Enabling Self-adaptivity in Component-based Streaming Applications

Onur Derin

Alberto Ferrante

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

ABSTRACT

Self-adaptivity is the capability of a system to adapt itself dynamically to achieve its goals. By means of this mechanism the system is able to autonomously modify its behavior or the way in which applications are run and implemented to achieve the goals set.

In this paper we propose a framework that uses a component-based approach to implement self-adaptivity at application level. By using this mechanism, the framework provides the ability to perform both adaptation on the structure of the application (i.e., how the components are connected together) and on internal parameters of each component. At application level, there is a mechanism to monitor different parameters and to check whether the system is meeting the assigned goals or not. A controller drives adaptations when goals are not met.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*; D.3.4 [**Programming Languages**]: Processors—*run-time environments*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program modification*

General Terms

Performance, Reliability, Theory

Keywords

Component-based design, Self-adaptive systems

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 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 [8].

Self-adaptation capabilities are used to implement autonomic and life-inspired systems. These systems 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 system components 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 different parts of the system and each part may or may not be aware of the self-adaptivity capabilities of the other.

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.

In this paper we propose an approach to enable self-adaptivity at application level along with a component framework that implements it. We propose to use a *component-based* approach by which 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); the use of components

ease self-adaptivity by easing their substitution and their reorganization to provide the same results in different ways. In this paper we only consider software components. Though, hardware components can also be used and mixed with software ones [3]. The model of self-adaptive system that we took as a reference [4] natively provides this capability. The framework we propose here not only support self-adaptation through the replacement of components or their reconfiguration, it also supports 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. Adaptation control is not discussed therefore no considerations are done on performance characteristics of adaptation (e.g., timing, convergence). A case study is discussed to show the self-adaptivity features provided by the model.

In the remaining part of this paper we first provide an overview of related works (Section 2). In Section 3 we discuss our framework for enabling self-adaptivity at application level. Section 4 provides the description of a case study application that can be implemented by using our framework.

2. RELATED WORK

In [9] 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* between adaptation management and software functionalities is proposed. Self-adaptivity is reached by applying a set of *adaptation policies* on software components. The adaptation of these policies is triggered by certain system events that can be configured. Possible adaptations are both in component behavior and parameters. Unfortunately, the authors do not discuss if and how general goal achievement is obtained.

In [2] WildCAT, a java framework for context-aware software applications, is described. The framework provides a dynamic model to represent the execution context of applications. In [1] a similar approach is proposed and tested on software for embedded systems; a list of challenges for offering self-adaptivity in embedded systems is also provided. The challenges listed are: flexibility, efficiency and minimality, safety, and simplicity.

In this paper we build a new framework for self-adaptivity based on the model of self-adaptive systems discussed in [4]. This model is generic enough to include a large number of electronic systems. The model 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. We extended this model with a component framework at the application level that allows managing self-adaptation and works hand-in-hand for system-wide goal-achievement with the rest of the system. Yet, this framework provides separation of concerns. The work proposed in [9] does not provide these possibilities. Furthermore, it does not take into account the possibility of also including hardware components along with software ones. The model we are using

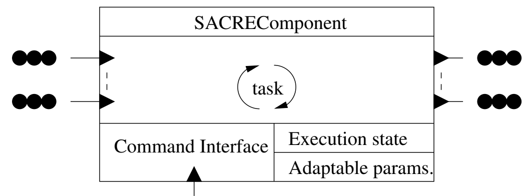

Figure 1: A SACRE component

takes into account this possibility and future versions of the framework will also include it. Furthermore, the framework we are proposing allows the system to adapt the topology of components used to form applications.

3. SELF-ADAPTIVE APPLICATIONS

In order to show that a component-based approach is a viable way to enable self-adaptivity at application level, we are developing the SACRE (Self-Adaptive Component Runtime Environment) component framework that allows creating self-adaptive applications based on software components and incorporates the Monitor-Controller-Adapter loop with the application pipeline. This framework is being implemented in Java. In the remaining part of this section we describe the framework and the adaptations that it supports.

3.1 The SACRE Framework

A component framework is the ground on which developers can define components and put them together to create an application. The main constituents of SACRE are its component model, connectors, and pipeline.

The *component model* is based on the KPN model of computation [6], it has a simple language for creating component pipelines. A component is defined by extending from the *Component* abstract class and specifying its input and output ports as well as its *task()*. The framework provides a base class for new components to be created. Each component has a thread of its own and can have named input and output ports through which it can exchange typed messages (tokens).

A *connector* allows to transmit messages from the output port to the input port that it is connected to. Blocking FIFO queues are used as connectors between component ports. A connector holds a queue of typed messages. The component thread is blocked in case it tries to read a message from an empty connector through any of its input ports. Presently, there is no bound on the size of the queue in the model, thus there is no blocking during writing to an output port. A port can be connected to at most one connector.

A *pipeline* is a graph of components that are connected through connectors along their ports. More formally a *pipeline* is a tuple (P, D) where P is a set of component ports; and D is a connectivity relation, $P \times P$ that defines the links between component ports. As long as there are no cycles (i.e., if there is a message path from component 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 from and write to their ports in a specific order in order to guarantee deadlock-freedom.

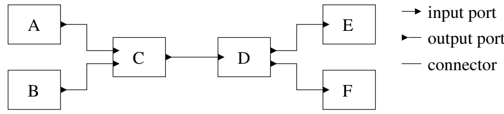

Figure 2: A pipeline example: $A\&B!C!D!E\&F$

A pipeline language for assembling components have been implemented as shown in Figure 2. A pipeline is a list of statements separated by a semi-colon(*;*). Each statement consists of *!*- or *&*-separated list of components. *!* and *&* are used for serial and parallel composition respectively. In the statements *&* has a precedence over *!*.

3.2 Adaptations Supported

The possible adaptations in a component-based application are: adaptation of a parameter of a component, on-the-fly replacement of a component with another compatible component, adaptation of the level of parallelism of the application by creating parallel instances of the component, and adaptations by transforming the component graph to achieve goals such as security and dependability. The last three kinds of adaptations can be classified as structural adaptations as they impact on the structure of the graph.

There are a number of features that make the SACRE framework support these adaptation capabilities. *i)* Command interface allows the component to accept adaptation commands. It is implemented as a port and it gets polled for commands every time before the task of the component is invoked. *ii)* Components can declare their adaptable parameters by name. *iii)* Ports can be connected to a different connector at run-time. *iv)* Ports can be blocked. A blocked port always returns a *stop* token. *v)* Hooks can be attached to input and output ports at run-time. A hook is a piece of code that processes the token going through the port that it is attached to. A hook may choose to drop a token. *vi)* Tokens can be tagged. Additional information can be attached to tokens dynamically as they go through the component network. Figure 1 presents a visualization of a SACRE component.

3.2.1 Parametric adaptations

In order to enable parametric adaptations of components, a parameter value update command is sent along with the name and the new value of the parameter to the command queue of the component. However such updates get effective only after the component is finished with the processing of the message that is already being processed by the component at the time of the parameter update request. The update command is handled by setting the parameter to its new value. In case the semantics of the application requires some more actions to be done in case of a parameter update, an abstract method can be implemented by the application developer to add custom update code.

3.2.2 Structural adaptations

In order to enable on the fly modifications to the structure of the pipeline, ports are made blockable, meaning that, if blocked during adaptation, connectors won't deliver messages. Therefore a component replacement type of adaptation is achieved by blocking all the input ports of a component; flushing the component by adding hooks to the output ports that filter *stop* tokens and set components execution state to *flushed*; disconnecting the ports from the connectors; connecting the ports of the new component to the unbound connectors; and starting the new component's thread. An important point here is the behavioral compatibility of the new and the old component. At syntactic level, this can be checked by the compatibility of port types (determined by the message type of the port). Other structural adaptations can be done by following some adaptation patterns. Below, we describe parallelization and dependability patterns.

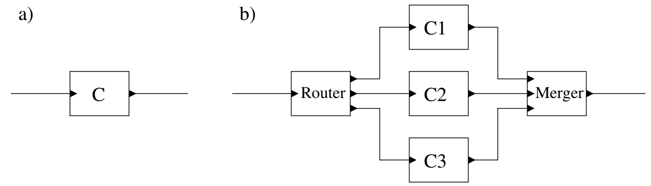

Figure 3: Adaptation pattern for parallelization. Component in (a) is parallelized by three as shown in (b)

Parallelization of a component is one type of structural adaptation that can be used to increase the throughput of the system as shown in Figure 3. This is done by creating parallel instances of a component and introducing a router before and a merger after the component instances for each of the input and output ports. A router is a built-in component in our framework that can work in a load-balancing or round-robin fashion; this component routes the incoming messages to either one of the instances depending on its policy. If there is no ordering relation between incoming and outgoing messages, the merger components simply merge the output messages from the output ports of the instances into one connector disregarding the order of messages on the basis of whichever message is first available. However, for the general class of KPN applications, semantics require that the processes comply with the monotonicity property. In that case, the ordering relation has to be preserved. For that purpose, the router component tags every message that would originally go to the component with an integer identifier that counts up for each message from value 1. Then the merger components have to queue up the output messages so as to achieve an order in terms of their tags. If there are multiple processor cores available, this mechanism would increase the parallelism of the application. However the condition for applicability of such an adaptation is the absence of inter-message dependencies.

By using a similar mechanism, our framework can adapt the dependability at the application level by structural redundancy as seen in hardware design. Parallel instances of the component are created on different cores along with multiplicator components and majority voter components for each input and output ports respectively as shown in Figure 4. Multiplicator component creates a copy of the incoming message for each redundant instance and forwards it to them along with a unique tag identifying the set of copied messages. Majority voter component queues up all the output messages until it has as many messages with the same tag as the number of redundant instances. Then it finds out the

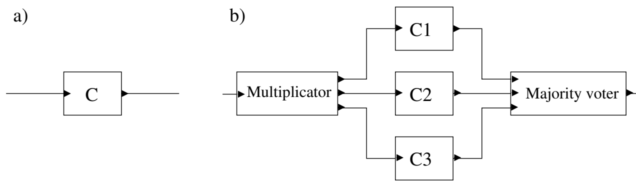

Figure 4: Adaptation pattern for dependability. Component in (a) is replicated by three as shown in (b)

most recurrent message and sends it to its output connector. A time-out mechanism can also be put in place to tolerate when a core is faulty and no message is being received by a component.

4. CASE STUDY

As shown in Figure 5, a self-adaptive MJPEG streaming server is being realized on our self-adaptive component framework. *Source* component grabs the raw image frames and feeds the pipeline with tokens of type *Frame*. *Frame* contains the image data along with the width, height, maximum grayscale value and quantization table. Initially the image data is in PGM format and it gets changed along the pipeline as the frame goes through the components (2D discrete cosine transform, zigzag scanning, run-length encoding). Sink component creates the MJPEG stream and serves it over to the network.

We are considering a scenario where the application goal is to maintain a fixed streaming rate in frames per second (FPS). This goal implies a maximum latency of $1/FPS$ seconds for the pipeline. To create a sample adaptation space, the quantization component is designed with an adaptable *picture quality* parameter; the source can be also tuned to provide images with different sizes (*picture size* parameter). Reduced quality in encoding results in smaller frame size. Similarly, reducing picture size results in a smaller frame size as well as a smaller latency. Parallelization pattern can be applied to the DCT as it is the most computation intensive component. Monitoring of the latency is done by measuring the time it takes for a frame to reach the sink component. Network bandwidth is monitored by a boolean variable that indicates whether the throughput before the network queue is greater than the throughput after the network queue. An adaptation control algorithm can be implemented that takes the difference between monitored and target latency as its error value and controls the picture quality and size taking also into account the current bandwidth condition. Currently we are implementing the self-adaptivity features of our component framework on this video streaming server case study.

5. CONCLUSION

In this paper, we propose an approach to implement self-adaptive streaming applications on component frameworks. This approach is being implemented in the SACRE framework. SACRE currently supports on-the-fly replacement and parametric adaptations. Adaptation patterns are currently being implemented. A self-adaptive video streaming server is being developed on our component framework to validate the proposed approach. Currently our approach supports KPN applications where processes are stateless. Future work will focus on overcoming this limitation by transferring the state during component replacement, completing the case study and investigating automatic synthesis of application level controllers and monitors from application goals.

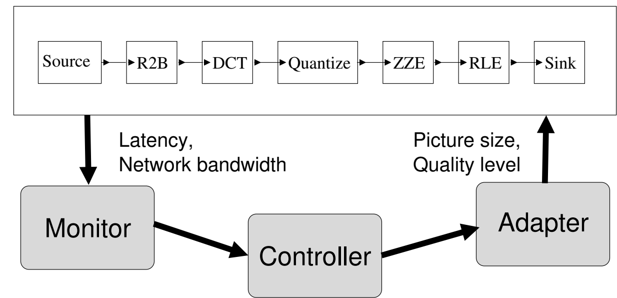

Figure 5: Block diagram of the self-adaptive MJPEG streaming server

6. REFERENCES

[1] J. Buisson, C. Carro, and F. Dagnat. Issues in applying a model driven approach to reconfigurations of satellite software. In *HotSWUp '08: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, pages 1–5, New York, NY, USA, 2008. ACM.

[2] P.-C. David and T. Ledoux. Wildcat: a generic framework for context-aware applications. In *MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7, New York, NY, USA, 2005. ACM.

[3] O. Derin and A. Ferrante. Simulation of a self-adaptive run-time environment with hardware and software components. In *Proceedings of the Workshop on Software Integration and Evolution @ Runtime (SINTER'09)*, New York, NY, USA, 2009. ACM.

[4] O. Derin, A. Ferrante, and A. V. Taddeo. Coordinated management of hardware and software self-adaptivity. *J. Syst. Archit.*, 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] 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.

[9] 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.