Simulation of a Self-adaptive Run-time Environment with Hardware and Software Components

Onur Derin Alberto Ferrante

ALaRI Institute, Faculty of Informatics
Università della Svizzera italiana, Lugano, Switzerland
{derino, ferrante}@alari.ch

ABSTRACT

In this paper we describe a new way for simulating self-adaptive systems developed by relying on a component-based approach, this approach proves to be useful both in easing self-adaptivity and in providing the ability to mix hardware and software elements.

Our simulation method is based on SACRE (Self-Adaptive Component Run-time Environment), a framework we have defined in Java for simulating self-adaptive systems.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; D.3.4 [**Programming Languages**]: Processors—*run-time environments*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program modification*

General Terms

Experimentation, Performance, Reliability, Theory

Keywords

Component-based design, HW/SW co-design, Self-adaptive systems, Simulation

1. INTRODUCTION

Multi-core architectures are inevitable if one needs to increase performance and still keep lower power profiles. Computational requirements for some applications even demand these architectures to be heterogeneous. This doesn't help but add on to the problems of exploiting these architectures to the most possible extent. To use the parallelism offered by these architectures new programming models are required. Furthermore, obtaining dependability, low-power consumption, and security is not automatic and proper control mechanisms should be put in place to reach these goals. One possible answer to these challenges is adopting a component-based approach where the application is specified as a network of components that are mapped to the heterogeneous units of the architecture. This calls for a middleware that provides standard interfaces to the components that may be residing on different cores, custom functional blocks, or reconfigurable fabric. Moreover, this middleware can provide the basis for the development of a distributed run-time environment that manages the adaptation of the application for high-level goals such as fault-tolerance, high performance and low-power consumption. Adaptations, at this level, can be performed by migrating the components between the available resources and/or by increasing parallelism by instantiating multiple copies of the same component on different resources.

Such a self-adaptive run-time environment constitutes a fundamental part in enabling system-wide self-adaptivity. 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 throughput). By defining requirements and adaptation mechanisms we give self-adaptive computational systems the ability to adapt to mutating internal and external conditions without requesting any intervention of the user [9]. 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. Self-adaptation capabilities are used to implement autonomic and life-inspired systems. These systems will have the ability to self-adapt and self-configure to provide the performance and the quality required [7] [5]. 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). Software-defined radio (SDR) [1] is a good example application that demands a platform such as the one mentioned above. Introducing self-adaptivity to an SDR system requires self-adaptivity at all levels of the system from the application to the operating environment and the hardware platform.

In this paper we propose a way to simulate hardware-software component-based self-adaptive systems. Application components and their mapping onto hardware resources are simulated at functional level. Components, that implement different functionalities, are selected and arranged together to compose applications. Components can be substituted by other components offering similar functionalities (e.g., different implementations of an encryption algorithm, or different encryption algorithms of the same class), thus easing self-adaptivity through replacement and reconfiguration of components and through reconfiguration of the network of components. We not only support self-adaptation through the replacement of components or through their reconfiguration, but we also support changes in the topology of applications, for example to accommodate parallel components with the aim of satisfying high-level goals such as performances and/or enhanced reliability.

In the remaining part of this paper we first provide an overview of related works (Section 2); in Section 3 we then provide a description of the model of self-adaptive system that we take as a reference in this work. In Section 4 we describe our simulation environment for self-adaptive component-based systems.

2. RELATED WORK

Different works on using a component-based approach to design hardware/software systems have been published. In [3] an approach to mix hardware and software components seamlessly at design time is presented. Robocop [8], with its run-time and resource-management frameworks, proposes a service-oriented architecture for designing consumer electronics devices. Both in [3] and in [8] wrappers are used to make hardware component exposing the same kind of interface as software ones. DeepCompass [2] is a framework, based on the Robocop model, for performance analysis and design space exploration. In [11] system-level design space exploration for reconfigurable heterogeneous systems is discussed. Kahn Process Networks (KPN) are used as the model of computation at application level. The paper also discusses how to avoid deadlock when executing KPN nodes in a heterogeneous platform. While our work shares a number of concepts with the aforementioned ones (namely, the use of wrappers for hardware components, the introduction of a middleware to support distributed components, and the problem of deadlock for KPN nodes), what we propose is aimed at supporting runtime self-adaptivity of devices and not just design time integration of components.

In [10] a framework for self-adaptive component-based software applications is described. The idea behind this framework is to provide a standardized way to manage self-adaptivity in software. For this reason, *separation of concerns* (i.e., putting different concerns, that are functionalities and adaptation management, into different components that address them independently) between adaptation management and software functionalities is adopted. Also in our work, which is based on the model of self-adaptive systems discussed in [4], we put emphasis on separation of concerns.

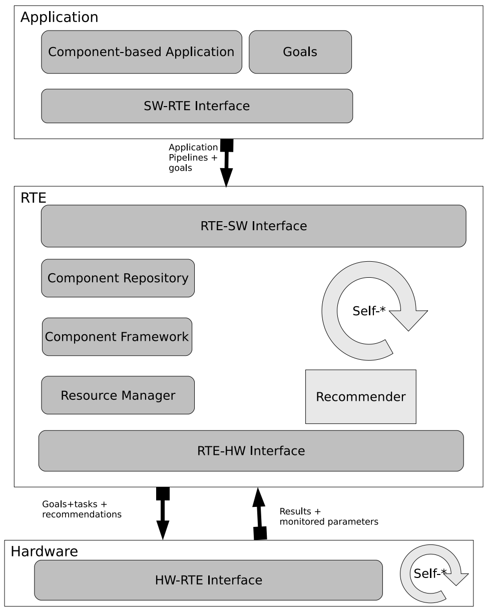

Figure 1: Component-based self-adaptive system.

The model discussed in [4] is based on a layered approach and it provides the capability to manage hardware and software self-adaptivity globally to satisfy system and application non-functional requirements (i.e., goals such as performances and power consumption). The two system layers defined are the hardware and the software ones. These two layers have separate self-adaptation mechanisms.

3. MODEL

Figure 1 shows the model of the self-adaptive system refined from [4] for component-based applications. The model is composed of a hardware and an application level, plus an intermediate level named Run Time Environment (RTE). The RTE and the application level are grouped together as a macro-level (*software macro-level*). 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 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 changing both software parameters or by changing using different implementations of the same functionalities. 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.

Compilers are used to bridge the semantic gap between the application programming language and the platform language. However compilation is a design-time activity whereas self-adaptation is done at run-time. In order to support adaptation at the application level and to truly separate the application logic from self-adaptivity, the semantics

```
resource CPU1: A, B, D
resource CPU2: C, G
resource FPGA1: E, F
pipeline App1: A ! B & C ! D ! E ! F ! G
```

Figure 2: An application described in SACRE.

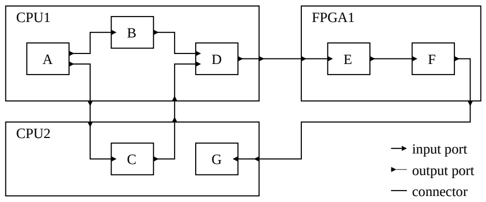

Figure 3: A sample application and its mapping.

of the application has to be present even at run-time. To address this problem, we propose to use component-based software for the development of self-adaptive applications. This will bring into the overall picture a component platform that consists of a component framework and a component repository. The component framework is the run-time system that implements the glue logic in compliance with the component model. Component model defines the standard interfaces between components and thus allows the framework to be aware of the run-time characteristics of software components. This awareness can be exploited to act as a semantic bridge between the application logic and the platform. This calls for a middleware that provides standard interfaces to the components residing on different cores, coprocessors, custom functional blocks on reconfigurable fabric for their communication. Moreover, this middleware can provide the basis for development of a distributed run-time environment that manages the adaptation of the application for high-level goals such as fault-tolerance, high performance and low-power consumption by migrating the components between the available resources and/or increasing the parallelism of the application by instantiating multiple copies of the same component on different resources. Such a self-adaptive run-time environment constitutes a fundamental part in enabling system-wide self-adaptivity. The middleware will be a software layer for the processor cores and a hardware wrapper for the custom hardware blocks. The eventual goals of this work are to realize a self-adaptation scenario by measuring the performance and power metrics as a result of changing the mappings of components on the platform at run-time; and to realize a fault-tolerance scenario in presence of a dynamically faulty unit. In achieving these goals, as the first phase, the system is being simulated at a functional level in order to identify the requirements for the functionality of system blocks shown in Figure 1. At later phases, the system will be refined to be able to run on real platforms.

4. SIMULATION

In order to simulate the self-adaptive HW/SW component run-time environment vision, we have chosen to extend our SACRE (Self-Adaptive Component Run-time Environment) framework.

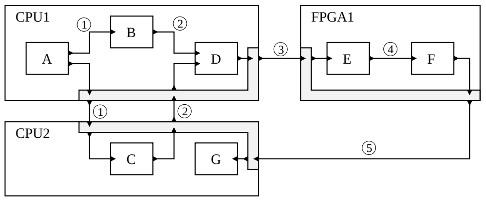

Figure 4: Middleware components (in gray) added to resource assemblies.

4.1 SACRE

SACRE has been developed originally for enabling self-adaptivity at application level. It allows creating self-adaptive applications based on software components and incorporates the Monitor-Controller-Adaptor loop with the application pipeline. It is based on the KPN model of computation [6], it has a simple language for creating component pipelines, and it is written in Java. A SACRE component is defined by extending from the *Component* abstract class and specifying its input and output ports as well as its *task()*. More formally a *pipeline* is a tuple (C, D) where C is a set of components; and D is a connection relation, $C \times C$ that defines the links between component ports. As long as there are no cycles (i.e., if there is a path from c_i to c_j, there is no path from c_j to c_i), there are no constraints on the read/write orders within a component. Otherwise, components are constrained to read and write in a specific order in order to guarantee deadlock-freedom. SACRE supports parametric and structural run-time adaptations over such pipelines.

4.2 Extending SACRE for the RTE layer

In order to simulate self-adaptation at RTE level, SACRE needs to be extended with new concepts such as *resources* and *mapping* of components onto resources. An *assembly* is a composition of components into a single component that has a set of input ports consisting of the unconnected input ports of the composed components; and a set of output ports consisting of the unconnected output ports of the composed components. A component contained in an assembly can not be part of another assembly. An assembly of software components is called a *composite* when the composition is meant to create a new software component. An assembly of software components is called a *resource* when it is meant to show that those components are running on the resource. Resources allow us to specify mapping of HW/SW components onto resources. A *mapping* is a total function from components to resources. Figure 2 and 3 shows a sample pipeline and its mapping corresponding to the sample SACRE description.

Implementing the assembly concept as an extension to a SACRE component brings some concerns for the deadlock-freedom of the system. If we define the *resource pipeline* as the pipeline that emerges from the composition of resource components, there is no guarantee that a mapping from a cycle-free application pipeline onto a set of resources will result in a cycle-free resource pipeline. For example, the resource pipeline in Figure 3 has two cycles (*CPU1-CPU2, CPU1-FPGA1-CPU2*). This can be overcome if we could impose an order on the reads and writes that go through the ports of the resource. If we implement read/write operations as a forwarded method call from the ports of the internal components of a resource to the ports of the resource, the read/write order cannot be imposed. Instead, access to the resource ports should be controlled by a middleware component that imposes an orderly read/write to the external resource channels in order to guarantee deadlock-freedom. Figure 4 shows how these custom middleware components are inserted into the pipeline. The *task()* of a middleware component is simply implemented by calling read and write operations on its ports (i.e., forwarding of tokens from its inputs to outputs) without causing a deadlock. This order is determined by a labeling function that labels the edges of the application pipeline with an integer value that represent the maximum number of application components that a token may go through until arriving to that edge. The middleware is responsible for forwarding the tokens from the input channels with a smaller-labeled-channel-first fashion. Figure 4 shows the labels for the example pipeline.

Having extended the SACRE framework with resource and mapping concepts, in order to implement the adaptation capabilities at RTE level, we need to extend the adaptable application pipeline concept that currently exists in SACRE to adaptable resources and adaptable resource pipelines. Since resources are SACRE components, parametric adaptation of resources comes for free. Structural adaptation of the resource pipeline implies having a dynamic number of CPUs and FPGAs on board. Since this is not possible at run-time on a real platform, we should actually consider the structural adaptation of the application pipeline over the resource pipeline. One type of such RTE level adaptations is the migration of a component from one resource to another. Considering again the example in Figure 3, a more favorable mapping in terms of performance and communication bandwidth may be if component C was mapped to *CPU1*. In that case component C could have had a better cache hit rate by running on the same core as its source component. To migrate a component, the adaptor has to be able to create a new component of the same type as the migrated one in the destination resource; the middleware components should be able to remove/add the ports that will serve to the communication of C in the destination resource. The parallelization adaptation pattern involves creating a parallel instance of a component and introducing input routers and output mergers. This adaptation would increase the throughput if the instances were mapped onto different CPUs. The reliability adaptation pattern involves parallelizing instances of a component on different cores along with multiplicator components and majority voter components for each input and output ports respectively. Similarly mapping decisions for these components may have implications on the performance measurements of the system.

5. FUTURE WORK

We are currently implementing the extensions as described in this paper to the SACRE framework. This work will allow us to refine the self-adaptive system and eventually implement it on a real platform. However even at such functional level, simulation of the self-adaptive RTE can allow us to experiment with the adaptation control algorithms that may vary from centralized to decentralized strategies. Associating costs (e.g. cycles, latency, power) to application components in relation with the resource they execute on, we can obtain some estimates to drive different mapping policies. Moreover application of adaptation patterns in presence of resources needs to be studied in future refinements of the system.

6. ACKNOWLEDGMENTS

This work was partially 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.

7. REFERENCES

[1] JTRS software communications architecture (SCA). WWW page. http://sca.jpeojtrs.mil.

[2] E. Bondarev, M. R. V. Chaudron, and E. A. de Kock. Exploring performance trade-offs of a jpeg decoder using the deepcompass framework. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pages 153–163, New York, NY, USA, 2007. ACM.

[3] C. Bunse and H.-G. Groß. Unifying hardware and software components for embedded system development. In R. H. Reussner, J. A. Stafford, and C. A. Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 120–136. Springer, 2004.

[4] O. Derin, A. Ferrante, and A. V. Taddeo. Coordinated management of hardware and software self-adaptivity. *Journal of Systems Architecture*, 55(3):170 – 179, 2009.

[5] L. Józwiak. Life-inspired systems and their quality-driven design. In W. Grass, B. Sick, and K. Waldschmidt, editors, *ARCS*, volume 3894 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2006.

[6] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

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

[8] H. Maaskant. A robust component model for consumer electronic products. In F. Toolenaar and P. van der Stok, editors, *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, pages 167–192. Springer, The Netherlands, 2005.

[9] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software, 1999.

[10] T. L. Pierre-Charles David. Towards a framework for self-adaptive component-based applications. *Lecture Notes in Computer Science, Distributed Applications and Interoperable Systems*, 2893:1, 14, 2003.

[11] K. Sigdel, M. Thompson, A. Pimentel, T. P. Stefanov, and K. Bertels. System-level design space exploration of dynamic reconfigurable architectures. In *the 8th international workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (Samos)*, pages 279 – 288, July 2008.