Timing driven architectural adaptation

Andrew Wils, Yolande Berbers, Tom Holvoet and Karel De Vlaminck

K.U.Leuven DistriNet
Department of computer science
Celestijnenlaan 200 A, 3001 Leuven
`andrew.wils | yolande.berbers |`
`tom.holvoet | karel.devlaminck @ cs.kuleuven.be`

Abstract. Self-adaptation is currently addressed in general frameworks and reference architectures but not in the application architecture. This paper defines concrete concepts to specify timing driven self-adaptation in the software architecture. This self-adaptation is aimed at high-end embedded component based applications. We create an architectural view of a music application describing this kind of adaptation and discuss its implementation. The novelty of our approach is the definition of separate constructs for the monitoring, the adaptation decision logic and the adaptation itself. This allows independent specification of policy and mechanisms and the possibility to adapt other applications in order to satisfy important constraints. The implementation itself consists of reusable run-time counterparts of the constructs. These counterparts are managed by the component middleware and configured by the architectural specification. This way one does not need to write additional self-adaptation code.

1 Introduction

The increasing diversity and interconnection of applications leaves much of their configuration to be dealt with at run-time. The vision of autonomic computing acknowledges this as a problem leading to new levels of complexity [1]. The autonomic computing solution is that computing systems should manage themselves given high-level objectives. One of these objectives is to maintain a satisfactory performance regardless of the available resources. This paper addresses self-adaptation to uphold timing constraints. We focus on applications with CPU intensive tasks for which the user has performance expectations. An example of such an application is the music community application presented in Figure 1. This application enables the user to browse, play and share music as well as chat about it. These tasks are CPU intensive, yet we want an acceptable Quality-of-Service (QoS) for all of them, even in situations with widely varying resources. The self-adaptation will change the resource consumption by means of coarse-grained adaptations to uphold a satisfactory QoS. To master the complexity of this kind of self-adaptation, we need mechanisms and policies that specify and control the adaptation process.

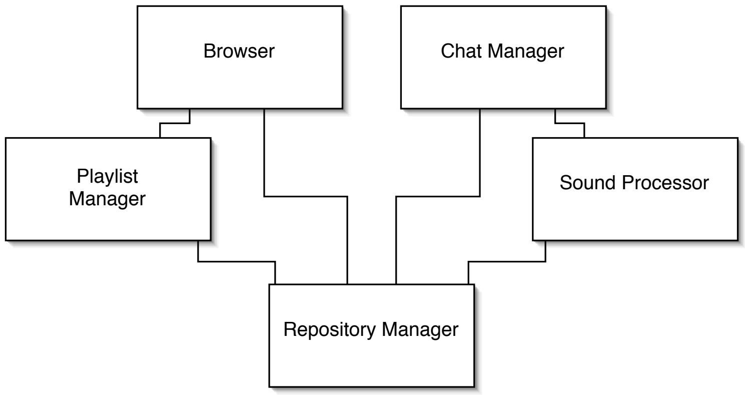


Fig. 1. High level component diagram of a music community application

The ideal solution relies on the perfect prediction of an application's resource behavior based on resource profiling, but this is not an option for unknown platforms and interactive behavior. A more realistic approach is the use of general feedback based frameworks and architectures with reusable adaptation mechanisms (e.g. [2], [3]). This approach usually employs a general "observe – process – adapt" cycle. Although this cycle provides a good starting point for application adaptation, there is still a gap to fill to enable the actual implementation of such applications.

In particular, the frameworks advocate reusable mechanisms but do not define a high-level approach to specify the use of the mechanisms. Timing constraints are non-functional requirements and pertain to large portions of the application. Likewise, coarse-grained application adaptation should also be specified at a level that is close to the functional requirements. To illustrate this, let us reconsider the music community application. The basic functionality of the application is built around distributed music repositories. Using these repositories, users can browse, play and share each other's music. Although the diagram in Figure 1 only shows a coarse architectural view, much of the timing driven adaptation can already be defined at this level. Obvious constraints are that the music must not stutter (a constraint on the Sound Processor), and that the GUI control widgets are responsive enough. Appropriate adaptation actions could be: omitting certain components (such as the optional chat service), switching components (one could use an alternative codec component in the Sound Processor) and "tuning" the resource behavior of a component (e.g. by setting a codec's compression rate).

We propose a feedback-based solution to uphold timing performance that is twofold. First, we introduce two architectural constructs to specify independently the monitoring of timing constraints and the execution of architectural configuration changes. Such coarse grained adaptations have a large influence on resource use, are easy to specify on a high level and do not require changing the application's functional components. Second, we offer an explicit architectural construct to encapsulate the decision logic that links constraint monitoring and adaptation. This way, the application developer can specify a coarse-grained run-time adaptation policy in the architecture at design-time.

Throughout the paper, we will illustrate the introduced concepts with the music community application. We show that reified run-time counterparts of our constructs reduce the addition of timing driven adaptation to providing the right architectural specifications.

2 Architectural approach

The software architecture of an application provides a coarse-grained decomposition in components and connectors. It abstracts away the complexity of the low level design and enables reuse of functionality. Also, a reified implementation of components allows to adapt the application while it is running [4]. Apart from the user requirements, software architecture should also cover important non-functional requirements and show how it can uphold these requirements.

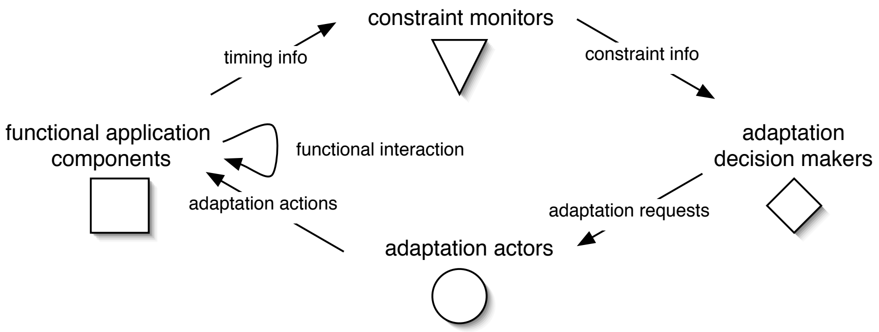


Fig. 2. Architectural constructs and the "observe - process - adapt" cycle

Figure 2 shows how we achieved this. In particular, we concretized the "observe - process - adapt" cycle [4] for timing based application adaptation using special architectural constructs. Figure 2 introduces three new constructs to complement application components:

constraint monitors specifying timing constraints on message flows between application components and how these constraints must be checked at run-time;

adaptation actors specifying how application components and component compositions may be altered at run-time;

decision makers detailing an adaptation policy to connect monitor evaluations with the appropriate adaptation actors.

We call the architectural view that describes this monitoring-based adaptation the *run-time adaptation view*. This view is based on a component instance diagram but defines a run-time adaptation process with reified versions of the above constructs as follows. Decision makers encapsulate QoS levels for a component group. Constraint evaluations determine this QoS and the decision maker maps QoS levels to adaptation actions that are to be executed to uphold the QoS.

The next sections further detail the syntax and semantics of constraint monitors, adaptation actors and decision makers. Following this, we present and evaluate a reusable run-time implementation to carry out the architectural monitoring and adaptation specification.

3 Case study

As an illustration of a diverse and interconnected distributed application, we chose to implement a prototype version of the mentioned music community application. This application, called Dale, was designed to use timing driven self-adaptation so that it could run on a variety of platforms. Adding constraints and adaptation in the implementation or even the object oriented design is not an easy task: the application consists of over 50 classes, whereas the architectural component view is much simpler and closer to the user requirements. Figure 3 shows a part of the Dale component instance diagram focusing on playlists. It is this diagram for which we will create a run-time adaptation view.

The diagrams in the paper use a component model with asynchronous message based communication. Messages can only be sent out and received through ports. Connectors relay messages from one port to one or more others.

The `Repository` component forms the heart of the Dale application: it contains songs and their meta-data. The `Playlist Manager` defines this playlist and dictates songs to play to the `Player` component. The latter is responsible for setting up the right `Codec` component and getting the audio through a `Fader` component to a `Sound Output` component. Finally, the `Queuer` component keeps the playlist automatically filled and the optional `Assistant` offers advice on related songs to queue.

4 Architectural Constructs

4.1 Constraint Monitors

At the software architecture level, ports declare the messages that are processed and passed around between components. We introduce *constraint monitors* to encapsulate declarative timing constraints on a number of events pertaining this processing and passing around of messages. Architecturally, we link constraints

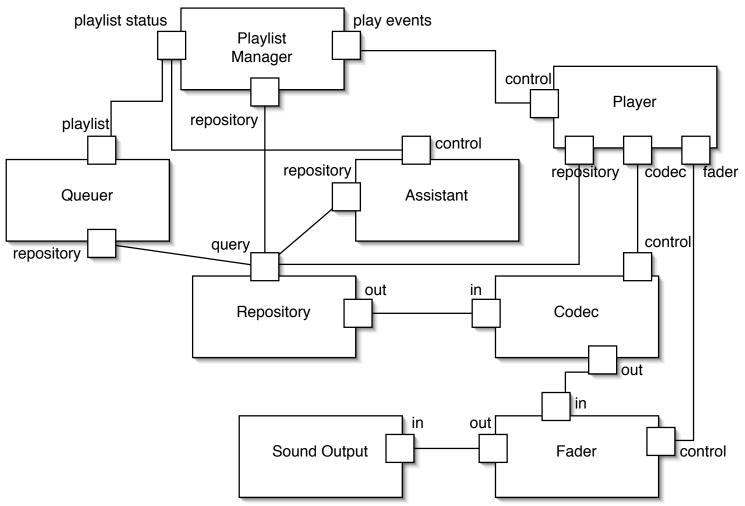


Fig. 3. Playlist component instance diagram

to ports rather than to connectors in order not to limit a constraint to events involving two directly connected components. For example, in Figure 3, one could specify a deadline involving ports from the `Playlist Manager` and `Sound Output`, although these are not directly connected to each other. The constraint monitor construct is depicted as a triangle and is attached to all ports that are involved in the timing constraints.

Constraint monitors do not only encapsulate constraint information, they also state that the constraints should be monitored at run-time. For this reason, we define time as it is handled at the target platforms: a series of discrete time events (monotonously increasing) generated by a system clock. To express the message related events, we adopt an event model defining three types of events:

send corresponds to the sending of a message through a port;
receive corresponds to the reception of a message on a port;
processed corresponds to the end of processing in the component. When this event is reached, a component without a thread of its own will no longer send outgoing messages (send events) until it receives a new message.

The BNF syntax is as follows:

$$\langle\text{message event}\rangle \longrightarrow \langle\text{port}\rangle\texttt{:}\langle\text{message}\rangle_{\texttt{send}\,|\,\texttt{receive}\,|\,\texttt{processed}}$$

For example, for an audio decoding component, $\texttt{mp3:packet}_{receive}$ signifies the arrival of the message `packet` on the port `mp3`. Similarly, $\texttt{mp3:packet}_{processed}$ means that the component to which the port belongs has processed the given packet and sent out the decoded audio, if any.

To model deadlines, we chose a modified language based on RTL ([5], [6]), although our approach can be used with other formalisms. The RTL syntax uses the @ function to denote the occurence time of a particular event. To give an example involving the `mp3` port, here is a constraint limiting the processing time of the request to 20 ms:

$$@(\mathrm{mp3\colon packet}_{processed}) \leq @(\mathrm{mp3\colon packet}_{receive}) + 20\mathrm{ms}$$

This particular form applies to all instances of `mp3:packet`. Another use of the @ function has more fine-grained control. For example, the following function indicates that the time between two successive instances of `mp3:packet` should be equal or less than 20 ms:

$$@(\mathrm{mp3\colon packet}_{receive},\ i + 1) \leq @(\mathrm{mp3\colon packet}_{receive},\ i) + 20\mathrm{ms}$$

Figure 4 shows the constraint monitors that were defined in the earlier presented playlist related instance diagram. Curved connections distinguish the interaction from software connector-based interaction: monitors do not influence (functional) behavior, they merely observe. Of course, the act of observation always influences non-functional aspects, such as performance. Section 5 shows that the involved run-time overhead is limited.

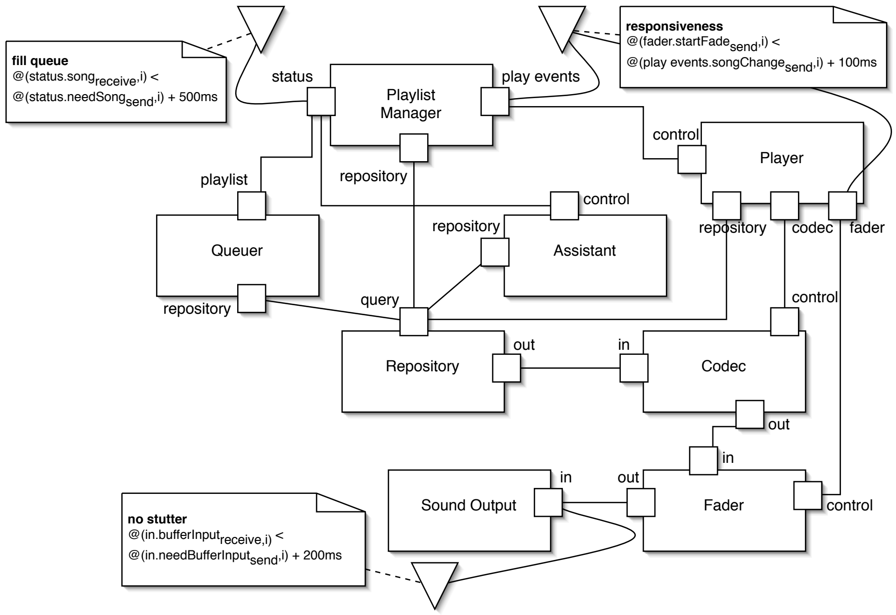


Fig. 4. Constraint monitors for the Dale playlist diagram

The `fill queue` constraint monitor in Figure 4 places an upper-bound on the time between the playlist reporting it needs a new song, and the queuer providing the song. The `responsiveness` constraint monitor checks that song changes are timely processed by the player (eventually resulting in a fader command). Finally, the `no stutter` constraint monitor checks for audio buffer under-runs.

4.2 Adaptation Actors

As mentioned in the introduction, we want to specify and support coarse-grained changes in resource consumption for component based applications. We define constructs to describe the manipulation of messages and components for the following adaptation actions:

Component configuration: a component's resource consumption is changed by configuring its settings. This is the equivalent of changing variability parameters [7];

Run-time component optionality and variability: resource use is changed by omitting components or by choosing alternative components. A static approach for this has already been investigated in [8].

We put forward two major objectives for our constructs:

Adaptations are fully determined by the specification. No extra application code is necessary to enable the adaptations at run-time. The adaptation code can be automatically generated or the middleware coordinates the adaptations. This reduces the developer's burden and speeds up development time;

Adaptations are carried out efficiently. The user of the system should only notice the effects of an adaptation, not the adaptation action itself. The described adaptations should therefore be easily translatable to an efficient adaptation mechanism, to keep user distraction to a minimum.

An *adaptation actor* is an architectural construct that encapsulates such architectural modifications. These manipulations are formulated into adaptation action recipes that are to be executed at run-time by reified counterparts of the actors. These counterparts can send messages just like regular components. The graphical representation of an adaptation actor is a circle that is associated to a block containing the action recipes. We define two types of actors: the *message router* and *component tuner*. Recipes have a name and body and are specified as follows (curly brackets in fixed font represent the beginning and ending of the recipe body):

$$\begin{aligned}\langle\text{adaptation recipe}\rangle \longrightarrow\ & \textit{recipe}\,\langle\text{recipe name}\rangle\,\texttt{\{}\\ & \{\langle\text{adaptation action}\rangle\,;\}\\ & \texttt{\}}\end{aligned}$$

In what follows, we describe the different adaptation actors and how they can be used to achieve the earlier mentioned adaptations. Just like regular components, adaptation actors have ports, but they are not annotated with rectangles to avoid overloading the diagrams. All described adaptation actions are accomplished by sending and receiving messages through these ports.

The purpose of a message router is to specify run-time "switching" of message flows. This is done by a reified run-time counterpart that relays messages across its ports. To specify this, we define the `link` and `unlink` actions:

$$\langle \text{link action} \rangle \longrightarrow \textit{link}\ (\langle \text{port} \rangle , \langle \text{port} \rangle)$$
$$\langle \text{unlink action} \rangle \longrightarrow \textit{unlink}\ (\langle \text{port} \rangle , \langle \text{port} \rangle)$$

When a link action is executed, all messages that are received through the first stated port must be relayed to the second. Using this, one can enable "multicast" or "router" like message flows. Figure 5 shows the message routers we have defined for the music application. The `queuer router` switches requests for new songs from an ad-hoc based queue algorithm to a more CPU intensive one that takes into account user preferences. The other two routers switch between a normal fader and one that cross-fades between songs. These two have been paired by a dotted line, meaning that the recipe declaration applies to both. The `Crossfader` component needs 2 decoded streams at a time and is more CPU intensive. Note that we make some assumptions about component state and message synchronization. First, switching of message flows should only be used when the components that receive messages directly or indirectly from a message router do not require reception of all messages. Second, a component receiving messages via different "switched" paths must not rely on the order of received messages, as we are using an asynchronous component model.

A component tuner configures one or more components. The actor achieves this by sending configuration messages to regular component ports.

The action for sending a message is as follows:

$$\langle \text{send action} \rangle \longrightarrow \langle \text{port} \rangle ... \langle \text{message} \rangle\ (\,[\,\{\langle \text{key} \rangle : \langle \text{val} \rangle\}\,]\,)$$

The triple-dot notation denotes the sending of the message with the supplied key-value parameters.

The semantics of a component tuner are straightforward: when a recipe contains one or more of these send actions and the former is executed, the described messages are sent out through the denoted port, optionally containing the specified key-value parameter pairs. Figure 6 shows a component tuner in the Dale music application that deactivates the assistant, freeing up CPU resources.

4.3 Decision Makers

Now that we have adaptation actors and timing monitors, we need to link them. We base the logic on levels of perceived QoS that are defined in another architectural construct called a *adaptation decision maker*. Decision makers encapsulate how the timing constraints and adaptation actions relate to the perceived quality at run-time. A level's quality is defined by one or more constraint monitor

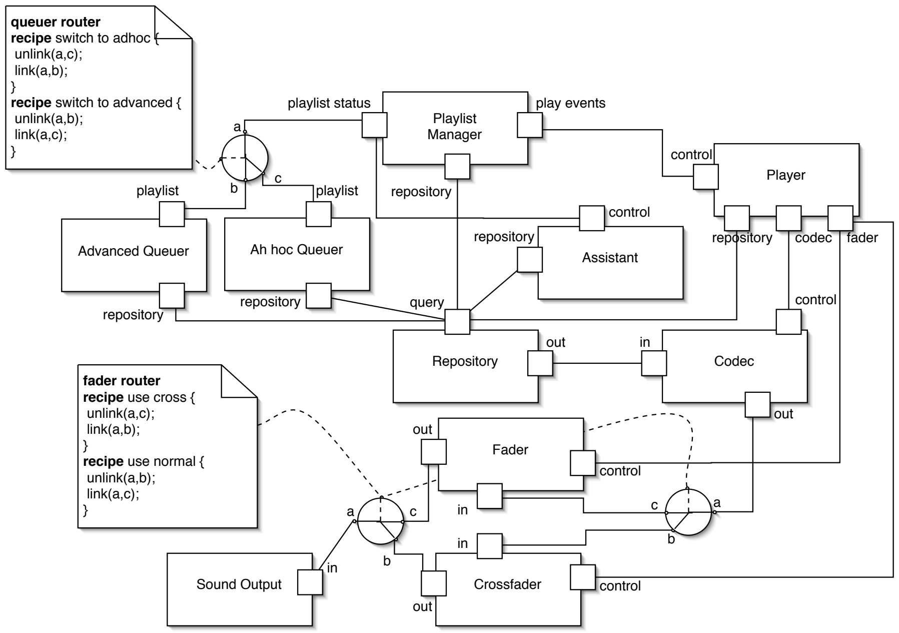


Fig. 5. Dale message routers

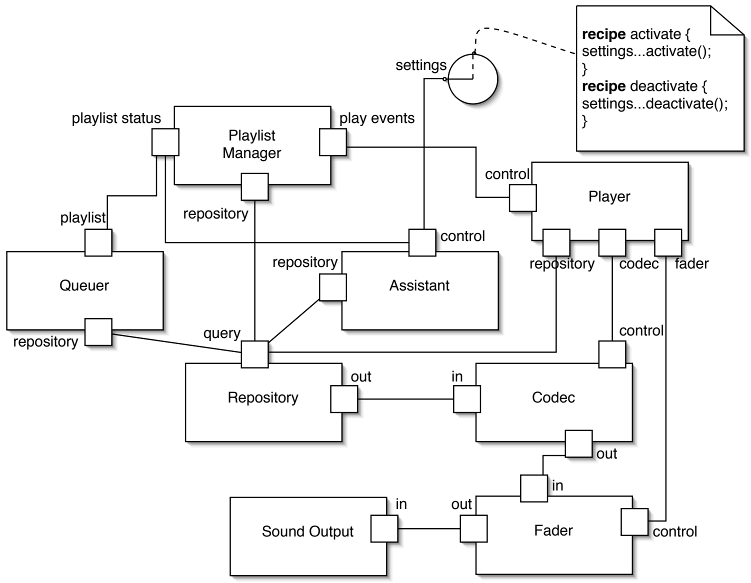


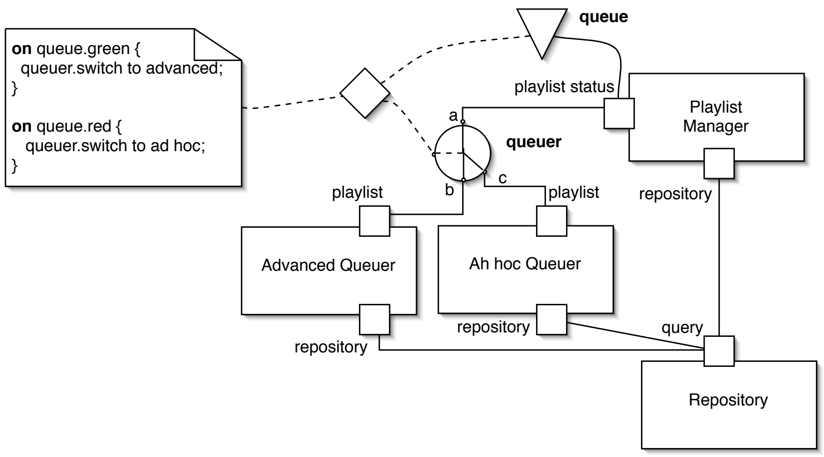
Fig. 6. Playlist component tuners

conditions. When the conditions are met, the level is entered. Adaptation recipes can be linked to be executed upon entering the level. For our purposes we defined three monitor conditions that provide an abstract representation of the constraint state, but the constraint state can be divided otherwise. The red level indicates that the involved constraint is violated and adaptation is necessary. The yellow level indicates that there are few or no constraint violations. Green indicates that the constraints are easily respected and that there may be room for inverse adaptations.

The syntax for this is as follows:

$$
\begin{array}{rcl}
\langle\text{decision level}\rangle & \longrightarrow & \textit{on}\,\langle\text{monitor clause}\rangle\ [\{,\langle\text{monitor clause}\rangle\}]\ \texttt{\{} \\
 & & \quad\{\langle\text{recipe name}\rangle\,;\} \\
 & & \texttt{\}} \\
\langle\text{monitor clause}\rangle & \longrightarrow & \langle\text{monitor name}\rangle\,.\,\langle\text{monitor condition}\rangle \\
\langle\text{monitor condition}\rangle & \longrightarrow & \textit{red} \mid \textit{yellow} \mid \textit{green}
\end{array}
$$

The exact interpretation of monitor conditions is done by the run-time infrastructure. The latter should also avoid so-called "yo-yo" effects. An example mapping for switching to the green level could be that there are no deadline violations and there is a CPU margin for which the linked recipes did not already cause violations. A complete run-time mapping algorithm is outside the scope of this paper.



```
on queue.green {
  queuer.switch to advanced;
}

on queue.red {
  queuer.switch to ad hoc;
}
```

Fig. 7. Dale queue decision maker

Figure 7 shows one of the three decision makers we have defined in the Dale application. The decision maker links the `fill queue` constraint with the queue adaptation actors. The other decision makers are created similarly. The second one links the responsiveness of the `Playlist Manager` controls to the components that access the repository frequently: the `Assistant` and `Queuer`. The third one links the `no stutter` constraint to the fader actors to control the decoding load.

5 Evaluation

The first aspect of evaluation is expressiveness. Timing monitors only monitor events that can be made visible in the software architecture, e.g. that involve the handling of messages. This is deliberate: if an event is important enough that it needs a constraint, it should be visible at this level. However, next to message events, there may be extra events that could be useful, such as the initialization of a component or redeployment. These events may be added to the event model of the timing constraints.

Although adaptation actors do not require any additional code, components may need to be rewritten to support the assumptions made in Section 4.2.

Second, we tested the run-time aspects. We have implemented our Dale case, the run-time monitors and adaptation logic in DRACO [9], a Java based component run-time platform. In terms of code overhead, the monitors and actors can be kept quite small (each less than 20 KB), as DRACO allows interception and injection of messages in an application.

To test the effectiveness of the adaptations, we ran the application with and without the adaptation activated to see in which circumstances it performed adequately. Figure 8 shows the run-time behavior of the earlier discussed component setup. We ran the tests on a 1.5 GHz PowerPC computer with Sun's Java 1.4.2 SDK. We simulated different CPU conditions (slower CPU's and different CPU loads) by slowly increasing the time to process the calls of the music repository and codec. We recorded the number of deadline violations with and without adaptation. Noticeable difference between adapted and non-adapted scenario's can be seen when the available CPU power decreases below 65 % and the adaptation actors for the fading algorithm execute the adaptation recipe `normal`. Also, starting from 45 % and less, the advanced queue algorithm cannot keep up with the requests of the `Playlist Manager` and the `queuer` message router switches to the ad-hoc queue algorithm. As can be seen, the adaptations keep the deadline violations at an acceptable level until the CPU power drops below 30 %, when the audio stream cannot be decoded fast enough anymore. Also note that the responsiveness constraint is not affected by the less powerful CPU.

As for efficiency of the mechanisms, the additional overhead is limited for CPU intensive tasks. We measured the message throughput of a connector with messages that took 5 ms to process. Adding a timing monitor decreased the throughput with less than 2 %. Similarly, the decrease in throughput of a message router is less than 5 %. A component tuner imposes no additional overhead. The decision makers are periodically activated and do not influence message throughput.

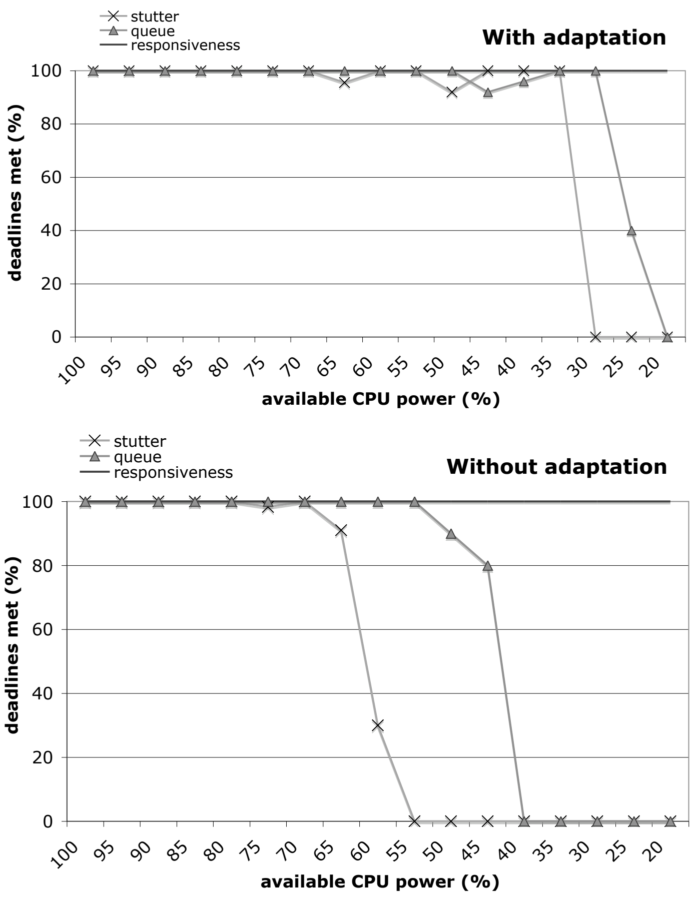


Fig. 8. Dale constraint violations with and without adaptation

6 Related Work

Traditional real-time software typically resides on a dedicated system. The timing constraints involved have been extensively tested or proven and there is little need to ensure those constraints at run-time (apart from the addition of some exceptional counter-measures). That is why the many formalisms to specify timing constraints (e.g. UML [10] and extensions [11] [12]) lack support for added run-time behavior and adaptation logic. However, timing constraints tend to be re-adopted in the larger context of Quality of Service (QoS) and resource awareness. QoS management frameworks try to integrate resource management and adaptive behavior. The Quality Object's Contract Description Language (CDL) and CQML specify changes using callback functions. Rainbow also uses low-level adaptation mechanisms but aims for reusability of abstract adaptation strategies and operators [3].

The $2K^Q$ methodology [13] offers middleware-supported adaptation by specifying component dependencies in functional graphs. From these graphs, all possible component configurations are translated. QoS adaptation is then defined and associated with transitions between component configurations. The adaptation behavior is thus somewhat hidden in the set of component configurations. $2K^Q$ suggests the use of middleware entities to recreate a new configuration.

The Quality Objects framework is perhaps the best known example of adaptive middleware. QuO specifies an architecture for implementing distributed adaptive applications; the adaptation itself however is worked out at a low level. Also, efforts have been made to package the QuO monitoring and adaptation into reusable entities called Qoskets [14]. Qoskets offer pre-defined but reusable adaptation code that can be added to CORBA objects, provided that the right wrapper code is written.

In order to tackle the development of adaptive applications, some research efforts explored the concept of QoS developer. They claim that the application developer needs help specifying and implementing adaptive QoS and propose that this work must be done by another person. The people behind the Quality Objects framework call this person a qoskateer [14]. If such a person would be necessary in a project, specifying the adaptation architecturally reduces the responsibilities of the qoskateer to a minimum.

7 Conclusions and future work

The handling of non-functional constraints such as timing is an important requirement for upcoming pervasive distributed applications. Defining constraints late in the development process may lead to the discovery of structural flaws in the architecture and entanglement of the adaptation in the functional design. We defined simple architectural constructs that have a clear goal: monitor timing constraints for component based applications and uphold them by carrying out architectural adaptations. These concepts do not offer a general replacement for domain specific adaptation solutions such as bandwidth control or grid application management, but can be used in most distributed resource-intensive or time-critical applications.

Throughout the paper we worked out an architectural run-time adaptation view of a music application we implemented. Although the architecture may need to be tailored to clearly define timing constraints and adaptation opportunities, the tailoring itself adds clarity to the architectural design. If the state and synchronization assumptions are respected, no extra code is needed to enable the adaptation actors at run-time. Although the overhead of the run-time mechanisms is limited, it is best to only define adaptations that have a significant influence on resource consumption.

The separation of constraints, decision logic and adaptation opens up possibilities to execute adaptations to uphold constraints that belong to other applications. This will be a topic for future research. Finally, it would be interesting to define adaptation actors that handle distribution. This way, resource intensive components could be migrated to uphold constraints.

References

1. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer (2003)
2. Cheng, S.W., Garlan, D., Schmerl, B., Sousa, J.P., Spitznagel, B., Steenkiste, P., Hu, N.: Software architecture-based adaptation for pervasive systems. In: International Conference on Architecture of Computing Systems (ARCS'02): Trends in Network and Pervasive Computing, LNCS (2002)
3. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer **37** (2004) 46–54
4. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems **14** (1999) 54–62
5. Mok, A.K., Liu, G.: Efficient run-time monitoring of timing constraints. In: RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97), Washington, DC, USA, IEEE Computer Society (1997) 252
6. SEESCOA Consortium: Software engineering for embedded systems using a component-oriented approach, (SEESCOA). http://www.cs.kuleuven.be/∼distrinet/projects/SEESCOA (2002)
7. van Ommering, R.: Building product populations with software components. In: ICSE '02: Proceedings of the 24th International Conference on Software Engineering, New York, NY, USA, ACM Press (2002) 255–265
8. Meijler, T.D., Schoenmaker, S., de Ruijter, E.: Modeling in an architectural variability description language. In: Proceedings of the Workshop "Planen, Scheduling und Konfigurieren, Entwerfen" (PUK2003). (2003)
9. Vandewoude, Y., Rigole, P., Urting, D.: Draco: an adaptive runtime environment for components. Appendix of the EMPRESS deliverable for Run-time Evolution and Dynamic (Re)configuration of Components (2003)
10. Berkenkötter, K.: Using UML 2.0 in real-time development: a critical review. In: Proceedings of the workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS). (2003)

11. Graf, S., Ober, I., Ober, I.: Timed annotations in UML. STTT, Int. Journal on Software Tools for Technology Transfer (2005) under press.
12. Gu, Z., Shin, K.G.: Synthesis of real-time implementation from UML-RT models. In: Proceedings of the 2nd RTAS Workshop on Model-Driven Embedded Systems. (2004)
13. Nahrstedt, K., Wichadakul, D., Xu, D.: Distributed qos compilation and run-time instantiation. In: Proceedings of IEEE/IFIP International Workshop on QoS 2000(IWQoS2000). (2000)
14. Schantz, R., Loyall, J., Atighetchi, M., Pal, P.: Packaging quality of service control behaviors for reuse. In: ISORC '02: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Washington, DC, USA, IEEE Computer Society (2002)