Coordinated Management of Hardware and Software Self-adaptivity

Onur Derin Alberto Ferrante Antonio Vincenzo Taddeo [*]

ALaRI, Faculty of Informatics, University of Lugano
Via G. Buffi, 13
6904, Lugano, Switzerland
Phone: +41-58.666.4709
Email: {derino, ferrante, taddeo}@alari.ch

Abstract

Self-adaptivity is the capability of a system to adapt itself dynamically to achieve its goals. Self-adaptive systems will be widely used in the future both to efficiently use system resources and to ease the management of complex systems. The frameworks for self-adaptivity developed so far usually concentrate either on self-adaptive software or on self-adaptive hardware, but not both.

In this paper we propose a model of self-adaptive systems and we describe how to manage self-adaptivity at all levels (both hardware and software) by means of a decentralized control algorithm. The key advantage of decentralized control is in the simplicity of the local controllers. Simulation results are provided to show the main characteristics of the model and to discuss it.

Key words: self-adaptivity, reconfigurable, autonomic, goal, architecture, model, application, hardware, software, run-time environment

1 Introduction

Self-adaptivity is the capability of a system to adapt itself dynamically to achieve its goals. Goals are specified by programmers or by users and define application requirements at high-level (i.e., as human readable requirements, such as, for example, throughput). By defining requirements self adaptive computational systems have the ability to adapt themselves to mutating internal and external conditions [20] without requesting any intervention of the user.

[*] Authors appear in alphabetical order.

Self-adaptation capabilities are used to implement autonomic and life-inspired systems: the increasing complexity of components and the difficulties of their integration are pushing the designers toward self-managing systems. These systems will have the ability to self-adapt and self-configure to provide the performance and the quality required [16] [14]. Self-adaptive devices can be utilized in pervasive systems to cope with mutating environmental conditions. For example, a portable device may be frequently moved from an office environment (where power and network plugs are available) to an external environment (where the device can only be battery operated and the network may be available in different wired or wireless forms). In this case the behavior of different hardware and/or software components of the system needs to be adapted to the new conditions (e.g., to reduce power consumption). Furthermore, different functionality may be proposed to the user in the new environment.

Self-adaptation can be supported by the software (in the applications, in the operating system, or both) or by the hardware. In a system both hardware and software components may be self-adaptive and each part may or may not have the knowledge of the possible self-adaptivity of other parts. For example, specific functions can be mapped on the hardware at run-time to optimize the execution of certain applications. The management of hardware or software self-adaptivity is a complex task as different conflicting goals may need to be considered. Jointly managing hardware and software self-adaptation is of course even more complex. Furthermore, a unified self-adaptation controller, would impose a close cooperation between hardware and software. Thus, it would be difficult to provide portable self-adaptive applications. A self-adaptive system lives in an *environment* which can be defined as the complementary set of the self-adaptive system (i.e., all the things surrounding the system). Self-adaptation can be triggered by different events, like changes in the environment, changes in the applications to be executed, or changes in the system operational conditions (e.g., a battery operated system detects a change in the battery status, or a component that becomes faulty). Self-adaptivity not only provides functional and operational benefits, but it also allows for self-healing. In fact, a faulty hardware or software component will be automatically replaced (if replacements are available) to keep satisfying the application goals.

This paper provides a comprehensive approach to self-adaptivity management. The target of this a approach is the model of self-adaptive system described in Section 3. This model, as shown in Section 2, is general enough to include many of the self-adaptive systems already discussed in the literature and to consider a comprehensive approach to self-adaptivity both in hardware and in software components. The model implies a subdivision of the self-adaptive system in different macro-layers named *software* and *hardware*. The software macro-level is further divided into two levels: the former is for software applications; the latter is for providing an uniform interface between hardware and software applications. The framework aims at providing a control mechanism that is both simple and efficient. The simplicity is reached by using separation of concerns and decentralization of control. The efficiency is reached by adopting a coordination mechanism, called *recommendation system*, among the controllers. The paradigm used for self-adaptation management at all levels is the monitor-controller-adapter (MCA) one. In this paradigm each component is endorsed a different role: the first one monitors the parameters of interest, the second one decides on possible self-adaptations, and the third one enforces them. Some simulations are presented in the paper to show the properties of the recommendation system and to provide a ground for discussing the model.

The remaining part of the paper is organized as follows: Section 2 gives an overview of related research about self-adaptivity in hardware and in software; Section 3 presents our model; in Section 4 the results of model simulations are shown and discussed.

2 Related Work

Looking at previous works, it is possible to see some similarities and differences. As the most prevalent technique for self-adaptation, we see the use of the monitor-control-adapter paradigm. In an effort to classify the existing solutions, we identify four main design decisions: *Adaptation coverage* is defined by the parts of the system affected by adaptations. Majorly it may consist of hardware, software, part of a distributed system (through adaptive middleware) or any combination of them. *Separation of concerns* is a design principle that decouples the functionality of the system from the implementation of its adaptation. *Adaptation management* is the decision making process on the evolution of the system. *Adaptation requirements specification* is the form of describing the non-functional requirements of the system. In the remaining part of this section we provide an overview of publications related to self-adaptivity, classified according to their most prominent characteristics in the view of these categories.

2.1 Adaptation coverage

A number of works addresses self-adaptivity in software; the simplest approach adopted is to manage adaptation in the application code. Although this approach enables the development of ad-hoc solutions for specific adaptation problems, it is clearly not flexible enough to support a wide range of adaptations. The use of an *architecture-based approach* eases self-adaptivity:

the system is viewed as a composition of concurrent *components* interconnected by *connectors*. A comprehensive adaptation methodology is presented in [20]. The authors propose an evolution and adaptation management infrastructure. The evolution management process adapts the architecture and the topology of the components and of the system; the adaptation management process gathers information from the operating environment, evaluates the observations with respect to the system requirements, plans and deploys adaptation changes. Moreover the need of composable components is underlined. Another work describing a component-based architectural approach is presented in [9]. The authors propose a framework that is both reusable, to cope with a large set of systems, and that supports mechanisms to specialize the infrastructure for specific cases. To achieve such objectives, the framework is divided into two logical parts: an adaptation infrastructure and a system specific adaptation model. The former provides common functionality that is reusable across different self-adaptive systems; the latter is specific to a certain system and it is used to tailor the entire framework for it. In [2] an architecture description language named ArchWare is modified to support self-adaptation. Feedback obtained by means of software probes is used to control software self-adaptations.

A formal approach to the design of adaptive software is introduced in [23] where an architecture-based approach is not considered. In particular, the adaptation is conceived as a state transition from a source program to a target program inside of a suitable set of adaptation states. Each adaptive software is represented by a state machine, where each state exhibits a different behavior and operates in a certain domain. To guarantee system integrity and consistency, local and global properties (requirements, constraints, and invariants) that should be satisfied by an adaptive program for every state change are introduced.

Self-adaptation has been addressed in distributed systems for the management of quality of service (QoS) mostly through adaptive middleware mechanisms and custom adaptation management protocols. A control theory approach to QoS is proposed in [3], [4]. In the first paper, the authors introduce a passive adaptation task mechanism, located in the middleware level to support application-specific adaptation. Basically, passive adaptations can be viewed as transformations of the data input stream incoming into a task (e.g. a software component) to fit a requires QoS. The middleware performs adaptation between applications and the transport layer based on certain QoS metrics. In the second paper, the same authors extend and improve such a mechanism to balance and support both application-specific adaptations and system-wide requirements, such as stability and fairness. In [10] a *mirror-based reflection* approach for self-adaptivity is proposed. By definition, a reflective system is able to perform computations about itself; moreover, it provides introspection and control through a reflective interface. By applying this reflective mechanism to software components, the middleware can perform self-adaptation by using the reflective interface of each component. Adaptation behavior, architecture and implementation of a component can be specialized to fit a specific context by annotating each implementation with quality of service metrics. Therefore, the middleware uses such quality of service metrics to trigger the adaptation of components.

In [1] an approach for managing quality of service grid systems is presented: system resources are managed in an adaptive way both to satisfy the quality of service requirements and to use the resources efficiently. Thus, self-adaptivity is in the process management software included in the operating system of each node. In [8] a similar approach is used to provide network quality of service. The self-adaptivity is included in the network routers and it allows the system to efficiently use the network bandwidth. In both the aforementioned examples the control system is composed by sensors, a set of decision procedures, and actuators. [12] presents two protocols for QoS adaptation which allow to recover from QoS violations by changing the distribution of QoS levels assigned to the network components in distributed multimedia applications.

Some works related to hardware self-adaptation have also been proposed. In [7] a self-adaptive hardware architecture is presented; this architecture provides self-configuration, self-repair and/or fault tolerance capabilities by means of self-placement and self-routing. In [5] a self-adaptive embedded processor is described. This processor is able to deploy different special instructions at run-time; the decision on which special instructions to deploy and when, is based on their monitored usage. A compile-time analysis of the applications is performed to reduce run-time overhead: the information extracted from this analysis is used to forecast the kind of instructions that will be used by the applications in the immediate future. Thus, self-adaptation can happen without introducing delays in the computation.

In our work we propose a model of a system that supports both hardware and software self-adaptivity. None of the previous works describes a comprehensive approach to self-adaptivity considering both self-adaptive hardware and software. Our model applies to a stand-alone system with both adaptable hardware and software. Such a system can be a component of a distributed system. However we don't address management of self-adaptivity for distributed systems. The mechanisms for self-adaptivity that we describe both for hardware and for software are to be classified as architecture-based [20] [9] [15] [18].

The approaches described in [7] and in [5] for hardware can be easily adopted in our hardware layer. Though to obtain the best possible performances, these approaches should be modified to include the recommendation system.

2.2 Separation of concerns

Separation of concerns between the regular system functionality and the adaptation processes is about putting different concerns into different components that will address them independently; this approach, even if not essential for self-adaptivity, is very important as it offers benefits in terms of generality, level of abstraction, integrated approach, and scalability. In [17] a vision of architecture-based self-adaptation is provided and a reference software architecture is proposed.

In [15] another architecture for software self-adaptivity is presented; one of its main goals is again separation of concerns. Thus, a *ground-level* that includes baseline processing and a *supervisory-level* that is responsible for adaptation and reconfiguration are considered. The former provides components that are highly optimized for specific situations; the latter select the most optimal components for the different situations. The adoption of the supervisory-level enables the system to provide flexibility and robustness. [21] implements a QoS management framework in a distributed system where adaptation strategies are separated from the core functionality by means of aspect languages and an encapsulation model for packaging adaptive behaviors.

One of the key concepts used in our paper is separation of concerns; this concept is adopted in the two works described above as well as in [9] and [23].

2.3 Adaptation management

Most of the approaches proposed in the literature use a centralized controller for self-adaptation. For example, [18] proposes a centralized controller based on constraint-guided design space exploration. The proposed approach is to use models to represent the different points in the design space of the application. The design space is composed of different software component alternatives. The basic idea is to create multiple-aspect models of the design points at design time. These models, along with system constraints, are then embedded into the run-time system and used for self-adaptivity decisions. Each constraint can be associated with one or more values that are continuously measured at run-time. Whenever one of these values crosses the threshold associated with it, the controller is triggered and the constrained design space exploration starts.

A different approach is to use decentralized controllers instead of a centralized one. This idea is mentioned in [22]; its main goal is to propose a software architecture that enables applications to be self-tuning and persistent. The work relies on strictly defined and controlled layering of policies and mechanisms, and on the complete control of all layers. Layer coordination is also utilized to obtain a stable behavior of the software. [21] uses a mix of centralized and localized QoS management in a distributed real-time system setting. Central control drives the QoS management via policies throughout the network whereas local control is guided by the contract attached to the network component.

In our work self-adaptivity is managed by means of a decentralized mechanism to simplify the local controllers. However this requires some coordination among the local controllers as also explained in [22]: we solved this problem through the recommendation system. In our approach the MCA paradigm similar to the one proposed in [9, Figure 1] is adopted at all levels for managing self-adaptation. This scheme is based on the feedback coming from probes as explained also in [2]. In our work we extended this mechanism to the control of the whole hardware/software system.

2.4 Adaptation Requirements Specification

Adaptation requirements have been specified differently in various works. In [18] they are specified as constraints by Object Constraint Language (OCL); in [21] they are expressed as policies via rule-based contracts. In [13] adaptation requirements are defined as constraints in a custom requirement description language (RDL). In [6] the authors introduce a method to specify adaptation requirements by means of goals. Goals are represented by using a graphical language named KAOS; by using this language a full goal-oriented specification of an adaptive system can be drawn.

In our work we propose a goal specification interface based on XML. Goals are specified as human readable requirements for the applications.

3 Model of Self Adaptive Systems

The design of self-adaptive systems is challenging due to the great number of variables to consider. For this reason, separation of concerns as well as an architecture-based approach have been adopted in several scientific publications. In this section we describe a model that is based on the same concepts and that provides the capability of managing self-adaptive software coupled with self-adaptive hardware, yet providing software portability. In the first part of this section we provide an analysis of self-adaptivity requirements of the different system components; we then present our model and we show how this model satisfies the requirements. The model has been developed to be general, thus, no reference to any specific implementation is made.

3.1 Self-adaptation Design

At software level self-adaptivity depends on events related to the environment (i.e., on events that are external to the system). Software can be self adaptive without necessarily relying on specific support mechanisms provided by the system (self-adaptation embedded in the source code). For example, if a network congestion is detected by the software, a compression algorithm can be activated on the data sent over the network; this self-adaptation mechanism can be entirely embedded in the software application. When software is considered, a number of possibilities for self-adaptation are available: run-time and dynamic change of the application goals (i.e., the application changes its high-level requirements for the system), adaptation based on selection of different behaviors (i.e., a different implementation of the same algorithm is selected), and intra-algorithm adaptation (i.e., some of the parameters of the considered algorithm implementations are changed at run-time). All of these self-adaptation mechanisms could be directly implemented in the software application, even though the first method requires support from the system to be effective. As shown in [9] an enabling technology for efficient adaptation is a component-based approach. The duty of managing adaptation (i.e., choosing one component among the others) is delegated to specialized software components, thus providing separation of concerns. For this purpose we require:

- an explicit specification of application goals along with a mechanism to change goals at run-time;
- the ability to modify the application components and their interconnections to achieve the expected results;
- the software components to support different working configurations (interchangeable at run-time).

At hardware level there are two possible kinds of self-adaptation: structural (i.e., change in the functional units or in the interconnections) and on parameters (i.e., hardware parameters – such as frequency – are changed run-time). Also in this case a component-based approach can be used to ease structural self-adaptation. Self-adaptive hardware can either manage adaptations internally or it can delegate (partly or entirely) this management to the software layer. For providing internal self-adaptation, the hardware needs to be able to change its configuration in a transparent way with respect to the software layer. Whenever software support is required for self-adaptivity, the hardware must notify to the software its reconfiguration capabilities. In both cases, the hardware may provide some information on the application execution and on the parameters that can be monitored and/or directly controlled by the software layer.

As discussed in Section 2.1, different kinds of hardware architectures may be utilized in a self-adaptive system. Although software developers should be enabled to write applications without necessarily knowing the structure of the underlying hardware and the mechanisms used for self-adaptation. In fact, the management of all of these details would make the job of the programmer too complex, it would break portability of applications, and it would remove any convenience in using self-adaptive systems.

3.2 The Model

The model that we propose here has been conceived taking into account the points discussed in the previous section; it was designed with the following assumptions in mind:

- adaptation is performed by *algorithm selection*: selection of the best algorithm implementation in a fixed set of available implementations, based on the observation of the operating environment [20];
- focusing on the adaptation management and assuming that the evolution management (topology and components architecture) is performed somehow at middleware layer;
- complete separation of goals among different levels (i.e., a goal that is managed at software level is not managed also at hardware level).

The last point is a restrictive condition that may be relaxed in future versions of the model.

The model is composed by a hardware and a software level, plus an intermediate level named Run Time Environment (RTE). The main purpose of RTE is to provide a standard interface between software and hardware. These functionalities are similar to the ones provided by other run-time environments such as, for example, the Java Runtime Environment [11]. The high-level architecture of the proposed model is shown in Figure 1. The RTE and the software level are grouped together as a macro-level (*software macro-level*). To implement the middleware functionality for a self-adaptive system, the RTE must be able to handle both hardware and software self-adaptivity. Thus, the RTE must provide an adaptation management framework to monitor, model, control, and adapt each software application. Furthermore, it must manage the available hardware resources by means of proper hardware interfaces.

Software macro-level and hardware level will monitor the applications being executed and self-adapt to reach their goals (whenever it is possible with the available system resources). Each level will check the results provided by lower

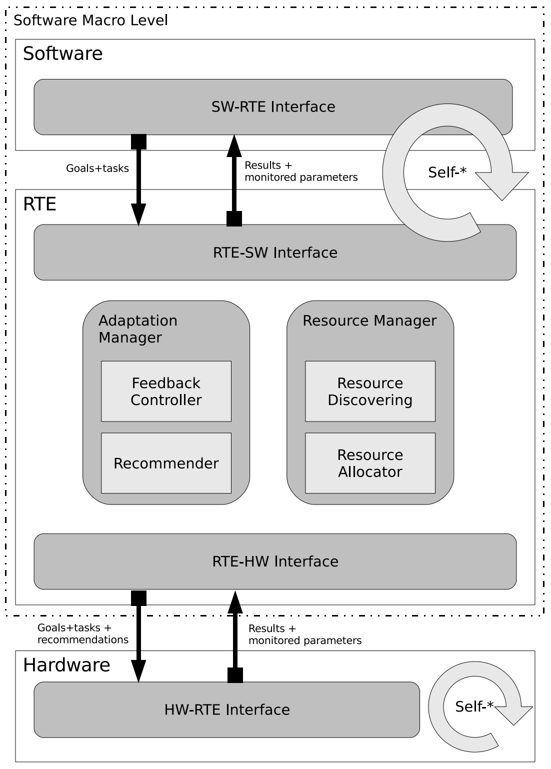

Fig. 1. Model of self-adaptive systems

levels along with their timing to check whether the required goals have been met. By this mechanism each macro-level is the only responsible for its own goals; goals propagate with a waterfall mechanism from the software level to the hardware level. The software macro-level self-adapts by choosing different implementations of the algorithms that are being executed or by changing their parameters. The hardware level has the capability to self-adapt by changing both hardware parameters (e.g., the clock frequency) and the hardware architecture to satisfy the goals imposed by either the software macro-level or at design time (e.g., temperature thresholds). Decisions on reconfigurations are anyway made locally at each level.

Figure 1 also provides a general view of the RTE architecture. The interfaces with hardware and with software embody two of the most important capabilities of the RTE: the propagation of goals from the applications down to the hardware layer (if required) and the management of adaptivity. In fact, to accomplish the main adaptation features, RTE must at least provide the following capabilities: interface with hardware and software, resource management, and adaptation management. A description of the RTE blocks that provide these capabilities is presented in the following sections.

3.2.1 RTE Interfaces

The *RTE-SW* and the *RTE-HW* are the modules that provide the interfaces of the RTE with the software and with the hardware, respectively. By replacing the *RTE-HW* component, the RTE will be able to deal with different kinds of hardware by always providing the same interface to software applications. The *RTE-SW* provides a standard interface for self-adaptive applications. The *RTE-SW* interface differs from the interfaces currently provided in normal operating systems in that it also provides a proper way to specify application goals and alternative implementations of the algorithms used within the application. In the interface a number of standard goals that can be provided by the applications, currently throughput and latency, is specified. Other possible goals must be translated into these ones by application programmers. RTE may also enable the software application to directly monitor some system parameters. A list of the available parameters is published in the interface. Even though the monitoring of the goals is performed by the RTE, the application is not allowed to perform any direct reconfiguration of the system.

The *RTE-HW* interface is used to manage the execution of operations by the hardware and to collect the results of these executions. This interface may also allow the RTE to send *adaptation recommendations* to the hardware. The capability of managing hardware functional reconfiguration may also be provided. This is useful for hardware modules that are not able to self-manage.

3.2.2 RTE Resource Manager

The *Resource Manager* is a fundamental component of the RTE; it is responsible for discovering the available hardware resources, increasing/decreasing parallelism (given that enough resources are available), and changing the distribution of tasks over the available resources.

3.2.3 RTE Adaptation Manager and Self-Adaptation

The two levels contained in the software macro-level of Figure 1 (software and RTE), strictly cooperate to provide self-adaptation. As mentioned before, the software level provides the RTE with a list of alternative software implementations of parts of the applications. The RTE monitors different parameters and tries to satisfy the goals by selecting a proper software implementation or by reconfiguring the underlying hardware, whenever functional reconfiguration of the hardware is available to the RTE. The *Adaptation Manager* is the component that actually manages such an adaptation mechanism.

The Adaptation Manager receives as input the applications to be executed along with their goals from the software layer and it uses a *MCA paradigm*

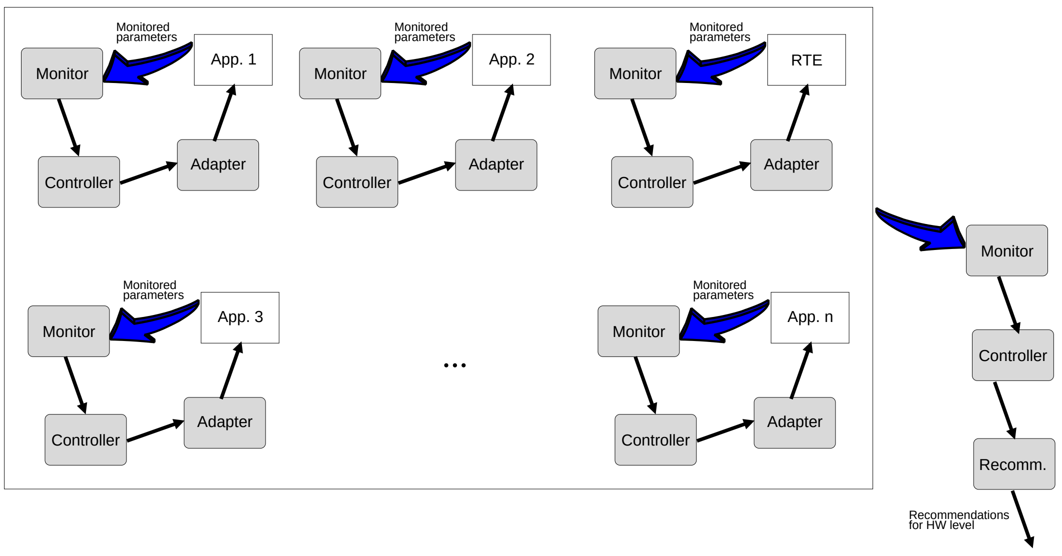

Fig. 2. Adaptation scheme at software macro-level.

to handle the software self-adaptation. The Adapter connects the *monitored variables space* with the *adaptation space*. The monitored variables space is generated from the goals that, in turn, are translated into a combination of monitorable variables. The adaptation space is obtained from the description of the possible alternative algorithm implementations provided by the applications. The MCA loop uses the monitored variables space to observe the operating environment and to awake the controller for an adaptation transition. An adaptation transition is performed every time a goal is not reached. Therefore, we can assert that once a goal is specified using the available monitored variables, such a goal is used to build arcs in the adaptation graph, where each node represents a specific component implementation. The MCA feedback loop is the goal-driven mechanism to move in the adaptation graph.

Figure 2 shows the MCA self-adaptation scheme proposed for the software macro-level. Each application is monitored by a specific monitor; for each application a controller receives information from its own monitor and provides proper control signals to the adapter. The latter reconfigures the application in a proper way when required. A RTE-level MCA scheme is also adopted to check that system goals, and not only the application ones, are reached. This system monitor checks the system behavior and sends proper information to the system recommendation unit that, in turn, sends proper reconfiguration recommendations to the hardware level.

The RTE itself is also self-adaptive; in fact, even the *controller* can be changed depending on the system conditions (e.g., different algorithms can be used to manage self-adaptation depending on the number of running applications).

As mentioned before, self-adaptivity is locally managed at each macro-level. Unfortunately, a completely decentralized management of self-adaptation may lead to a non optimal utilization of the system resources and to the inability to satisfy goals even when system resources would be sufficient. To solve this problem a coordination mechanism between the software and the hardware self-adaptivity controllers has been introduced. This mechanism is called the *recommendation system.* Whenever the RTE-level global monitor senses that a global goal is not reached and cannot be reached by means of software adaptations, it notifies the hardware (i.e., it sends a recommendation). The hardware receives this notification and may use it for future self-adaptations, or not. A feedback on the recommendation may be given to the RTE. The recommendation system activates some state transitions that may not normally be used by the hardware controller. This helps the system to move toward other possible design points which may be the optimal ones.

3.3 Goal Management Interface Proposal

In this section we propose a possible interface for specifying application goals and alternative algorithm implementations. This is a very important part of the system as all the self-adaptation process depends on goals. Goals are defined by means of a XML file, such as the one shown in Figure 3. The goal *name* identifies the kind of goal to be considered; as mentioned before, it can be *throughput* or *latency.* The *weight* field is used to assign a proper priority to the different goals and it is in the $0-1$ range; the sum of all the weights for the different goals must be 1 as shown in Equation 1. Threshold *condition* can be *gt* or *lt* that stand for greater than and less than, respectively. *Threshold* is used to specify the threshold for the given goal. There is an alternate syntax, specified by the keyword *MINMAX*, for specifying more generic goals requirements, such as "maximize throughput". In the *MINMAX* the *type* field can assume the values *min* or *max* to specify if a goal has to be minimized or maximized, respectively. Thresholds are expressed in *bit/s* in throughput goals and in *ns* in latency goals.

In the RTE, overall goal achievement (G) is calculated as a weighted sum of all application goal achievements by using the following formula:

$$G = \sum_{i=1}^{n} w_i \times g_i, \quad where \quad \sum_{i=1}^{n} w_i = 1, \quad w_i > 0 \tag{1}$$

n is the number of application goals, g_i signifies the degree of achievement for the goal i, and w_i is the weight of the goal i. G is always greater than 0 and lower than 1. The closer G is to 1, the closer the application is to satisfy its goals. $G = 1$ means that all the goals are satisfied. The g_i values can be determined in two different ways, depending on the kind of goal considered. For a goal g_i a threshold value (m_i^T) of the corresponding monitored variable is provided by the application through the goal management interface. At any

```
<GOAL name="goal_name", weight="goal_weight">
  <THRESHOLD condition ="gt | lt" threshold="value"/> | <MINMAX type="min | max"/>
</GOAL>
```

Fig. 3. XML for the specification of application goals.

```
<ALGORITHM id="app_id">
  <IMPLEMENTATION id="impl_id">
    <PARAMETER name="param_name">
      <RANGE type="incremental | set" range="[start, end, step] | [v_1, ..., v_n]"/>
    </PARAMETER>
    <GOAL_EFFECT position="goal_position"/>
  </IMPLEMENTATION>
</ALGORITHM>
```

Fig. 4. XML for the specification of alternative algorithm implementations.

instant, the actual monitored value (m_i) is obtained through the monitoring capabilities of the RTE. The achievement level of the goal can then be calculated using these two values. If the threshold represents a lower bound, the following formula is used to compute g_i:

$$g_i = \begin{cases} m_i/m_i^T, \; m_i < m_i^T \\ 1, \; m_i \geq m_i^T \end{cases} \tag{2}$$

Whenever the threshold, instead, provides an upper bound, the formula for g_i becomes:

$$g_i = \begin{cases} m_i^T/m_i, \; m_i > m_i^T \\ 1, \; m_i \leq m_i^T \end{cases} \tag{3}$$

As mentioned before, some goals may be of the kind "maximize/minimize the monitored value m_i" (*MINMAX* goals); in this case, $g_i = m_i/MAX(m_i)$ for goals that must be maximized and $g_i = MIN(m_i)/m_i$ for goals that must be minimized where MIN and MAX functions give the minimum and maximum values of the monitored variable ever encountered during the lifetime of the application.

Different algorithm implementations are specified by means of the interface defined in Figure 4. In this XML file we can define different alternative implementations, their parameters, and a range of values for the parameters. Implementations are sorted in the file according to increasing values of the cost function with respect to a reference architecture. This sorting can be done by using a technique similar to the one used in [18] in which a performance model of different design points is created at design time. Unfortunately, in our case we cannot know in advance which hardware architecture will be used. Thus, the sorting is done on a reference architecture and will be updated by the RTE at run-time to reflect the real performance of the different algorithm implementations on the considered system. In the figure, *id* in the *ALGORITHM*

```
<GOAL name="throughput" weight="0.8">
  <THRESHOLD condition="gt"
             threshold="1024"/>
</GOAL>

<GOAL name="latency" weight="0.2">
  <MINMAX type="min"/>
</GOAL>
```

(a)

```
<ALGORITHM id="mpeg4_encoding">
  <IMPLEMENTATION id="1">
    <PARAMETER name="quality">
      <RANGE type="incremental"
             range="[1, 100, 1]"/>
    </PARAMETER>
    <GOAL_EFFECT position="2"/>
  </IMPLEMENTATION>

  <IMPLEMENTATION id="2">
    <PARAMETER name="quality">
      <RANGE type="set"
             range="[10, 50, 75, 90, 100]"/>
    </PARAMETER>
    <GOAL_EFFECT position="1" />
  </IMPLEMENTATION>
</ALGORITHM>
```

(b)

Fig. 5. Example of application specification. Specification of application goals in (a) and alternative implementations of an algorithm in (b)

tag defines the identifier of the algorithm; *id* in the *IMPLEMENTATION* tag defines the identifier of the implementation considered. The *PARAMETER* tag is used to define the parameters of the specific implementation; each parameter will be named through the *name* attribute. The *RANGE* tag allows the programmer to specify a range for the considered parameter; *type* defines the type of range considered: it can be either *incremental* (i.e., a range is defined) or *set* (i.e., a list of values is provided). Whenever the range type is *incremental*, the range must be provided as $[start, end, step]$; in the other case, a list of values ($[v_1, v_2, ..., v_n]$) must be given. *GOAL_EFFECT* provides the sorting with respect to the goal function to be maximized; this sorting is provided through the *position* parameter.

Figure 5 shows an example of application goal specification and of alternative algorithm implementation specification; regarding the goals, a minimum throughput of 1kbit/s is specified. This is the most important requirement with weight 0.8. The second goal is specified on latency, that is required to be minimized. From the implementation alternatives stand point, two alternative implementations for the MPEG4 encoding algorithm are specified; the second is the one exhibiting best performance on the reference system with respect to the mix of goals considered. Though, the first implementation provides the ability to specify different qualities for the image, ranging from 1 to 100% in increments of 1; the second implementation only allows for five different quality values (10%, 50%, 75%, 90%, 100%) to be utilized. Thus, the adaptation space is defined by 105 points, corresponding to the different configurations given by any algorithm implementation and parameter combination. The variable space is given only by throughput and latency.

3.4 Discussion on the Model

With our model, the general problem of designing self-adaptive systems has been partitioned into several quasi-independent tasks such as creating adaptable systems, creating monitorable systems, and managing adaptation to meet goals. Furthermore, each of these tasks are simplified by separating the system into three levels such as software, run-time environment and hardware. The focus in this paper has been more on the management of adaptation given that run-time environment and hardware are equipped with adaptation and monitoring capabilities. On such a model, management of self-adaptation reduces to specification of goals, distribution of goals onto RTE and HW levels, and designing controllers for these levels. With such a decentralized controller approach, the control algorithms at each level are expected to be simpler as they would be responsible for achieving smaller number of possibly conflicting goals due to their distribution among different levels. Unfortunately, this mechanism may introduce convergence problems; furthermore, just sub-optimal solutions can be achieved by using this scheme. To avoid both of these problems, we introduced the mechanism of recommendations. This mechanism provides coordination among the controllers at different levels by also preserving the advantages of local self-adaptivity. In the most general sense, it can be thought of as a mechanism to redistribute the goals among levels at run-time.

A number of simulations has been performed to help understand better the advantages and disadvantages of our approach. The results of these simulations are presented and discussed in Section 4

4 Model Simulation

The model presented in Section 3 has been described in SystemC and simulated. The SystemC language [19] was selected to describe our model as it easily allows to co-simulate concurrently running HW/SW components as the ones we are proposing in our model. The main purpose of our simulations is to show that the proposed model for self-adaptation works. Furthermore, the simulations provide a better view on the model properties.

In the SystemC model we analyzed an example in which we imposed two goals on the system: the power consumption should be less than a threshold m_P^T and the throughput should be greater than a threshold m_{Th}^T. Figure 6 shows the simulation model that incorporates the MCA paradigm. Our focus in this simulation is to analyze the coordination between the controllers at different levels. Therefore we combined the adaptor and the monitor modules with the system module. The adaptable parameters are the clock frequency

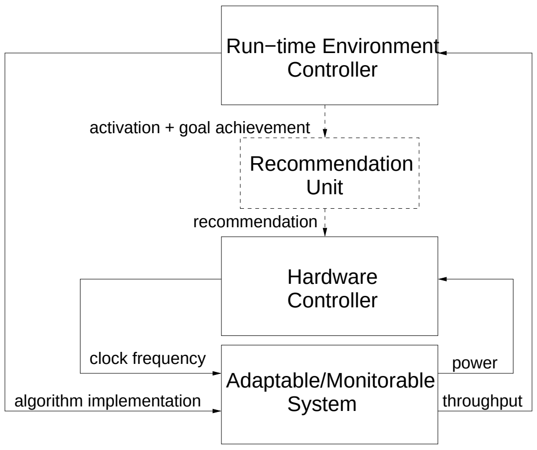

Fig. 6. Simulation model

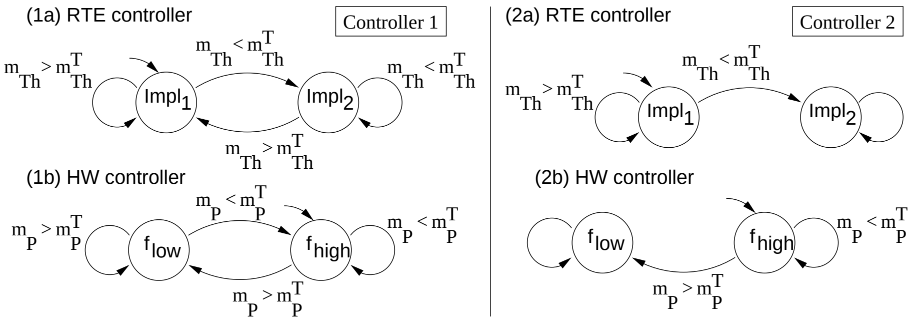

Fig. 7. Two example control algorithms with their RTE and HW level controllers

and the implementation of the algorithm that is being run by the system. The hardware supports switching between two frequencies (f_low and f_high). Adaptation space for the RTE is given by two different implementations of the application ($Impl_1$ and $Impl_2$). The parameters that can be monitored are power dissipation (m_P) and the throughput (m_{Th}) of the system. RTE controller and HW controller have been assigned to meet the throughput and power goals, respectively. It is well-known that running the hardware at a higher frequency increases power consumption. Moreover it is given that $Impl_2$ yields higher throughput than $Impl_1$ on a reference architecture. The use of self-adaptivity eases the achievement of the desired performance in complex systems in which some of the system parameters cannot be decided at design time. However, for simulation purposes, the system behavior has been set to produce values for monitored parameters in accordance with the given adaptation parameters. Moreover the controllers have been considered to be activated in defined periods.

As the first simulation, the model shown in Figure 6 has been used excluding the recommendation unit (shown with dashes). The controller used is *Controller 1* consisting of the RTE controller and the HW controller as shown in Figure 7.1a and 7.1b. The adaptation behavior obtained by means of *Controller 1* is given in Figure 8 showing the change of throughput and power

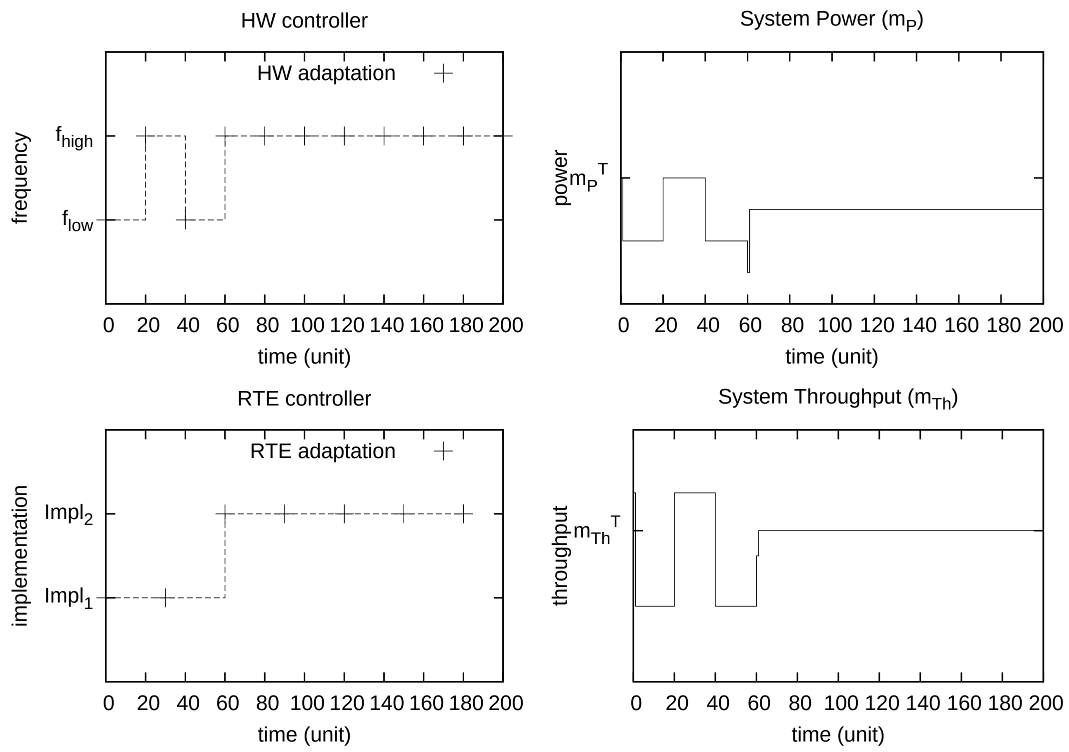

Fig. 8. Simulation results with *Controller 1* given in Figure 7. ($T_{system} = 1 timeunit, T_{RTEController} = 30 timeunits, T_{HWController} = 20 timeunits$)

values of the system in relation to the adaptation decisions taken by the controllers. The adaptation is completed at time 60 with a success by reaching high enough throughput and low enough power consumption. This shows that it is possible to create decentralized controllers that would reach the goals for all levels. However, designing such controllers is not intuitive: for example, in the controller shown in Figure 7.1b a state change from low to high frequency is performed even if the power goal is achieved in the low frequency state. Moreover the selection of the periods of the controllers may be tricky. Taking the period of the RTE controller as 40 instead of 30 time units results in a live-lock situation in which both controllers decide to switch states at the same time continuously while goals never get reached and adaptation never stops.

A more intuitive and simple way of designing controllers is to adopt a greedy approach. *Controller 2* as shown in Figure 7.2a and 7.2b is such a controller where the state is preserved if the goals assigned to the controller are met. In a real-life system, controllers like these need to be reset to their initial state when goals or algorithms to be executed change. However, this controller set does not allow the system to satisfy the throughput goal (as shown in Figure 9, w/o recommendation) even if the necessary resources are available.

As a next step, the effect of the recommendation unit is observed. The recommendation unit is used to connect the controllers as shown in Figure 6. RTE

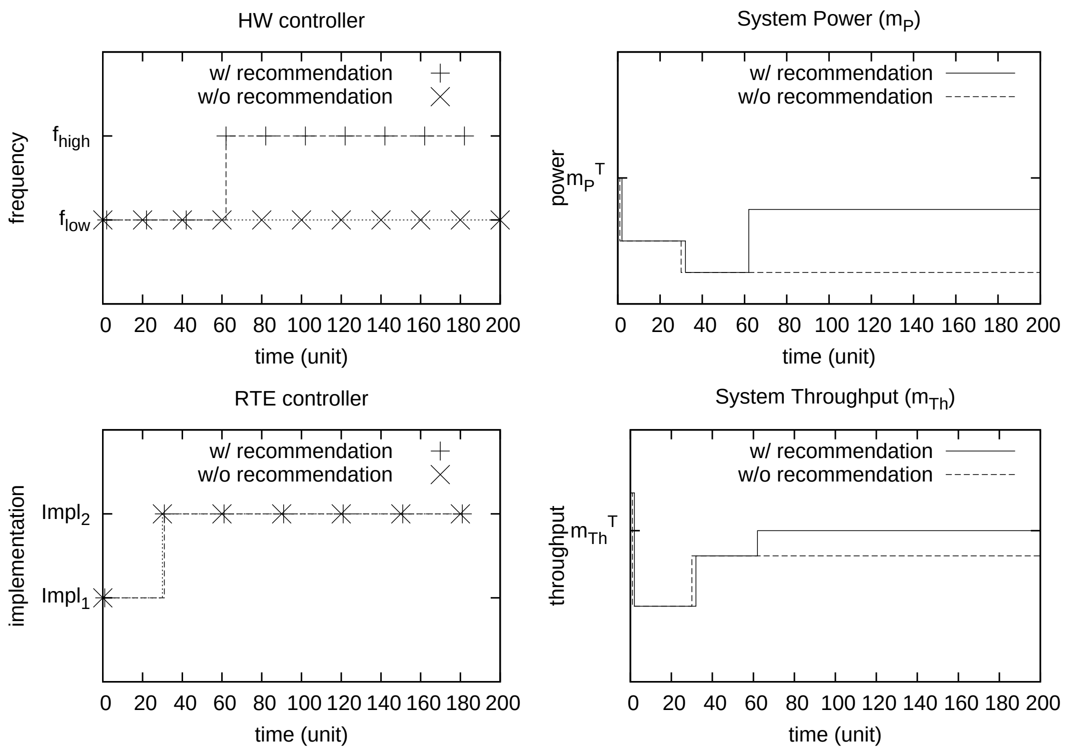

Fig. 9. Simulation results with *Controller 2* with and without the recommendation unit. ($T_{system} = 1 timeunit$, $T_{RTEController} = 30 timeunits$, $T_{HWController} = 20 timeunits$)

controller signals the activation of the recommendation unit. In this example, the only possible recommendation to the HW controller is a signal that means RTE controller wasn't able to meet its goals and that HW controller has to do something about it. Figure 9 shows also the adaptation behavior obtained with *Controller 2* and the recommendation unit. At time 60, we observe the intervention of the recommendation unit seeing the change of the operating frequency from f_{low} to f_{high} resulting in meeting of the goals at both levels.

As observed in the results above, different controllers at different levels give different results in terms of convergence of system self-adaptivity. The lesson learned is that either some restrictions are applied on local controllers (e.g., different reconfiguration periods), or a coordination mechanism (the recommendation system) is utilized.

5 Conclusions and Future Work

In this paper we have proposed a model for self-adaptive systems that incorporates many of the models proposed in the literature. A goal management methodology and goal specification interface, along with a decentralized and coordinated control mechanism, have also been proposed.

The model has been discussed – also with the help of a system state simulation – and its critical points have been put into light. The adoption of a decentralized control system for self-adaptivity provides simplified management by means of separation of concerns. Unfortunately, this decentralized mechanism also introduces some complications due to the need for coordination among the different controllers. This is required to guarantee convergence on the decisions taken by the controllers. Otherwise some restrictions on them should be applied to guarantee at least a sub optimal management of self-adaptivity. Globally, the decentralized control system wins over the centralized one, not only for simplicity reasons, but also because it provides an easier interchangeability of the hardware layer and better scalability.

As future work and model enhancements several different self-adaptivity control algorithms and parameters will be analyzed and tested. Moreover different schemes to partition the goals onto different levels will be explored and their effects will be observed. Finally, a system for checking the fulfillment of application goals in a specific system will be studied and proposed.

6 Acknowledgments

This work was partially supported and funded by the European Commission under the Project AETHER (No. FP6-IST-027611). The paper reflects only the authors' view; the European Commission is not liable for any use that may be made of the information contained herein.

References

[1] R. Al-Ali, A. Hafid, O. Rana, D. Walker, An approach for quality of service adaptation in service-oriented grids: Research articles, Concurr. Comput. : Pract. Exper. 16 (5) (2004) 401–412.

[2] D. Balasubramaniam, R. Morrison, K. Mickan, G. Kirby, B. Warboys, I. Robertson, B. Snowdon, R. M. Greenwood, W. Seet, Support for feedback and change in self-adaptive systems, in: WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, ACM, New York, NY, USA, 2004.

[3] L. Baochun, N. Klara, An Open Task Control Model for Quality of Service Adaptation, in: Proceedings of the 14th International Conference of Advanced Science and Technology (ICAST 98), Naperville, Illinois, 1998.

[4] L. Baochun, N. Klara, A Control-Based Middleware Framework for Quality-of-Service Adaptations, IEEE Journal on Selected Areas in Communications 17 (9) (1999) 1632–1650.

[5] L. Bauer, M. Shafique, D. Teufel, J. Henkel, A self-adaptive extensible embedded processor, in: SASO, IEEE Computer Society, 2007.

[6] G. Brown, B. H. C. Cheng, H. Goldsby, J. Zhang, Goal-oriented specification of adaptation requirements engineering in adaptive systems, in: SEAMS '06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems, ACM, New York, NY, USA, 2006.

[7] J. A. Casas, J. M. Moreno, J. Madrenas, J. Cabestany, A novel hardware architecture for self-adaptive systems, in: AHS '07: Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007), IEEE Computer Society, Washington, DC, USA, 2007.

[8] I. Foster, A. Roy, V. Sander, A quality of service architecture that combines resource reservation and application adaptation, Quality of Service, 2000. IWQOS. 2000 Eighth International Workshop on (2000) 181–188.

[9] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, P. Steenkiste, Rainbow: Architecture-based self-adaptation with reusable infrastructure, Computer 37 (10) (2004) 46–54.

[10] E. Gjørven, F. Eliassen, K. Lund, V. S. W. Eide, R. Staehli, Self-adaptive systems: A middleware managed approach, in: A. Keller, J.-P. Martin-Flatin (eds.), SelfMan, vol. 3996 of Lecture Notes in Computer Science, Springer, 2006.

[11] J. Gosling, H. McGilton, The java language environment (1999).
URL `http://java.sun.com/docs/white/langenv`

[12] A. Hafid, G. v. Bochmann, Quality-of-service adaptation in distributed multimedia applications, Multimedia Systems 6 (5) (1998) 299–315.

[13] M. J. Hawthorne, D. E. Perry, Exploiting architectural prescriptions for self-managing, self-adaptive systems: a position paper, in: WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, ACM, New York, NY, USA, 2004.

[14] L. Józwiak, Life-inspired systems and their quality-driven design., in: W. Grass, B. Sick, K. Waldschmidt (eds.), ARCS, vol. 3894 of Lecture Notes in Computer Science, Springer, 2006.
URL
`http://dblp.uni-trier.de/db/conf/arcs/arcs2006.html#Joz%wiak06`

[15] G. Karsai, Á. Lédeczi, J. Sztipanovits, G. Péceli, G. Simon, T. Kovácsházy, An approach to self-adaptive software based on supervisory control, in: IWSAS, 2001.

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

[17] J. Kramer, J. Magee, Self-managed systems: an architectural challenge, in: FOSE '07: 2007 Future of Software Engineering, IEEE Computer Society, Washington, DC, USA, 2007.

[18] S. Neema, Á. Lédeczi, Constraint-guided self-adaptation, in: R. Laddaga, P. Robertson, H. E. Shrobe (eds.), IWSAS, vol. 2614 of Lecture Notes in Computer Science, Springer, 2001.

[19] Open SystemC Initiative, SystemC Official Website, http:/www.systemc.org/.

[20] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, A. Wolf, An architecture-based approach to self-adaptive software (1999).
URL `citeseer.ist.psu.edu/oreizy99architecturebased.html`

[21] R. E. Schantz, J. P. Loyall, C. Rodrigues, D. C. Schmidt, Controlling quality-of-service in distributed real-time and embedded systems via adaptive middleware: Experiences with auto-adaptive and reconfigurable systems, Software–Practice & Experience 36 (11-12) (2006) 1189–1208.

[22] F. Vaughan, D. Munro, Self-adaptive compliant persisent architectures, in: Proceedings of the Seventh Integrated Data Environments - Australia (IDEA'07) Workshop, 2000.
URL `http://www.cs.adelaide.edu.au/users/jacaranda/publicati%ons/papers/idea-francis.pdf`

[23] J. Zhang, B. H. C. Cheng, Model-based development of dynamically adaptive software, in: ICSE '06: Proceeding of the 28th international conference on Software engineering, ACM, New York, NY, USA, 2006.