of the strategies of Fig. 2, 8 and 9.

Colors are used as in the line-forming strategy. The first two transitions (from state `IP` to `AP`, and from `AP` to `AGG`) are taken from the strategy of Fig. 8, the only difference is that a further LED emission is considered (magenta): initially, an s-bot navigates independently towards the 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 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 arise.

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

In the second case a magenta LED emission is perceived, and neither a red LED nor a hole are perceived: the s-bot changes 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 tail of the line (i.e. a red emission). This is the behaviour offered by the controller `OUTFLANK_EFFECT`. Once a red LED emission is perceived 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 not perceived anymore, then the s-bot reverts to `Aggregate`.

In the third case, neither assemblies nor holes are perceived, but only single s-bots (green emissions). As for the other strategies, 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 to `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 assembly by allowing to be gripped only in the directions opposite to the light source (as depicted in state `AS` of Fig. 9). Now, after the expiration of a timeout, if the s-bot has been gripped it changes

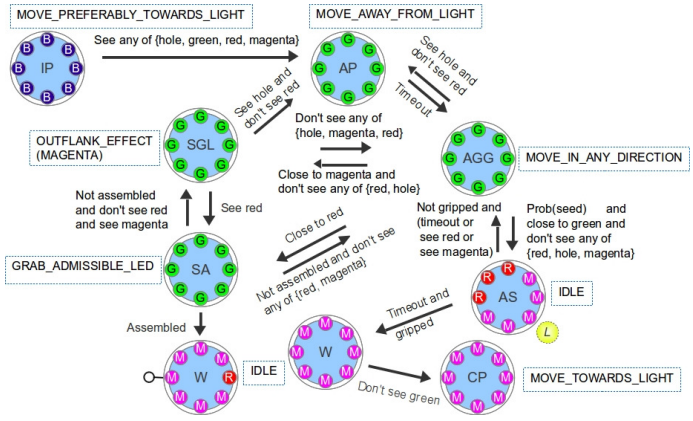

Figure 9: Variant of the basic self-assembly response strategy where s-bots assemble in lines.

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 for the other strategies, if an s-bot in state `Assembly_Seed` is not gripped before the expiration of a timeout, or if it perceives other assemblies, then it changes back to state `Aggregate`.

6. Analysis of Self-Assembly Strategies

In the early development phases we mainly focused on debugging the implemented strategies by resorting to discrete-event simulations (§6.1). A couple of trial-and-error iterations 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 (via Maude's 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*, we resorted to statistical model checking (§6.2).

6.1. Simulations

Simulations are performed thanks to the discrete-event simulator mentioned in §3.3. Valuable help has been obtained implementing an exporter from Maude `Configuration` terms to DOT graphs [22] offering the automatic generation of images from states, and of animations from images: they have greatly facilitated the debugging of our code. Fig. 10 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 target

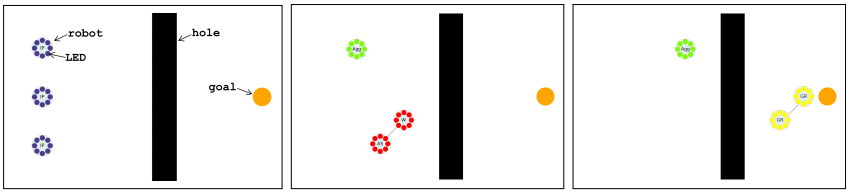

Figure 10: Three states of a simulation: initial (left), assembly (middle), final (right).

of the s-bots, i.e. a light source (the orange circle to the right). After some steps, two s-bots finally get assembled (middle of Fig. 10), and can safely cross the hole (right of Fig. 10), while the third one remains in the left part of the arena.

While performing simulations with different scenarios and parameters we observed several unexpected behaviors, e.g. not-assembled s-bots erroneously believing to be part of an assembly, and hence moving into the hole. In other simulations we noticed instead 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* [7]?

Examining the strategy we discovered that the two behaviors are not disallowed, and that they are originated by the two transitions (see Fig. 2) outgoing from the state `AS` (willing to be grabbed). The first transition leads to state `W`, triggered by the expiration of a timeout, while the second one leads to state `AGG` (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 `AS` to state `W` even if no other s-bot is attached to it; in this case it can evolve to state `CP` and thus move towards the target falling in the hole. Considering the other transition, once an s-bot i grabs an s-bot j, i becomes "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 `AS` to state `AGG`, thus trying to grab i. Interestingly, we noted that the two unexpected behaviors strongly depend on the duration of the timeout: a short timeout favors the first behaviour, a long timeout favors the second.

We also noted further unexpected or undesired behaviors: for example, an s-bot may remain stuck in state `SA`, because no recovery mechanism is provided for the cases in which an s-bot in state `SA` fails to assemble with another s-bot.

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 of [7] 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 unexpected behaviors can be fixed as depicted in Fig. 8 by adding a transition from state `SA` to state `AGG`, and further conditions to the two mentioned transitions of the strategy. In particular, the transition from `AS` to `W` requires a further condition to ensure that the s-bot has been gripped, while the transition from `AS` to `AGG` requires that the s-bot is not gripped.

6.2. Statistical Model Checking

We resort to statistical model checking (see e.g. [23, 24, 9]) in order to answer questions like *what is the probability that* n *robots reach the target?* Or *what is the expected number of a given measure?* The technique, where probabilities and quantities are estimated, does not yield the absolute confidence of qualitative or probabilistic model checking, but allow us to analyze (up to some statistical errors) larger scenarios and to deal with the stochastic nature of probabilistic systems. This is helpful to compare different strategies on the basis of their estimated performances. As usual with statistical analysis techniques, we made our models totally probabilistic, getting rid of all unquantified non-determinism.

We use PVeStA [9], a parallel statistical model checker that evaluates quantitative temporal expressions (QuaTEx) [8], allowing to query about expected values of real-typed properties. Estimations are done with respect to a confidence interval specified by α and δ: the tool performs n independent simulations, with n large enough to grant that if a property is estimated as x, then, with probability $(1 - \alpha)$, its actual value belongs to the interval $[x - \delta/2, x + \delta/2]$. 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 less simulations are required. A presentation of PVeStA and of QuaTEx is out of the scope of this paper, we refer to [9, 24] for the first and to [8] for the latter.

6.2.1. Assumptions, parameters and hardware specifications.

Experiments ran on a Linux machine with 64 GB of RAM and 48 cores with 2.00 GHz clock. For all our experiments we fixed α to 0.05, while we used different δ depending on the range of values of the estimated properties (e.g. 0.05 when estimating probabilities and 0.2 when counting the number of robots satisfying a given condition). To reach such a level of confidence PVeStA ran hundreds of simulations for each property, with an average run time per property 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 §4, we abstracted arenas to discrete grids. Each cell can contain at most one s-bot, and it is accordingly dimensioned (i.e. 12×12 cm^2). Our representation of s-bot's actions and perceptions is influenced by this abstraction: s-bots perform one-step movements to one of the eight surrounding cells. We fixed 0.6 seconds as the time necessary to move to an adjacent cell. Similarly, s-bots can grip other s-bots in one of the eight surrounding cells. In particular, we decided to abstract from the time necessary to rotate the gripper, and we set to 2 seconds the time to grip an s-bot. In the same way, s-bots perceptions are limited to the eight surrounding cells, the only global information perceived being the direction towards the target light source.

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

We performed a comparative analysis of the basic self-assembly response strategy of Fig. 2, our variant of Fig. 8 to get rid of the unexpected behaviors, and the one of Fig. 9 where s-bots assemble in lines. The aim of these experiments was twofold: on the one hand we wanted to show that the strategies of Figg. 8

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	0.58	0.61	0.57	0.96
	6-BOTS	0.88	0.94	0.82	0.99
	9-BOTS	0.97	0.97	0.93	1.00
Q_2	3-BOTS	1.03	1.26	1.20	1.98
	6-BOTS	3.38	3.40	3.16	5.10
	9-BOTS	4.75	5.85	5.14	8.04

Table 2: The result of the statistical model checking procedure.

and 9 do not generate the unexpected behaviors arising in (our implementation of) the original one. We focused on the behaviour in which two s-bots grab each other, and 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 success rate of the strategies to study the influence of the unexpected behaviours and of the shape of the assemblies. We thus defined the 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?*

Our analysis regarded three configurations: 3 s-bots in a 11×7 grid, 6 s-bots in a 12×8 grid, and 9 s-bots in a 14×10 grid. As usual we fixed a sufficiently large simulation duration. Finally, we fixed `Prob(seed)` to 0.7 (i.e. the probability to become the seed of a new assembly), three times the time necessary to grip an s-bot as timeout for state `AS`, and the time necessary to perform a movement as timeout for state `AP`.

Summarizing the results, the double gripping behaviour is absent in our two strategies, while it arises often in the original one. Moreover, our variant depicted in Fig. 8 exhibits the best success rate both for Q_1 and Q_2, while the cross hole line-forming strategy of Fig. 9 does not have the expected performances.

Table 2 details the results of our analysis: "BSRS" stands for basic self-assembly response strategy (Fig. 2), "BSRS$^+$" stands for our variant (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 discussed later.

We already discussed Q_0. Considering 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 target for the 3- and 6-BOTS cases. For the 9-BOTS case we have high probabilities for any strategy: clearly, as there are more s-bots, the probability that at least one reaches the target is pushed closer to one. Finally, considering Q_2, the strategy BSRS$^+$ has the best performances.

Noteworthy, we expected BSRS$^+$LINE to have much better performances. We suspected that the strategy has low performances due to the fact that often the formed lines are parallel to the hole, and hence the s-bots fall in it. Note that the light is perceived from one to three directions: e.g. 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 that are parallel to the hole. In the considered configurations we only have vertical holes, hence it is sufficient to forbid the seed of the line to turn red the LEDs in direction up and down. 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 performances of the other strategies.

As previously stated, our aim is not the design of new self-assembly strategies, but rather showing how these can be specified and evaluated following our approach. In order to better understand and compare the performances of the four strategies, in Fig. 11 we depict the ratio of robots reaching their target when varying the number of robots according to Q_2. Interestingly, what we found is that the extra time spent in forming lines pays back if we manage to obtain lines that are not parallel to the hole.

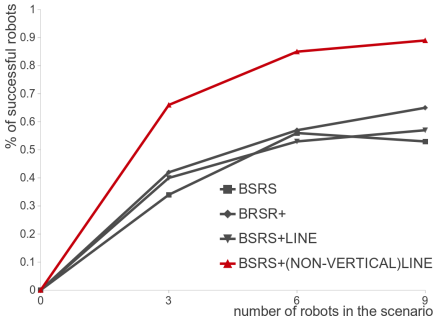

Figure 11: Ratio of successful robots (Q_2).

7. Related Work

We discuss in this section some works that have influenced and inspired us, together with further related works. We recall that the key characteristics of our approach are (i) a white-box approach to adaptation based on the notion of control data; (ii) a hierarchical architecture that provides the basic structure of the adaptation logic; (iii) computational reflection as the main mechanism to realize the adaptation logic; (iv) probabilistic rule-based specifications and quantitative verification techniques to specify and analyze the adaptation logic. In particular, the above points (ii)–(iv) are discussed in §7.1–7.3, respectively. For a detailed discussion of point (i) we refer to [1].

7.1. Hierarchical architectures and models for adaptive systems

As we have already mentioned, the hierarchical architecture of our approach is mainly based on seminal architectural models for autonomic and adaptive systems like the MAPE-K [2] and the Russian Dolls architectures [17, 6]. We discuss here some of the approaches that build upon such models as well.

Among those, PAGODA (Policy And GOal based Distributed Autonomy) [21] is one of the closest as it also relies on the Russian Dolls model and on Maude. PAGODA is a modular architecture for specifying and prototyping systems of autonomous cooperating agents. One of the main differences with respect to our work is that in PAGODA adaptation (called *coordination*) is mainly realized by intercepting and manipulating messages (e.g. to monitor and enforce communication policies) rather than on the meta-programming mechanisms

based on reflection adopted in our approach. In this regard, the advantages of PAGODA are that it can be realized in platforms not supporting computational reflection and that even if realized in a platform supporting it, controlled objects can be kept as black boxes that hide their internal behavior which may be mandatory in those scenarios (different from ours) where code privacy is a key concern. The advantage of our approach is that computational reflection can be exploited to realize sophisticated forms of adaptation as already explained in §3.1.3 and later in §7.2.

From the architectural point of view there are other important differences. PAGODA imposes a two-layered architecture (the local nodes and the global system). Similar two-layered structures can be found in other models for adaptive systems as well (we refer to the examples cited in [1]). Our approach, instead, allows us to build a hierarchy up to any level which enables us to use different levels of adaptation. This feature allows the designer to decompose the adaptation logic into different levels of abstraction, providing a flexible and modular way of realizing adaptive behaviors. Many approaches featuring more than one layer of adaptation can be found in the literature (e.g. [25, 26, 27]), as well as examples of architectures with an arbitrary number of layers (cf. the discussion of [28]).

The component architecture of PAGODA is also different from ours. In particular, PAGODA components (called *nodes* or *agents*) are composed of a knowledge base, a reasoner, a monitor, a learner, and a hardware abstraction layer. Similar structures are used in many other agent-oriented systems, since they provide convenient adaptation mechanisms. An example can be found in simpAL [29], which combines the concurrent object and actor paradigms. Our approach does not impose the use of reasoning or learning techniques but they may be certainly adopted in concrete instances. However, we have preferred to keep the architecture of nodes as simple as possible to focus mainly on an explicit representation of control data and basic ingredients for communication. In addition, we have observed that the self-assembly strategies proposed by the swarm computing community often adhere to the so-called *reactive pattern* [30], typical of reactive agents, which are characterized by the lack of explicit goals and of sophisticated adaptation mechanism such as planning or learning.

An approach similar in spirit to PAGODA is PobSAM (Policy-based Self-Adaptive Model) [15], a 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 [31] 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 only two layers, composed of *managed actors*, implementing the functional behavior of the system, and *autonomic manager (meta-)actors*, controlling 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, despite the differences in the architecture and in the use of reflection (see the

discussion on PAGODA and the one in §7.2), PobSAM and our approach share the idea of having rules as a high-level object of adaptation.

Close to our work is also the *composite actor model* of [28] 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. The focus of [28], however is not on adaptive systems, but rather on well-formedness scheduling criteria to ensure correctness of statistical model checking techniques (discussed later in §7.3). Self-organising assembly systems, instead, are addressed in [32] using Maude as specification language and the Chemical Abstract Machine (CHAM) [33] as architecture and programming paradigm. CHAM shares some similarities with RRDs in general and our approach in particular, including the use of (reactive) rule-based specifications and nested architectures. The approach described in [32], however, does not rely on reflection nor on quantitative analysis.

There are many other modelling frameworks for adaptive systems based on hierarchical structures, besides Russian Dolls. A differentiating advantage of our approach, however, is that each component of an adaptive system (be it a manager, a managed component, or both) is represented with the same programming abstraction, allowing us to reuse the same techniques at each layer.

7.2. Reflection-based adaptation

Computational reflection is widely accepted as one of the key instruments to model and build self-adaptive systems [3], but of course other linguistic mechanisms can be used as well (see e.g. the discussion of [34]). One of the main advantages of reflection against other adaptation mechanisms (such as the previously discussed policy-based adaptation of PAGODA [21] and PobSAM [15]) is that it enables one to perform reasoning based on the actual model of each controlled object. For instance, a controller can test the reaction of an object under different (hypothetical) environments and/or explore and solve the non-deterministic choices of the controlled object. Exploiting such *internal models* is widely applied in areas such as Robotics and Control Theory, and it is advocated as a necessary feature in adaptive software sytems [35]. As a matter of fact, we use this idea in our case study, where the component of each layer controls which of the possible actions of the component in the underlying managed component are available. For instance, the component in layer 0 just exposes all possible functionalities of the robots which are selected by the basic behaviors of layer 1. Such mechanisms cannot be immediately realized in policy-based approaches like the ones discussed above.

A prominent example where reflection plays a major role in the development of adaptive systems is the FOrmal Reference Model for Self-adaptation (FORMS) [18]. 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 forbids a base layer to modify the upper layer directly (a hot-linking mechanism would be needed to synchronize them).

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 [36] as a convenient mechanism for modelling self-adaptive systems [37]. The authors describe how to adopt in KLAIM three paradigms for adaptation: two that focus on the language-level, namely, context-oriented programming [38] and dynamic aspect-oriented programming [16], 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 do not 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 [39] in order to specify and analyse collective robotic systems. As in our case, the analysis is based on statistical model checking. Letting apart the differences, this work shows the raising interest 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 [40]).

7.3. Modelling and analysing probabilistic systems with Maude

Moving to the concrete setting of Maude, several works (e.g. [28, 41, 42, 43, 44, 45]) can be found in the literature where probabilistic systems specified by probabilistic rewrite theories in PMaude [8] are analyzed using probabilistic analysis methods like statistical model checking. Among others, we highlight the already mentioned composite actors of [28] and the probabilistic strategy language of [45]. The authors of the latter face a typical problem that arises when modelling and analyzing probabilistic systems: one needs to ensure that the underlying system is free of non-determinism 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 [28] 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 [8, 42]. 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 or 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 [45] offers an elegant and flexible solution that allows to keep the non-determinism in the system, which is solved at the level of the strategy. Roughly, the idea is to assign probabilities to non-deterministic transitions by assigning weights to rule matches that are later normalized into values in the interval $[0, 1]$.

8. Conclusions and further works

The main contributions of our paper are: (1) a description of a generic architecture for adaptive systems that builds on the white-box approach to adaptation and the notion of control data [1] together with the MAPE-K architecture and the Reflective Russian Dolls model; and (2) the validation of our architecture and of MESSI, its instantiation in Maude, for the early prototyping of adaptive systems (§5) and for their analysis (§6) exploiting the Maude toolset.

The distinguishing features of our approach are: (i) a white-box approach to adaptation based on the notion of control data; (ii) a hierarchical architecture to modularize the design; (iii) computational reflection as the main adaptation mechanism; (iv) probabilistic rule-based specifications and quantitative verification techniques to specify and analyze the adaptation logic. The suitability of such features has been discussed throughout the paper and is witnessed by prominent examples from the literature on self-adaptive systems (cf. §7). The approach can be realized in any framework providing suitable support for (i)–(iv). In this paper we have shown that Maude is a good candidate and we plan to investigate the suitability of other frameworks in future work.

We discussed our sources of inspiration in §7, the main ones being early approaches to coordination and adaptation based on distributed object reflection [6, 17, 21]. The original contribution of our work does not lie on the individual features (i)–(iv) but on their combination in a natural, convenient and flexible way. We consider the support of each individual feature by well-accepted theories and techniques one of the strongest points of our work. Other interesting aspects of our work are the clear and neat representation and role of control data in the architecture, and the fact that, to the best of our knowledge, this is the first analysis of self-assembly strategies based on statistical model checking.

The case study of self-assembly strategies for robot swarms [7] has contributed to assess our approach and framework. Overall, the implemented strategies (§5) and the reuse of a small set of basic controllers witness the generality and accessibility of our approach, thanks to the modularity provided by our hierarchical structuring. Moreover, the experimentations show 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.

Some ongoing work is aimed to improve the overall performance of MESSI analysis engine. The idea is to extend PVeStA and QuaTEx in order to allow to query sets of properties at once, by reusing the output of the same group of simulations for all properties instead of running different simulations for each property. The tool, called MultiVeStA [46], is under development and its usability and performance need to be assessed by tackling new case studies and performing more complex analyses. 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 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 formal reasoning capabilities. Second, we plan to develop a compositional reasoning technique that exploits the hierarchical structure of the layered architecture.

Acknowledgements

We are grateful to the organizers of the *AWASS 2012 summer school* for the opportunity to mentor a case study based on the experience of this paper, and to Herwig Guggi, Ilias Gerostathopoulos and Rosario Surace for their work. We are also grateful to the anonymous reviewers for their constructive criticisms.

References

[1] 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.

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

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

[4] 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.

[5] J. Meseguer, Twenty years of rewriting logic, Journal of Logic and Algebraic Programming 81 (7-8) (2012) 721–781.

[6] 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.

[7] 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.

[8] 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.

[9] M. AlTurki, J. Meseguer, PVeStA: A parallel statistical model checking and quantitative analysis tool, in: A. Corradini, B. Klin, C. Cîrstea (Eds.), CALCO 2011, Vol. 6859 of LNCS, Springer, 2011, pp. 386–392.

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

[11] R. Bruni, A. Corradini, F. Gadducci, A. Lluch-Lafuente, A. Vandin, Modelling and analyzing adaptive self-assembly strategies with Maude, in: F. Durán (Ed.), WRLA 2012, Vol. 7571 of LNCS, Springer, 2012, pp. 118–138.

[12] 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.

[13] 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.

[14] G. Salvaneschi, C. Ghezzi, M. Pradella, Context-oriented programming: A programming paradigm for autonomic systems, CoRR abs/1105.0069v2.

[15] 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.

[16] P. Greenwood, L. Blair., Using dynamic aspect-oriented programming to implement an autonomic system, in: R. Filman, M. Haupt, K. Mehner, M. Mezini (Eds.), DAW 2004, RIACS, 2004, pp. 76–88.

[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] 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) 8:1–8:61.

[19] J. Dowling, T. Schäfer, V. Cahill, P. Haraszti, B. Redmond, Using reflection to support dynamic adaptation of system software: A case study driven evaluation, in: W. Cazzola, R. J. Stroud, F. Tisato (Eds.), OORaSE 1999, Vol. 1826 of LNCS, Springer, 2000, pp. 169–188.

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

[21] 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.

[22] GraphViz - Graph Visualization Software, http://www.graphviz.org/.

[23] 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.

[24] 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.

[25] I. Lanese, A. Bucchiarone, F. Montesi, A framework for rule-based dynamic adaptation, in: M. Wirsing, M. Hofmann, A. Rauschmayer (Eds.), TGC 2010, Vol. 6084 of LNCS, Springer, 2010, pp. 284–300.

[26] A. Bucchiarone, M. Pistore, H. Raik, R. Kazhamiakin, Adaptation of service-based business processes by context-aware replanning, in: K.-J. Lin, C. Huemer, M. B. Blake, B. Benatallah (Eds.), SOCA 2011, IEEE Computer Society, 2011, pp. 1–8.

[27] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, D. A. Karr, Building adaptive systems using ensemble, Software: Practice and Experience 28 (9) (1998) 963–979.

[28] J. M. Jonas Eckhardt, Tobias Mühlbauer, M. Wirsing, Statistical model-checking for composite actor systems, in: N. Martì-Oliet, M. Palomino (Eds.), WADT 2012, Vol. 7841 of LNCS, Springer, 2013, pp. 143–160.

[29] A. Ricci, A. Santi, From actors to agent-oriented programming abstractions in simpal, in: SPLASH 2012, ACM, 2012, pp. 73–74.

[30] 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.

[31] G. Agha, Actors: a model of concurrent computation in distributed systems, MIT Press, 1986.

[32] R. Frei, G. D. M. Serugendo, T.-F. Serbanuta, Ambient intelligence in self-organising assembly systems using the chemical reaction model, Journal of Ambient Intelligence and Humanized Computing 1 (3) (2010) 163–184.

[33] G. Berry, G. Boudol, The chemical abstract machine, Theoretical Computer Science 96 (1) (1992) 217–248.

[34] 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.

[35] R. Calinescu, C. Ghezzi, M. Z. Kwiatkowska, R. Mirandola, Self-adaptive software needs quantitative verification at runtime, Communications of ACM 55 (9) (2012) 69–77.

[36] R. De Nicola, G. L. Ferrari, R. Pugliese, KLAIM: A kernel language for agents interaction and mobility, IEEE Transactions on Software Engineering 24 (5) (1998) 315–330.

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

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

[39] 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 2012, Vol. 7635 of LNCS, Springer, 2012, pp. 54–70.

[40] M. Brambilla, C. Pinciroli, M. Birattari, M. Dorigo, Property-driven design for swarm robotics, in: AAMAS 2012, IFAAMAS, 2012, pp. 139–146.

[41] 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.

[42] M. AlTurki, J. Meseguer, C. A. Gunter, Probabilistic modeling and analysis of DoS protection for the ASV protocol, in: D. J. Dougherty, S. Escobar (Eds.), SecReT 2008, Vol. 234 of ENTCS, Elsevier, 2009, pp. 3–18.

[43] M. Wirsing, J. Eckhardt, T. Mühlbauer, J. Meseguer, Design and analysis of cloud-based architectures with KLAIM and Maude, in: F. Durán (Ed.), WRLA 2012, Vol. 7571 of LNCS, Springer, 2012, pp. 54–82.

[44] 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 2012, Vol. 7212 of LNCS, Springer, 2012, pp. 78–93.

[45] L. Bentea, P. C. Ölveczky, A probabilistic strategy language for probabilistic rewrite theories, in: N. Martì-Oliet, M. Palomino (Eds.), WADT 2012, Vol. 7841 of LNCS, Springer, 2013, pp. 77–94.

[46] S. Sebastio, A. Vandin, MultiVeSta: Statistical model checking for discrete event simulators, in: ValueTools 2013, ACM, 2013, to appear.