Prototyping Component-Based Self-Adaptive Systems with Maude

Juan F. Inglés-Romero[1], Cristina Vicente-Chicote[1], Javier Troya[2], Antonio Vallecillo[2]

[1] Dpto. Tecnologías de la Información y Comunicaciones, E.T.S.I. de Telecomunicación, Universidad Politécnica de Cartagena, Edificio Antigones, 30202 Cartagena, Spain
{juanfran.ingles, cristina.vicente}@upct.es
[2] GISUM/Atenea Research Group. Universidad de Málaga, Spain
{javiertc, av}@lcc.uma.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 component-based self-adaptive systems so that they can be formally simulated and analyzed. In order to illustrate the benefits of using Maude in this context, a case study in the robotics domain is presented.

Keywords. Self-adaptation, component-based architecture, prototyping, Maude

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 (1) a means for creating and sharing common foundations, based on their experience in self-adaptive system design; and (2) rigorous tools for testing and assuring the correctness of the adaptive behavior of their systems. The latter 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. The rewriting logic of Maude is simple, yet very expressive. This gives Maude good representational capabilities as a semantic framework to formally represent a wide range of systems, including distributed and concurrent systems, network protocols, etc. 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, formally 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, if the Maude prototype is simple, detailed and efficient enough, it could be directly used as the final system implementation.

In this paper we present our experience in using Maude for prototyping component-based self-adaptive systems. The results of this work derive from the implementation of a case study in the robotics domain. This research continues our previous works on self-adaptive system design [4] and implementation [5] using the framework developed within the EU 7FP DiVA Project [6].

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 details the proposed case study. Section 4 describes the Maude specification for modeling the self-adaptation logic of the case study. Section 5 reports the lessons learned and, finally, section 6 concludes and presents some future research lines.

2 Related Works

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]. Some contributions provide a conceptual guidelines describing the different stages of self-adaptation [7], while others focus on the specification of design patterns for adaptive systems [8] [9]. Kramer et al. [10] propose a three-layer architecture for self-managed systems, and Oreizy et al. [11] present an infrastructure that simultaneously supports system adaptation and evolution. 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].

The field of formal methods is very broad and there is a vast literature describing their many applications in different domains. The remaining of this section will focus on formal approaches targeting specific aspects of self-adaptation.

In the area of Architectural Description Languages (ADLs) supporting system adaptation, Wermelinger and Fiadeiro [12] present an algebra for formally specifying runtime architecture reconfiguration. Canal et al. [13] propose the use of LEDA (an

ADL supporting inheritance and dynamic reconfiguration) to specify dynamic programs. LEDA is based on the π-calculus, a simple but powerful process algebra that allows to automatically deriving prototypes from the specification. At a higher level of abstraction, Zhang et al. [14] present a model-based approach for formally specifying the behavior of adaptive programs, starting from high-level requirements. The resulting models can be analyzed using model checking techniques and can be used to generate rapid prototypes from them. Aligned to the latter, the work by Sama et al. [15] proposes a model-checking approach for detecting faults caused either by erroneous adaptation logic, or by the asynchronous updating of the context information that leads to inconsistencies between the physical context and its internal representation in the application. This proposal relies on the formalism provided by the Finite-State Machine theory. Cansado et al. [16] propose a formal framework that supports behavioral adaptation and structural reconfiguration. This approach relies on the formalisms provided by Labeled Transition Systems and model checking for (1) reasoning about whether it is possible or not to reconfigure the system; and (2) to verify certain reconfiguration-related properties. Weyns et al. [17] present a rigorous specification of a reference model for self-adaptation (called FORMS), defined using the Z notation. FORMS aims to (1) establish a shared vocabulary of primitives that can be used to precisely define arbitrary complex self-adaptive systems; (2) enable engineers to precisely express their design choices and assess them; (3) allow for comparison and evaluation of different types of self-adaptive systems; and (4) lay the foundation for a systematic method of developing a catalog of architectural patterns. Bruni et al. [18] propose the use of Maude to demonstrate the feasibility of their conceptual framework in which adaptation revolves around control data. The authors use the formal toolset provided by Maude (in particular, the statistical model checker PVesta) for simulation and analysis. As a case study, they consider an example based on robot swarms equipped with obstacle-avoidance and self-assembly capabilities.

3 Case Study

In this section, we introduce a robotic case study designed to illustrate the benefits of using Maude for prototyping self-adaptive systems. Firstly, we describe the adaptation scenario. Then, we briefly present the infrastructure we used to implement the case study. Finally, we provide some details about the component-based software architecture designed to cope with self-adaptation in the case study.

3.1 Adaptation Scenario

The case study takes place in a room (simulated with Lego blocks) where a small robot moves around randomly avoiding obstacles. In order to improve this basic functionality in terms of safety, power consumption and efficiency, 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 (namely, 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 integer values 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 efficiency. 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 *efficiency*, the higher the velocity the shorter the time it takes to the robot to reach its goal position. Obviously, maximizing safety and efficiency, 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.

3.2 Case Study Implementation

As previously stated, we intend to use the versatility of rewrite theories and, in particular, of Maude for prototyping self-adaptive systems. However, the computational limitations of the experimental robot we selected as our target platform made it impossible for us to deploy the whole application in it. As a consequence we decided to adopt the remote processing schema illustrated in Figure 1.

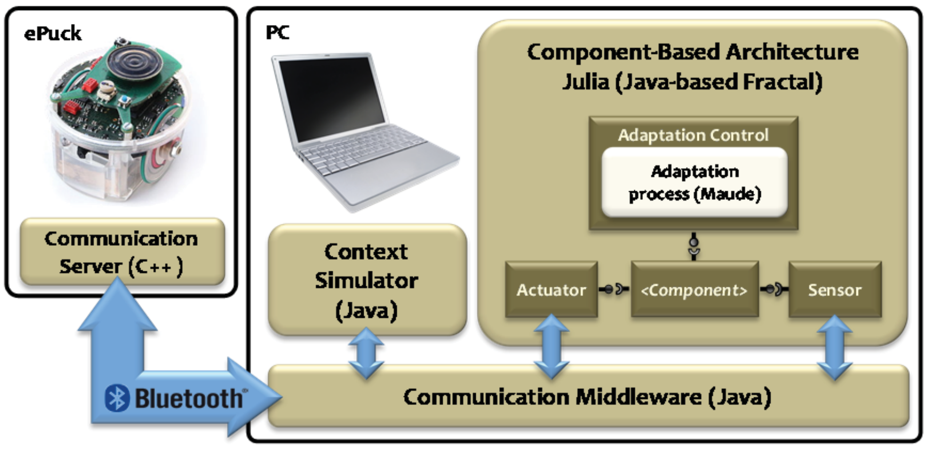
ePuck
PC
Component-Based Architecture
Julia (Java-based Fractal)
Adaptation Control
Adaptation process (Maude)
Communication Server (C++)
Context Simulator (Java)
Actuator
<Component>
Sensor
Bluetooth
Communication Middleware (Java)

Fig. 1. Deployment infrastructure: core devices, applications and components

We selected the E-puck robot [19] 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 the proposed self-adaptation strategy. The E-puck robot runs a server that communicates with a PC via Bluetooth. This server provides the PC with information about the robot sensor status. Besides, it executes the low-level commands it receives from the PC (e.g., to turn on/off the lights or the speakers, to move faster or slower, etc.). In turn, the PC runs three applications (see Figure 1), namely: (1) the self-adaptive system architecture, specifically developed for the proposed case study; (2) a generic context simulator; and (3) a generic communication middleware. The self-adaptive architecture developed for the case study was implemented using Julia: the Java reference implementation of the Fractal component model [20]. We selected Julia for its runtime reconfiguration capabilities, among other interesting features.

In order to make the PC applications as independent as possible from the selected robotic platform and from each other, we have developed a generic (case study independent) communication middleware. This middleware provides the PC applications with a standard interface to interact with the robot services as if they were local. Besides, it manages process concurrency and offers flexibility to transparently connect different applications, e.g., a virtual robot (instead of a real one), an application displaying execution statistics, etc. Related to this, we have developed a generic (case study independent) context simulator to emulate changes in (some of) the context variables. In particular, in our case study, we simulate the robot battery level as E-pucks do not have a battery sensor. Even if this sensor was available, the simulation of this context variable seems more practical than waiting for the battery to drain, and having to recharge it before running the next adaptation test.

3.3 Component-Based Software Architecture

The component-based software architecture developed for the case study is sketched in Figure 2. 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.

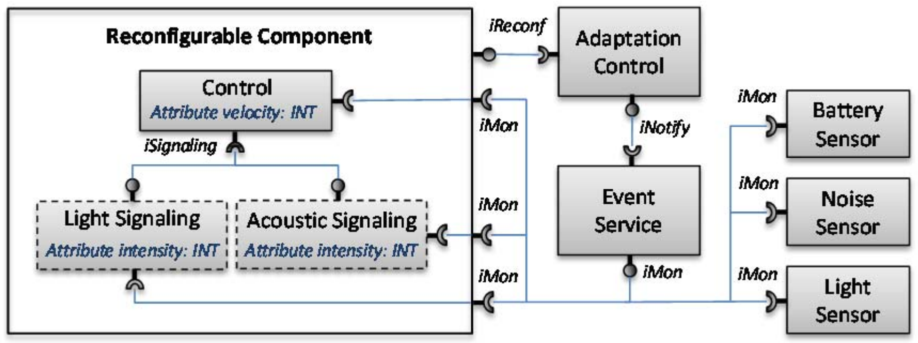

Fig. 2. Component-based software architecture for the case study

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. Next section details how the *Adaptation Control* component relies on Maude for executing this adaptation strategy.

4 Prototyping Self-Adaptation with Maude

This section describes the Maude specification of the self-adaptation strategy defined for the case study. Note that, for 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

Similarly to the systems described in [18], we have implemented our case study with Core Maude using an object-based programming approach. This allows us to model our 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 (e.g., `< 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 are described as operators that return a value of type `Msg`. Each message includes an identifier and a list of arguments (e.g., `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., evolve the system state.

We have used Maude for specifying both the main adaptation loop and a bridge aimed to enable the communication between Maude and the Java implementation of the *Adaptation Control* component. Regarding the later, the communication is performed through the standard I/O using the Domain Specific Language (DSL) summarized in Table 1. A *read-eval-print* loop has been implemented to handle this communication and to maintain the persistent state of the application.

Concerning the adaptation loop, as in most self-adaptive systems [2], it mainly 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, all these processes are carried out by the *Adaptation Control* component. Figure 3 outlines the steps of the algorithm that implements this adaptation loop. Each of these steps is further detailed in the following subsections.

Table 1. DSL for the interaction 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 Fractal 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

Prior to starting the adaptation loop, an initialization function needs to set up the context and the architecture models that will be used throughout the adaptation process. This function is labeled in Figure 3 as "*Init context and synchronize representation*".

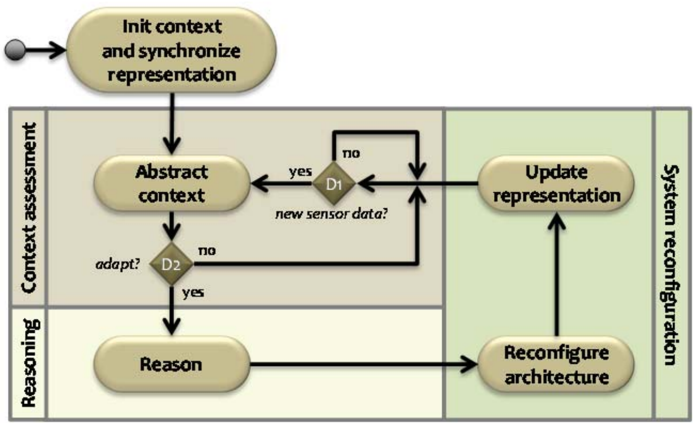

Fig. 3. Outline of the adaptation loop

The context model. To safely and efficiently adapt a running system, it is important to have a well-fitted model of its context. A context model 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. The proposed case study considers three low-level context variables, namely: the robot battery level and the ambient noise and light levels. The context model abstracts these low level variables by defining three new high-level variables: *batt* ∈ {*LOW*, *MEDIUM*, *FULL*}; *noise* ∈ {*NORMAL*, *NOISY*, *NOISIEST*}; and *light*, which is a Boolean. The function that computes the high-level context variables from the low-level ones (e.g., deciding when a *battery level* is LOW, MEDIUM or FULL) will be later detailed in section 4.3. A possible configuration of the context model could be as follows:

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

The architecture model. Similarly to the context model, maintaining an explicit reflection model that abstracts the actual running system is essential to efficiently decide on 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 case study system architecture (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>
```

Architecture and context model initialization. Before starting the adaptation loop, the context and the architecture models need to be created and synchronized for the first time to reflect the actual situation. Firstly, Maude creates and initializes a default architecture model containing all the components in the *Reconfigurable Component*. Secondly, when the Fractal architecture is created, all its components send (via *iMon*) their initial state and attribute values to *Event Service*. This component notifies (via *iNotify*) these values to *Adaptation Control* which, in turn, sends to Maude one or more `synchArch` message for each component. These messages trigger in Maude the `arch-synchronization` rewrite rule, which consumes the message and updates the state and attributes of the corresponding components in the architecture model. Messages from components not belonging to the *Reconfigurable Component* are discarded and produce no update. Finally, Maude waits until the *Adaptation Control* sends an `init` message with the initial context information. This message triggers the `init-context` rewrite rule, which (1) consumes the message; (2) creates and initializes the context model; and (3) creates a `reasoner` message that launches the reasoning process (later discussed) to assure that the system adapts (if necessary) to the initial conditions. This starts the adaptation loop and, from that moment on, no more `init` or `synchArch` messages are accepted (if they arrive, they are automatically discarded).

4.3 Context Assessment

The main functions of the *Context Assessment* process are: (1) to update the low-level context variables when Maude receives new sensor data (see the `battery`, `noise` and `light` commands in Table 1); (2) to update the high-level context model from the low-level values previously received; and (3) to launch the reasoning process in case the changes in the high-level context variables are significant enough. These three functions are represented in Figure 3 in the decision node *D1*, the operation "*Abstract context*" and the decision node *D2*, respectively.

In order to compute the high-level context model from the low-level sensor data we have implemented three rewrite rules (one for each context variable). These three rules share the same structure: the left-hand side term contains the current context object and the message with the new low-level context value (this message is consumed once the rule is executed) and the right-hand side term contains the abstraction and the activation functions detailed next.

The abstraction function. This function maps the low-level context (variables usually quantified as integer or float values) into the high-level context (variables usually modeled as enumerations or Booleans, providing a more qualitative than quantitative information). In the current implementation, we use fix thresholds (predefined at design-time) for segmenting the low-level context data. For instance, we consider the *batt* (high-level) to be *FULL* when the *battery* (low-level) value is greater than 80.

The activation function. This function determines how much the context must change to require a new adaptation step. If this function is not appropriately adjusted at design-time it may lead to a slow or ineffective adaptation or, what is worse, to an instable situation in which continuous reconfigurations are made to cope with every single small change in the context. In the current implementation, the adaptation process starts only if a high-level context variable changes. In this case, a `reasoner` message is created that triggers the rules performing the reasoning process, detailed next. This mechanism is more robust and stable than defining a fix variation range on a low-level context variable that, when exceeded, causes a new adaptation step.

4.4 Reasoning

The *Reasoning* function (see Figure 3) implements the core self-adaptation logic as it computes the best configuration possible for a given context, that is, it selects the set of abstract variants that jointly optimize the overall system performance (in our case the overall system safety, efficiency or power consumption). The abstract variants considered by Maude conform to the variability model described next.

The variability model. As detailed in section 3.1, the proposed case study considers three low-level variation points, namely: the signaling type (implies selecting the Light Signaling or the Acoustic Signaling component), the signaling intensity (integer ranging 0-100), and the robot velocity (integer ranging 0-100). The variability model abstracts these low-level variation points by defining three new high-level ones, namely: *signaling* ∈ {*LIGHT*, *ACOUSTIC*}; *intensity* ∈ {*LOW, MEDIUM, HIGH*}; and *velocity* ∈ {*SLOW*, *MEDIUM*, *FAST*}. The abstraction provided by this model, together with the one provided by the context and architecture models, significantly simplifies the reasoning process. As shown below, each abstract variant is modeled in Maude with an object containing the following attributes: *name*, *dimension* (ID of the high-level variation point the variant belongs to), *safety*, *consumption* and *efficiency* (impact of the variant in each property), *score* and *state*.

```
< 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 [21], which combines (1) the use of adaptation rules and (2) 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 model. The `non-available` rule sets the *state* of those variants that are inconsistent with the current context model (i.e., cannot be selected during the subsequent optimization process) as NON-AVAILABLE. For example, if the high-level *batt* context variable is not *FULL*, then the high-level variant *FAST* is marked as NON-AVAILABLE for the *velocity* variation point. 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 the high-level *light* context variable is *true*, then the high-level variant *ACOUSTIC* is marked as REQUIRED.

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 higher the *score* of a variant the better it fits the current context, i.e., the more likely to be selected as part of the new system configuration. The calculation of the *score* is based on: (1) the impact of each variant on the three system properties (ranging from 0: no impact to 5: very high impact); and (2) the importance of each property in the current context (also ranging from 0: no importance to 5: very high importance). The impact of each variant on the three properties is defined at design-time and stored in the abstract variant objects. The importance of each property depending on the context is also defined at design-time but, in this case, using Maude equations (e.g., there is an equation stating that if the *batt* is *LOW*, the importance of the *power consumption* property must be set to 5).

Finally, the `search-solution` rule finds the best possible system configuration for the current context, that is, the set of abstract variants that, together, obtain the highest score. This rule updates the *SystemConfig* object, which contains one attribute per high-level variation point. The following example shows the *SystemConfig* object resulting of a reasoning step in which the variants *LIGHT*, *MEDIUM* and *FAST* were respectively selected for the *signaling*, *intensity* and *velocity* variation points.

```
< c : SystemConfig | signaling : "light", intensity : "medium",
 velocity : "fast" >
```

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 model according to the decision made by the *Reasoning* function; and (2) to synchronize the architecture model with the runtime system architecture. To implement these functions, labeled in Figure 3 as "*Reconfigure architecture*" and "*Update representation*", we have implemented two Maude rewrite rules: `reconfigure` and `notification-when-pending`.

The `reconfigure` rule takes the *SystemConfig* object (updated by the *Reasoning* function) and the current architecture model (set of component objects) as its input, and produces a set of reconfiguration commands. In order to map the high-level variants, selected in the *SystemConfig* object, into the low-level ones, defined for the elements gathered in the *Reconfigurable Component*, we have defined a set of Maude equations (similar to those defining the relations among the system properties and the context variables). For instance, there is an equation that maps the *velocity* variant *FAST* with the value 90 for the attribute *velocity* of the *Control* component. When all the variants have been mapped, the rule generates the reconfiguration commands only 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 model and the one that has just been derived from the selected variants.

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. These notifications cause the architecture model to be updated to reflect the current situation. It is worth noting that we use an *Adaptation-System* object to register all the reconfiguration commands sent by Maude and not acknowledged yet with the corresponding notification. Context messages are discarded while this object is not empty. This prevents the execution of new adaptation loops while the architecture model and the running system are not completely synchronized.

5 Lessons Learned

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 activation function (see section 4.3) to make the adaptation stable, avoiding continuous system reconfigurations; (2) adjusting the design-time values that define the impact of the variants on the system properties, and the weight of these properties depending on the context (see section 4.4); and (3) establishing the right links between the high-level and the low-level context variables (see the abstraction function in section 4.3) and between the high-level and the low-level variants (see section 4.5).

Regarding simulation, Maude enables the execution of the system specification starting from any given state. This can be very useful for addressing the adjustments enumerated above. For instance, to test the Activation Function, we could measure the system reactivity by recording the number of reconfigurations performed per time unit, and relating this number with the context variation rate. However, this kind of analysis usually requires including some additional terms in the specification. This not only pollutes the prototype with analysis-specific code but also may influence the analysis results, as it might affect the overall performance of the prototype.

Concerning model checking, Maude provides the `search` command, which explores the reachable state space looking for a given configuration. This command is a simple, yet very useful method for checking invariants. For example, consider the following statement: the state of the LightSignaling and AcousticSignaling components cannot be simultaneously RUNNING. If we use the `search` command to find a counterexample and it returns an empty answer then we can assure that the statement is never violated. Maude also provides other tools supporting more complex model checking capabilities, e.g., a module for linear time temporal logic. It is worth noting that, in general, model checking processes are highly memory- and time-consuming.

Regarding the reusability of the Maude specification, it is worth noting that, although it is application-dependent, there some common structures (described in section 4) that could be easily reused. Also related to reusability, it is worth highlighting that the proposed design (see Figure 2) follows the separation of concerns principle, since the adaptation logic (implemented in Maude and embedded in the *Adaptation Control* component) is explicitly separated from “business” logic (in our case study implemented in the components gathered in the *Reconfigurable Component*). The benefits of such a decision are twofold: on the one hand, it reduces the complexity and improves the maintainability of the design and, on the other hand, it promotes the reuse and sharing of adaptation mechanisms among applications.

Some additional benefits of using Maude for prototyping self-adaptive systems are [3]: (1) the rapid development process and the reduced length of the programs, compared to other (traditional) programming languages. Prototyping in Maude has allowed us to focus on the development of the adaptation mechanisms without having to spend much time in implementation details; (2) the simplicity and versatility of using (a) a set of objects and messages to describe the system state; (b) a set of equations to model the system data; and (c) a set of concurrent rules to describe the system behavior; and (3) the performance of Maude prototypes is reasonably good. In fact, the performance of the prototype developed for our case study seems to be good enough for using it as part of a real self-adaptive system. The main limitation we found relates with the difficulty for debugging Maude programs, due to the concurrent nature of its rules and the scarcely legible traces it returns during the execution.

6 Conclusions and Future Work

This paper reports our experience in using Maude for prototyping, simulating and verifying component-based self-adaptive systems. In order to demonstrate the benefits (and also to assess the limitations) that Maude can bring in this field, we have developed a case study in the robotics domain that relies on a distributed processing schema. The robot adaptation logic, implemented in Maude, has been embedded (separate from the business logic) in one of the components of the Fractal-based implementation of the system. This component is fed with contextual information and controls the adaptation of the reconfigurable part of the architecture. In order to make the adaptation decisions efficient, the Maude specification works with abstract models of the context, the architecture and its variability. 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 the model-driven approach proposed by DiVA [6]. In this sense, our intention is to generate the Maude specification from the DiVA models describing the self-adaptive system design.

Acknowledgements

This work has been partially funded by the EXPLORE (MICINN, TIN2009-08572) and the MISSION (Fundación Séneca-CARM, 15374/PI/10) projects. Juan F. Inglés-Romero thanks Fundación Séneca-CARM for a research grant (Exp. 15561/FPI/10).

References

1. Cheng, B.H.C., et al.: Software engineering for self-adaptive systems: A research roadmap. In: Cheng, B.H.C., et al. (Eds.), Software Engineering for 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. Clavel, M., et al.: All About Maude. A High-Performance Logical Framework: how to specify, program and verify systems in rewriting logic. Springer-Verlag (2007). ISBN 978-3-540-71940-3
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), 3-8 October, Oslo, Norway (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'), pp. 79–86, 27-29 June, Paris, France (2011)
6. EU 7FP DiVA Project, `www.ict-diva.eu/`
7. Kephart, J., et al.: The Vision of Autonomic Computing. Computer, 36(1), 41–50 (2003)
8. Gomaa, H., Hussein, M.: Software Reconfiguration Patterns for Dynamic Evolution of Software Architectures. In: 4th Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 79–88, 12-15 June, Oslo, Norway (2004)
9. Ramirez, A.J., Cheng, B.H.C.: Design Patterns for Developing Dynamically Adaptive Systems. In: 2010 Int'l Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 49–58, 3-4 May, Cape Town, South Africa (2010)
10. Kramer, J., Magee, J.: A Rigorous Architectural Approach to Adaptive Software Engineering. Journal of Comp. Science and Technology, 24(2), 183–188 (2009)
11. Oreizy, P., et al.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems, 14(3), 54–62 (1999)
12. Wermelinger, M., Fiadeiro. J.L.: Algebraic software architecture reconfiguration. In: Nierstrasz, O., Lemoine, M. (Eds.), Software Engineering Conference (ESEC/FSE'99). LNCS, vol. 1687, pp. 393–409, Springer, Heidelberg (1999)
13. Canal, C., Pimentel, E., Troya, J.M.: Compatibility and inheritance in software architectures. In: Science of Computer Programming, 41(2), 105-138 (2001)
14. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: 28th Int'l Conf. on Software Engineering (ICSE), pp. 371–380, 20-28 May, China (2006)
15. Sama, M., Elbaum, S., Raimondi, F., Rosenblum, D.S., Wang, Z.: Context-Aware Adaptive Applications: Fault Patterns and Their Automated Identification. In: IEEE Trans. on Software Engineering, 36(5), 644–661 (2010)
16. Cansado, A., Canal, C., Salaün, G., Cubo, J.: A Formal Framework for Structural Reconfiguration of Components under Behavioural Adaptation. In: Electron. Notes Theor. Comput. Sci., 263, 95-110 (2010)
17. Weyns, D., Malek, S., Andersson, J.: FORMS: a formal reference model for self-adaptation. In: 7th International Conference on Autonomic Computing (ICAC), pp. 205–214, 7-11 June, Washington DC, USA (2010)
18. Bruni, R., et al.: Modelling and analyzing adaptive self-assembling strategies with Maude. In: 9th Int'l Workshop on Rewriting Logic and its Applications (WRLA), held in the context of the 2012 European Joint Conferences on Theory & Practice of Software (ETAPS), pp. 48-67, 24-25 March, Tallinn, Estonia (2012)
19. The E-puck website, `http://www.e-puck.org`
20. The Fractal Project, `http://fractal.ow2.org/`
21. Fleurey, F., Solberg, A. A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In: *12th Int'l Conf. on Model Driven Engineering Languages and Systems* (MODELS), pp. 606-621, 4-9 October, Denver, Colorado, USA (2009)