# Towards a Formal Approach for Prototyping and Verifying Self-Adaptive Systems

Juan F. Inglés-Romero[1] and Cristina Vicente-Chicote[2]

[1] Dpto. Tecnologías de la Información y Comunicaciones,
Universidad Politécnica de Cartagena, Edificio Antigones, 30202 Cartagena, Spain
juanfran.ingles@upct.es
[2] Quercus Software Engineering Group (QSEG),
Universidad de Extremadura, Avda. de la Universidad S/N, 10003 Cáceres, Spain
cristinav@unex.es

**Abstract.** Software adaptation is becoming increasingly important as more and more applications need to dynamically adapt their structure and behavior to cope with changing contexts, available resources and user requirements. Maude is a high-performance reflective language and system, supporting both equational and rewriting logic specification and programming for a wide range of applications. In this paper we describe our experience in using Maude for prototyping and verifying self-adaptive systems. In order to illustrate the benefits of adopting a formal approach based on Maude to develop self-adaptive systems we present a case study in the robotics domain.

**Keywords:** Self-Adaptive Systems, Prototyping, Maude, VML.

## 1 Introduction

Nowadays, significant research efforts are focused on advancing the development of (self-) adaptive systems. In spite of that, some major issues remain still open in this field [1][2]. One of the main challenges is how to formally specify, design, verify, and implement applications that need to adapt themselves at runtime to cope with changing contexts, available resources and user requirements.

Adaptation in itself is nothing new, but it has been generally implemented in an ad-hoc way, that is, developers try to predict future execution conditions and embed the adaptation decisions needed to deal with them in their application code. This usually leads to increased complexity (business logic polluted with adaptation concerns) and poor reuse of adaptation mechanisms among applications [1]. The use of formal methods can help alleviating the limitations of current approaches to self-adaptive system development. In particular, they can provide developers with rigorous tools for testing and assuring the correctness of the adaptive behavior of their systems. This is a remarkable open issue, since only a few research efforts seem to be focused on the formal analysis and verification of self-adaptive systems.

Maude [3] is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications. Maude and its supporting tools can be used in three, mutually reinforcing ways: as a declarative programming language, as an executable formal specification language, and as a formal verification framework. A Maude program can be seen as an executable mathematical model of a system. Thus, using Maude for prototyping self-adaptive systems enables their simulation, formal analysis (e.g., reachability/likelihood of certain system configurations), and verification (e.g., testing that the system reaches a consistent configuration for all given contexts). Furthermore, Maude can help designers to assure, among other properties, the consistency and correctness of self-adaptive system specifications.

This paper presents our experience in using Maude for prototyping and verifying component-based self-adaptive systems. The main contribution derives from the implementation of a robotic case study, in which we model the variability of the system and manually translate this logic into Maude. This experience has allowed us to witness some of the benefits of adopting a formal approach when developing self-adaptive systems. This research continues our previous works on self-adaptive system design [4] and implementation [5] using the DiVA framework [6], and on modeling self-adaptive system variability using the *Variability Modeling Language* (VML) [7].

## 2 Robotic Example

In this section, we introduce the robotic example designed to illustrate the versatility of Maude for prototyping self-adaptive systems. It is worth noting that, although the example presented here belongs to the robotics domain, in which the need for self-adaptation (to changing contexts and available resources) is quite obvious, we believe that our proposal is general enough to be adopted in other application domains in which modeling, simulating and formally verifying self-adaptation is an issue.

### 2.1 Adaptation Scenario

The proposed case study takes place in a room, where a small robot moves around randomly avoiding obstacles. In order to improve this basic functionality in terms of safety, power consumption and performance, the robot follows an adaptation strategy that decides on the following variation points: (1) the signaling type; (2) the signaling intensity; and (3) the robot velocity. There are two possible variants for the signaling type (light or acoustic), while the signaling intensity and the robot velocity may take any integer value in the range 0-100. The adaptation strategy decides the best possible configuration (selection of variants for each variation point) according to the current context. The context variables considered in the case study are the ambient light, the ambient noise and the robot battery level, all of them integers ranging from 0 to 100.

The goodness of each configuration is calculated based on the impact of each variant on the three properties being considered, that is: safety, power consumption and performance. The following considerations are made concerning *safety* (making others aware of the presence of the robot in the surroundings): (1) light signaling is more convenient than acoustic signaling when the ambient light is low; and (2) the higher the ambient noise (might indicate a crowded environment), the higher must be the signaling intensity and the lower the robot velocity. Regarding *power consumption*, the greater the signaling intensity and the robot velocity the greater the power consumption. Thus, if the battery level is low, both the velocity and the signaling intensity need to be limited. Finally, concerning *performance*, the higher the velocity the shorter the time it takes to the robot to reach its goal position. Obviously, maximizing safety and performance, while simultaneously minimizing power consumption, imposes conflicting requirements. Thus, the adaptation strategy will need to find the right balance among these requirements to achieve the best possible configuration for a given context, even if some (or none) of them are optimized individually.

## 2.2 Component-Based Software Architecture

The component-based software architecture developed for the case study is sketched in Figure 1. As other self-adaptive systems [2], the proposed design includes: (1) a reconfigurable part, comprising the optional and/or parameterized components; (2) a set of monitoring components; and (3) an adaptation control unit.

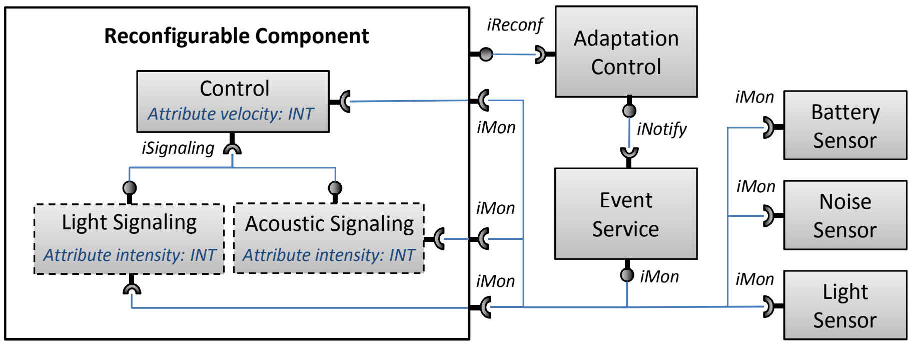

**Fig. 1.** Component-based software architecture for the example

The *Reconfigurable Component* gathers the elements of the system that are susceptible to change at runtime. Among them, the *Control Component* implements the core robot functionality, that is, the motion control and the obstacle avoidance. This component includes a parameter called *velocity* that regulates the robot motion speed, and is responsible for activating or deactivating the robot signaling through the *iSignaling* interface. The *Reconfigurable Component* also contains two optional components, each one implementing one of the alternative ways for signaling the robot position: *Light Signaling* and *Acoustic Signaling*. Both these components contain an *intensity* parameter that regulates the frequency of the light and the acoustic signals, respectively. The three variation points available at the *Reconfigurable Component* (i.e., selecting one of the two alternative signaling components and setting the velocity and the intensity parameters) will need to be fixed at runtime by the adaptation strategy, implemented by the *Adaptation Control*, as detailed later.

The monitoring part of the architecture provides the context-aware support for the adaptation. It comprises (1) a set of sensors (*Noise Sensor*, *Light Sensor* and *Battery Sensor*) and monitors (*Control*, *Light Signaling* and *Acoustic Signaling*) for acquiring information both from the environment (external context) and from the system itself (internal context); and (2) the *Event Service* component that receives the context information from the former components via the *iMon* interface, and notifies the changes in the context to the *Adaptation Control* component through the *iNotify* interface.

Finally, the *Adaptation Control* component implements the adaptation strategy which, on the basis of the context changes notified by the *Event Service* component, decides which is the best possible configuration (variant selection) for the *Reconfigurable Component* and applies the required changes via the *iReconf* interface.

We have selected the E-puck robot [8] as our target platform. E-pucks are low-cost mobile robots with a large range of sensors and actuators that make them appropriate for testing self-adaptive applications. However, this robot presents limitations that prevent executing Maude on it. For this reason, we use a distributed architecture where some components are deployed in an external PC, which communicates with the robot via Bluetooth (e.g., this is the case of the Adaptation Control, which relies on Maude for executing the adaptation strategy). As our intention is to use Maude for prototyping the system (as a support for its design, simulation, and verification) and not for its final implementation, this seems to be a valid approach.

## 3 Modeling Variability with VML

This section introduces the *Variability Modeling Language (VML)*. VML provides a mechanism to express how a system should adapt at run-time to improve its performance under changing conditions. The current version of VML has been developed using a Model-Driven Engineering (MDE) approach. We have created a textual editor for VML using the Xtext framework [9], including some advanced features such as syntax checking and coloring facilities, and a completion assistant. As the focus of this paper is not VML, we will only present the essentials needed for modeling the case study. Then, the resulting VML model will be used as the starting point for obtaining the Maude specification included in Section 4.

In a VML model, we first need to define the ***variation points***. Aligned with *Dynamic Software Product Lines (DSPL)* [10], VML variation points represent points in the software where different variants might be chosen to derive the system configuration at run-time. Therefore, variation points determine the decision space of VML, i.e., the answer to *what* can change. As shown in Listing 1, variation points (`varpoint`), as all the other VML variables, belong to a certain data type. VML includes three basic data types: enumerators, ranges of numbers, and booleans. For instance, the *velocity* variation point is an enumerator that set the robot speed using three possible variants: SLOW, NORMAL or FAST. After defining the variation points, we need to specify the ***context variables*** (`context`), which identify the situations in which variation points need to be adapted. Listing 1 shows three context variables: (1) the ambient lighting, (2) the ambient noise, and (3) the battery level. Note that contexts and variation points have been modeled to abstract the original parameters presented in Section 2, most of them defined as integers ranging from 0 to 100. This abstraction reduces the complexity considering only the relevant values for each variable.

At this point, we need to define *how* variation points are set according to the contexts. This is achieved through *properties* (`property`) and *ECA (event-condition-action) rules* (`rule`). ***Properties*** specify the features of the system that need to be optimized, i.e., minimized or maximized. Properties are defined using two functions: *priorities* and *definitions*. Definitions characterize the property in terms of variation points (i.e., definitions are the objective functions to be optimized). For instance, in Listing 1, we define the *performance* property as a linear function of the velocity variation point (the faster the robot accomplishes its task, the better its performance). It is worth noting that property definitions can be characterized using the technical specifications of the hardware (e.g., to know how much power each component consumes), simulation or empirical data from experiments. On the other hand, *priorities* describe the importance of each property in terms of one or more context variables (i.e., priorities weight the objective functions). For instance, *power consumption* becomes more relevant whether the battery level decreases below 20%. Otherwise, *power consumption* is not considered an issue and, as a consequence, its impact on the adaptation process is very small. Opposite to definitions, priorities are characterized in a more subjective way, depending on the designer experience. Regarding the ***ECA rules,*** they define direct relationships between context variables and variation points. As shown in Listing 1, the left-hand side of a rule expresses a trigger condition (depending on one or more context variables) and its right-hand side sets the variation point. For example, the decision of which signaling (acoustic or light) to select in each situation is modeled using rules. Basically, when there is ambient light (lightning = true) the first rule selects the acoustic signaling. Otherwise, the second rule selects the light signaling.

Regarding the execution semantics, VML models specify a constrained optimization problem, that is, they use syntactic sugar for describing the global weight function that optimizes the variation points and, as a result, allows improving the overall system quality. This global function is obtained by aggregating the property definitions (terms to be optimized), weighted by their corresponding priorities. Besides, the ECA rules state constraints that need to be satisfied. At this point, it is worth noting that VML variation points and contexts are high-level variables that somehow abstract architectural elements (e.g., components or component parameters). For instance, the *velocity* is linked to the parameter velocity in *Control* component, and the battery level is obtained from the *Battery Sensor* component (see Figure 1). This abstraction allows VML to be independent of the underlying architecture, what, among other benefits, enables the reuse of the models in different platforms and scenarios. In the next section we start from the VML model, shown in Listing 1, to obtain an equivalent Maude specification.

```
type impact : enum {VERY_HIGH(5), HIGH(4), MEDIUM(3),
                    LOW(1), VERY_LOW(2), NON_EFFECT(0)};
context lighting  : boolean;
context noise     : enum {LOW, MEDIUM, FULL};
context battery   : enum {NORMAL, NOISY, VERY_NOISY};

rule rule1 : lighting = true          => signalType = ACOUSTIC;
rule rule2 : lighting = false => signalType = LIGHT;
rule rule2 : battery <> FULL          => velocity <> FAST;

property safety : impact maximized {
    priorities:
        case noise <> NORMAL : impact.MEDIUM
        default: impact.NON_EFFECT;
    definitions:
        f(signalIntensity) = signalIntensity/20;
        f(velocity) = velocity/20 - 1; }

property powerConsumption : impact minimized {
     priorities:
       case battery = LOW : impact.VERY_HIGH
       default: impact.LOW;
     definitions:
        f(signalIntensity) = signalIntensity/40 + 1.5;
        f(velocity) =  velocity/40 + 1.5; }

property performance : impact maximized {
    priorities:
       case noise <> NOISY and battery = LOW:  impact.VERY_LOW
       default: impact.HIGH;
    definitions:
       f(velocity) = velocity/20;
}
varpoint signalType        : enum {ACOUSTIC, LIGHT};
varpoint signalIntensity   : enum {LOW(20), MEDIUM(60), HIGH(100)};
varpoint velocity          : enum {SLOW(20), NORMAL(60), FAST(100)};
```

**Listing 1.** Variability model described using VML for the robotic example

## 4 Prototyping Self-Adaptation with Maude

In order to learn about the convenience of using formal tools like Maude in the development of self-adaptive systems, this section describes the Maude specification for the adaptation logic modeled in Section 3. Note that, for the lack of space, we do not provide the complete Maude specification, but only the essential concepts for modeling the self-adaptation logic.

### 4.1 Overall Proposed Approach

We have implemented our example with Core Maude using an object-based programming approach. This allows us to model self-adaptive systems as configurations (collections) of objects and messages that represent (a snapshot of) a possible system state. Each object has an identifier, a class, and a set of attributes. For instance, the expression `< oid : cid | attr1, attr2 >` represents an object with identifier `oid`, belonging to the class `cid`, and with two attributes `attr1` and `attr2`. On the other hand, messages include an identifier and a list of arguments. For example, the expression `mid (arg0, arg1)` represents a message with identifier `mid` and arguments `arg0` and `arg1`. The idea behind using a set of objects and messages to represent the system state is that we can specify the adaptation behavior as a set of rewrite rules that consume and produce objects and messages, i.e., that evolve the system state.

As in most self-adaptive systems [2], the adaptation loop comprises three processes, namely: (1) gathering and assessing the current context, (2) reasoning on the best adaptation possible, and (3) performing the system reconfiguration. In our case, Maude carries out all these processes through the *Adaptation Control* component, which simply allows Maude to interact with the component architecture. The interface between Maude and the *Adaptation Control* is summarized in Table 1. Figure 2 outlines the steps of the algorithm that implements this adaptation loop. Each of these steps is further detailed in the following subsections.

**Table 1.** Interface between Maude and the *Adaptation Control* component

| **Command** | **Description** |
|---|---|
| *Start* | Starts the adaptation loop |
| *synchArch <component : String> <parameter : String> <value : String>* | Synchronizes the Maude architecture representation with the actual architecture implementation. Example: *synchArch "LightSignaling" "state" "running"* → Maude is notified of the actual state of the *LightSignaling* comp. |
| *init ( <battery : INT>, <noise : INT>, <light : INT> )* | Maude is notified of the initial low-level context variables Example: *init ( 100, 55, 20 )* |
| *battery <value : INT>* | Updates the battery (0-100). Example: *battery 23* |
| *noise <value : INT>* | Updates the ambient noise (0-100) Example: *noise 67* |
| *light <value : INT>* | Updates the ambient light (0-100) Example: *light 10* |
| *notify <component : String> <parameter : String> <value : String>* | Maude is notified of a change in a component. Example: *notify "control" "velocity" "23"* → The control component notifies that the velocity has changed to 23 |
| *command <component : String> <parameter : String> <value : String>* | Maude sends a reconfiguration command. Example: *command "control" "velocity" "11"* → The velocity of control component must be changed to 11 |

## 4.2 Initialization

At run-time, Maude needs to keep the dynamic state of the adaptation process through an internal representation (or model) of: (1) the *current context*, to know when a new situation (e.g., a new battery level) produces significant changes to require an adaptation, and (2) the *current component architecture*, to trace what modifications are needed to obtain an adapted configuration of the system from the current one. Consequently, prior to starting the adaptation loop, an initialization function needs to set up the context and the architecture representations. This function is labeled in Figure 2 as "*Init context and synchronize representation*".

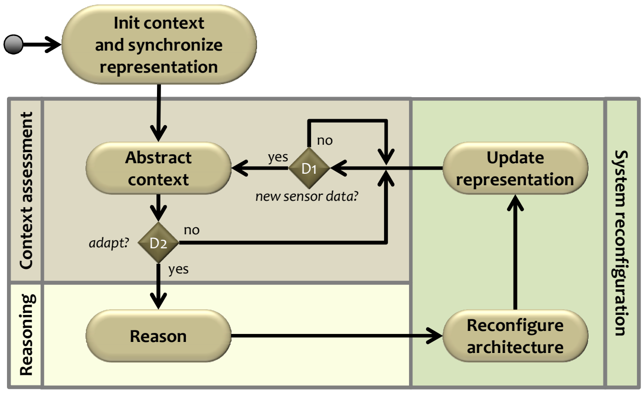

**Fig. 2.** Outline of the adaptation loop

**The Context Representation.** It should be both detailed enough to gather all the contextual information relevant for the adaptation, and abstract enough to enable the system to efficiently reason on it. This representation considers the same description as the one given for the context variables in the VML model (see Listing 1). The function that computes the values from the components to obtain a more abstract representation (e.g., deciding when the integer provided from the *Battery Sensor* is LOW, MEDIUM or FULL) will be later detailed in section 4.3. A possible configuration of the context model in Maude could be as follows:

```
< ctx : Context | batt  : FULL, noise : NORMAL, light : false >
```

**The Architecture Representation.** Similarly to the context model, maintaining an explicit reflection model that abstracts the actual running system is essential to efficiently decide on it and execute the required reconfigurations. This model needs to be synchronized with the actual component-based system architecture in order to provide the adaptation logic with up-to-date information. With regard to adaptation, the only relevant information contained in the system architecture for the example (see Figure 1) is the list of components gathered in the *Reconfigurable Component* (neither the component interfaces nor the connectors are modeled). Each component in this list is modeled in Maude with an object containing, at least, two attributes: *name* (String) and *state* ∈ {*RUNNING*, *STOPPED*}. An additional attribute will be added for each parameter defined in each component. A possible configuration of the architecture model could be as follows (please, note that the state of the AcousticSignaling component is STOPPED, meaning that it is not present in the actual system architecture):

```
<c : Control | name : "control", state : RUNNING, velocity : 5 >
<l : LightSignaling | name : "lsig", state : RUNNING, intensity : 50 >
<a : AcousticSignaling | name : "asig", state :STOPPED, intensity:50>
```

The initialization process is addressed through two rewrite rules. On the one hand, the `arch-synchronization` rewrite rule is triggered each time the Adaptation Control component sends to Maude a `synchArch` message (see Table 1), which updates the state and attributes of the corresponding component in the architecture representation. This occurs when the components belonging to the *Reconfigurable Component* initially notify their state via *iMon* interface. On the other hand, the `init-context` rewrite rule updates the context representation after receiving the measurements of all sensors through an `init` message (see Table 1).

## 4.3 Context Assessment

The main functions of the *Context Assessment* process are: (1) to receive the context information from the sensors (see the `battery`, `noise` and `light` commands in Table 1), (2) to translate this raw data into consistent values for the more abstract context representation, and (3) to launch the reasoning process in case the changes in the context variables are significant enough. These three functions are represented in Figure 2 in the decision node *D1*, the operation "*Abstract context*" and the decision node *D2*, respectively.

In order to address the context assessment, we have implemented three rewrite rules (one for each context variable). Thus, a new message from a sensor will trigger its corresponding rule (1) to apply an abstraction based on fixed thresholds (e.g., we consider the *battery* to be *FULL* when we receive a value greater than 80), and (2) to determine whether an adaptation is required according to how much the context has changed. In the current implementation, the adaptation process starts only if a variable in the context representation changes. In this case, a `reasoner` message is created to trigger the rules performing the reasoning process, as detailed next.

## 4.4 Reasoning

The *Reasoning* function (see Figure 2) implements the core self-adaptation logic as it computes the best configuration possible for a given context, that is, it selects the set of variants that jointly optimize the overall system performance. The Maude specification does not only rely on the description of the properties and rules described in the VML model, but also on their semantics. Therefore, as we mentioned in Section 3, Maude will need to solve the underlying constrained optimization problem described in the VML model.

**The Variability Representation.** It translates the variation points included in the VML model into Maude. As shown below, each variant is modeled in Maude with an object containing the following attributes: *name*, *dimension* (ID of the variation point the variant belongs to), *safety*, *consumption* and *performance* (impact of the variant in each property resulted from evaluating the definition function), *score* and *state*. Note that, as in the VML model, we abstract the architecture details. For instance, whereas *signaling intensity* is a component attribute in the range 0-100, both in VML and in Maude, we only consider three variants (*LOW, MEDIUM, HIGH*). This abstraction, together with the one provided by the context and architecture representations, significantly simplifies the reasoning process.

```
< v : Variant | name : "slow", dimension : "velocity", safety : 3,
  consumption : 2, efficiency : 1, score : 0, state : AVAILABLE >
```

The reasoning approach followed in this research is based on the method described in [11], which combines the use of adaptation rules and the optimization of property-based adaptation goals. Our adaptation rules have been implemented as two Maude rewrite rules, `non-available` and `required`. Both these rules are executed once for each variant object, updating its *state* attribute according to the current context. The `non-available` rule sets the *state* of those variants that are inconsistent with the current context (i.e., cannot be selected during the subsequent optimization process) as NON-AVAILABLE. For example, if the *battery* is not *FULL*, then the variant *FAST* is marked as NON-AVAILABLE (see *rule3* in the VML model). The `required` rule sets the *state* of those variants that, according to the current context, need to be compulsorily selected as REQUIRED. For example, if *light* is *true*, then the variant *ACOUSTIC* is marked as REQUIRED (see *rule1* in the VML model).

In order to cope with the optimization of property-based adaptation goals, we have implemented two additional rewrite rules: `calculate-scores` and `search-solution`. The first of these rules is triggered once for each variant and calculates the attribute *score* of those marked as AVAILABLE. The calculation of the *score* is based on: (1) the impact of each variant on the three system properties (evaluation of the function for the property *definition*); and (2) the importance of each property in the current context (the evaluation of the function for the property *priority*). Finally, the `search-solution` rule finds the best possible system configuration for the current context, i.e., the combination of variants that, together, obtain the highest score.

## 4.5 System Reconfiguration

The main functions of the *System Reconfiguration* process are: (1) to create a reconfiguration plan (sequence of reconfiguration commands) that adapts the architecture representation according to the decisions made by the *Reasoning* function, and (2) to synchronize the architecture representation with the run-time system architecture. To implement these functions, labeled in Figure 2 as "*Reconfigure architecture*" and "*Update representation*", we have implemented two Maude rewrite rules: `reconfigure` and `notification-when-pending`.

The `reconfigure` rule produces a set of reconfiguration commands (see *command* in Table 1) for those components that need to be modified (i.e., those for which the state or other attribute has changed). This is achieved by making the difference between the current architecture representation and the one that has just been derived from the selected variants. Moreover, the `notification-when-pending` rule is executed whenever a real component (belonging to the *Reconfigurable Component*) notifies that it has changed in response to a reconfiguration command (see *notify* in Table 1). These notifications cause the architecture representation to be updated to reflect the current situation. It is worth noting that Maude registers all the reconfiguration commands sent and not acknowledged yet. Context messages are discarded as long as there are pending notifications. This prevents the execution of new adaptation loops while the architecture representation is not completely synchronized.

## 5 Formal Verification of the Adaptation Strategy

Maude can be used not only for simulating self-adaptation strategies, but also for formally checking that the specification satisfies some important properties under certain conditions, or for obtaining useful counterexamples showing that some property is violated. As this kind of model-checking analysis can be a powerful tool for self-adaptive system designers, this section illustrates how Maude allows checking invariants using the *search* command.

As previously mentioned, the *Adaptation Control* component (see Figure 1) relies on Maude for executing the adaptation strategy. Thus, sensor components interact (indirectly) with Maude to provide context information (e.g., the battery level or the ambient noise). Although this scheme is appropriate for simulating or running the system, it is not for exhaustively checking whether the model satisfies a given condition. Therefore, we need to automatically generate context configurations to enable the Maude *search* command to explore the adaptation space. For instance, in the proposed example, the robot should not run simultaneously the light signaling and the acoustic signaling components. We can verify that this property holds in all situations using the *search* command as shown below.

```
search in SIMULATOR :
     start
=>+
     contextModel(<ctxObj : Context| light:LIGHT,noise:NOISE,batt:BATT>)
     archModel(
        < ctrlObj : Control | ATTS-CFG >
        < lSigObj : LightSignaling | state : LSIG-ST, ATTS-CFG' >
        < aSigObj : AcousticSignaling | state : ASIG-ST, ATTS-CFG" > )
     CONF:Configuration
such that (LSIG-ST == RUNNING) and (ASIG-ST == RUNNING) .
```

Given an initial configuration (declared before the arrow), this command explores the adaptation space looking for the pattern that has to be reached (declared after the arrow). Firstly, the starting configuration is the message *start* that activates the context generator and the adaptation loop (see Figure 2). Then, Maude searches for the context (i.e., the value for the variables LIGHT, NOISE and BATT) and the architecture that are set when the *state* of both the *LightSignaling* and the *AcousticSignalig* is RUNNING. If the search command returns no solution it means that on the initial state, and on all states reachable from it, the predicate is an invariant. Similarly, the command could also be used to analyze the reachability of a valid configuration, for example, to find out under which contexts the adaptation selects a velocity greater than 80.

## 6 Related Work

Significant research efforts are being invested to try overcoming the limitations of current ad-hoc approaches to (self-) adaptive system development. These efforts have given rise to new adaptation-enabling frameworks and middlewares, and new languages supporting adaptation primitives [2]. However, most current approaches do not offer either a formal specification of the adaptation processes, nor a formal reasoning support for testing, assessing and verifying the adaptation logic. This issue has been highlighted as a major challenge in several works [1][2].

Some of the existing frameworks provide some kind of support for self-adaptive system simulation and verification. For instance, as part of the DiaSuite Project [12], the DiaSpec Domain Specific Language (DSL) enables the specification of Sense/Compute/Control (SCC) applications, where the interaction with the physical environment is essential. Basically, DiaSpec allows designing applications independently of the platform, describing entities (e.g., components or devices) and controllers, which execute actions on entities when a context situation is reached. DiaSpec models can be simulated to test the SCC applications before their deployment. These models can be simulated using DiaSim without requiring any coding effort. DiaSim provides a graphical editor to define simulation scenarios and a 2D-renderer to monitor simulated applications. The main benefit of this approach is the graphical representation of the environment, where the user can easily see the interactions between entities. However, this approach is quite limited as it does not enable to extensively explore the possible states of the system to verify invariants, find out erroneous reconfigurations, or prove the correctness of the rules.

MUSIC [13] is a framework that supports self-adaptation in mobile applications based on component-based architectures. MUSIC proposes a methodology where the contextual information is modeled extending an ontology, and the architecture reconfigurations are expressed via goal policies, through utility functions in the architectural elements. MUSIC provides a tool for static validation aimed to detect errors and omissions in the specification (e.g., for type checking). On the other hand, MUSIC also offers a simulation tool that enables developers to observe and analyze the effects of context changes and adaptations. In contrast to our approach, where we model separately the application logic and the variability involved in the adaptation process, this approach is too coupled to the underlying architecture.

In the field of Dynamic Software Product Lines (DSPL) [10], MOSKitt4SPL [14] enables designers to model dynamic variability by means of (1) feature models, describing the possible configurations to which the system can evolve, and (2) resolution models, defining the reconfigurations in terms of feature activation/deactivation associated with a context condition. These specifications are automatically transformed into state machines representing the Adaptation Space, where the states are the possible system configurations and the transitions the migration paths. This approach enables to analyze the Adaptation Space and automatically refine the model specifications to ensure the following behavioral issues: (1) determinism, (2) reversibility, (3) absence of redundancy, and (4) nonexistence of cycles. The main advantage of this approach is the possibility of using the well-established Finite State Machine (FSM) theory to analyze the system specification. However, it does not rely on a formal framework like Maude to implement specific verification algorithms or the simulator.

Also in this line, the DiVA [6] Project provides a tool-supported methodology with an integrated framework for managing dynamic variability. DiVA proposes an early validation at design-time to discover faults in the adaptation specification using simulation, i.e., the user manually selects relevant contexts and then analyzes the results from the adaptation logic. Although DiVA has multiple solvers available both for simulating and running the system, this approach seems to be insufficient to check invariants in large models with many context variables or to prove the consistency and correctness of the specification.

## 7 Lessons Learned and Future Research Plans

To this point, we have presented our experience in using Maude for prototyping and verifying component-based self-adaptive systems. Next, we draw some conclusions and lessons learned from this experience, and outline our future research plans.

### Roles of Formal Tools in the Development of Self-Adaptive Systems

At design-time, formal tools, like Maude, can support self-adaptive system modeling, at least, in two ways. On the one hand, they can be used to check the ***semantic correctness*** of the variability models. For instance, considering the VML language presented in section 3, Maude can help checking data types (e.g., we cannot assign a Boolean value to an enumerated variable), detecting unused elements (e.g., a variation point that is never used), avoiding recursive assignments (e.g., x=x+1), or preventing contradictory statements (e.g., two incompatible rules). On the other hand, formal tools allow ***verification*** and ***simulation*** of the adaptation logic. For example, Maude provides the `search` command, which explores the reachable state space looking for a given configuration. This command is a simple, yet powerful method for checking invariants, as introduced in section 5. Furthermore, Maude enables the simulation of the system specification starting from any given state. This can be very useful for adjusting some parameters of the adaptation logic.

At run-time, we can apply formal tools to ***validate the system reconfigurations*** before they are actually performed, as it is not reasonable to make the system migrate to an invalid configuration. For example, DiVA [6] applies an online validation with Kermeta [15] at run-time to check that all the invariants hold. Apart from that, formal tools can also be considered as a means to ***operate with models*** in the adaptation process. The main benefit of this approach is that the implementation of the operations can be verified. For instance, as mentioned in section 4, we could firstly adapt an architecture model, and then, reflect the changes on the real system according to the difference operation between the adapted model and the current one. Therefore, it seems to be possible to deal with similar techniques in the field of *models@runtime* [16] combining model transformations with formal tools like Maude [17]. Finally, if the performance is satisfactory, we could consider formal tools as the final ***adaptation engine***.

### Maude for Prototyping Self-Adaptation

The first benefit of using Maude for prototyping self-adaptive systems stems from its capability to provide designers with ***executable mathematical models*** of these systems. This capability becomes essential for adjusting and validating their adaptation behavior. Specifically, Maude can assist designers in (1) adjusting the abstraction level of context variables and variation points in the VML model with regards to the architecture details (e.g., the integers provided by the light sensor are translated into a boolean context variable to indicate whether there is or not light enough). The designer needs to decide how many values are enough to characterize a context variable or a variation point; (2) adjusting the design-time functions that specify the VML properties to be optimized; and (3) establishing the right links between the architectural elements and the VML variables (e.g., selecting thresholds to establish what means *false* or *true* in a boolean context variable). Regarding the abstraction level of variables in VML, it is worth noting that the number of considered alternatives has an impact on the adaptation stability (e.g., the more context values, the more potential situations to manage and, consequently, it may cause continuous and inefficient system reconfigurations), the computational cost (e.g., the more variants to handle the higher the cost) and the effectiveness of the process (e.g., an insufficient number of alternatives makes adaptation useless).

Concerning the ***main limitations*** we found when implementing the robotic example, Maude seems not to offer a simple method for solving constrained optimization problems. The current approach relies on exploring all the possible combination of variants to finally decide the best one in terms of the adaptation properties. Thus, the practical feasibility of this problem critically depends on the number of variants. Another barrier for the developers is the difficulty for debugging Maude programs, due to the concurrent nature of its rules and the scarcely legible traces it returns during the execution. We figured out that this difficulty increases exponentially as the number of rules grows, since it becomes more and more complex to follow the interactions between the Maude rules, causing undesired states in the system.

### Automatic Code Generation from VML

We have identified some common Maude structures (partly described in section 4) that could be easily generated from a VML model (e.g., the context and the variability representation). However, part of the Maude specification cannot be generated from the VML model, as it also depends on the component-based architecture model, in particular of its reconfigurable and monitoring parts (related with the VML variation points and context variables, respectively).

For the future, we plan to continue exploring the potentials of Maude, in particular, for verifying the completeness and correctness of the self-adaptive behavior specifications. We also plan to link this work with our previous experience with VML [7] and with the MDE approach proposed by DiVA [6].

**Acknowledgements and Additional Material.** Juan F. Inglés-Romero thanks Fundación Séneca-CARM for a research grant (Exp. 15561/FPI/10). The authors would like to thank Prof. Antonio Vallecillo and Mr. Javier Troya for their helpful insights about modeling with Maude.

Additional material related to this work can be found in the following website: `https://sites.google.com/site/cvicentechicote/home/publications/varis2013`

## References

1. Cheng, B.H.C., et al.: Software engineering for self-adaptive systems: A research roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
2. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems 4(2), 1–42 (2009)
3. Maude website, `http://maude.cs.uiuc.edu/`
4. Inglés-Romero, J.F., et al.: Using Models@Runtime for Designing Adaptive Robotics Software: an Experience Report. In: 1st Int'l Workshop on Model-Based Engineering for Robotics (RoSym), Oslo, Norway, October 3-8 (2010)
5. Inglés-Romero, J.F., et al.: Towards the Automatic Generation of Self-Adaptive Robotics Software: An Experience Report. In: 20th IEEE Int'l Conf. on Collaboration Technologies and Infrastructures (WETICE), Paris, France, June 27-29, pp. 79–86 (2011)
6. EU 7FP DiVA Project, `http://www.ict-diva.eu`
7. Inglés-Romero, J.F., et al.: Dealing with Run-Time Variability in Service Robotics: Towards a DSL for Non-Functional Properties. In: 3rd Int'l Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2012), held in conjunction with (SIMPAR 2012), Tsukuba, Japan, November 5 (2012)
8. The E-puck website, `http://www.e-puck.org`
9. Xtext website, `http://www.eclipse.org/Xtext/`
10. Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. Computer 41(4), 93–95 (2008)
11. Fleurey, F., Solberg, A.: A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 606–621. Springer, Heidelberg (2009)
12. The DiaSuite Project website, `https://diasuite.inria.fr/`
13. The MUSIC Project website, `http://ist-music.berlios.de/site/index.html`
14. MOSKitt4SPL website, `http://www.pros.upv.es/m4spl/index.html`
15. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving executability into object-oriented meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
16. Blair, G.S., Bencomo, N., France, R.B.: Models@run.time. IEEE Computer 42(10), 22–27 (2009)
17. Romero, J.R., Rivera, J.E., Durán, F., Vallecillo, A.: Formal and tool support for model driven engineering with Maude. Journal of Object Technology 6(9) (2007)