

Timing driven architectural adaptation

Andrew Wils, Tom Holvoet and Karel De Vlamincx
K.U.Leuven DistriNet
Department of computer science
Celestijnenlaan 200 A, 3001 Leuven
andrew | tom @cs.kuleuven.be

ABSTRACT

Computing devices are becoming computing platforms. Not the operating system and hardware characteristics will determine the footprint of an application, but the resources that are available at runtime. To function well in both high and low resource availability situations, applications have to adapt themselves at runtime.

This paper presents a light-weight mechanism for specifying timing driven adaptation at the architectural level. A key contribution is the introduction of architectural quality reflection and adaptation points. Reflection points encapsulate end-2-end monitoring of timing constraints. Adaptation points offer run-time adaptation based on component and component group reconfigurations. Because of the clear separation of reflection and adaptation, the logic that links constraint violations to adaptation actions can work on two different levels. Constraint violations can be explicitly linked to adaptation actions for inter-application adaptation, or they can trigger adaptation actions of other applications to free up resources. A special requirement of this logic is that it does not rely on resource profiling information or fine-grained resource monitoring mechanisms.

We will elaborate more on the details of these concepts and discuss their implementation and use on DRACO, a Java based component platform.

Keywords

timing constraints, architecture, adaptation, CBD

1. INTRODUCTION

Adaptation comes in many forms. It already occurs before software gets installed on a physical computing device. Software product line development [?] and platform-specific tailoring strip or add features for commercial reasons or to ensure non-functional constraints. The latter is getting increasingly difficult: supporting a multitude of platforms is no longer a matter of providing the correct binary. The heterogeneity of the target devices and the unpredictability of

the applications that run on them force us to start considering a trade-off between runtime adaptation and runtime efficiency.

Software architecture expresses the fundamental functional and non-functional aspects of the software in a number of course-grained views. Although fundamental quality requirements are expressed in the software architecture design, there currently is no support to link these to runtime adaptation mechanisms. This paper introduces an approach to express timing driven application adaptation at the component-connector view.

The usefulness of runtime adaptation is primarily determined by the extent it can influence non-functional constraints. That is why we first discuss the ways adaptation can be linked to timing constraints in section ???. We will see that existing non-architectural solutions mostly offer low-level direct link adaptation or theoretical resource brokers in which the resource behavior of all software must be completely specified. A high level strategy is then suggested, consisting of explicit violation-adaptation rules and rules that are dynamically created at runtime. It is left to the QoS platform to steer the latter, counting in factors such as utility and global resource contention.

We will discuss two architectural adaptation concepts: the quality reflection point in section ?? and the adaptation point in section ??. Reflection points describe a number of monitored QoS regions. QoS regions are operating regions indicating possible measured QoS for a (part of an) application [?]. Each region consists of a number of timing constraints and basic violation handling. Adaptation points describe the manipulation of messages and components for the following adaptation scenario's:

- component configuration (also called variability parameters [?])
- runtime component optionality and variability (for a static approach, see [?])
- semi-unanticipated changes, such as anticipated updates of new component versions

Adaptation points have regions as well, each one consisting of a particular set of adaptation actions. We elaborate on the different uses of a reflection and adaptation point, as well as their semantics.

Section ?? shows how to link reflection points with adaptation points using the strategy presented in section ?? and the earlier mentioned QoS regions. An ad-hoc algorithm is presented to illustrate how indirect links are created at

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE '2006 Shanghai, China

Copyright 2005 ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

runtime. Finally, we present the results of applying the concepts on a component based music management system in section ???. This system is a typical example of an application without any explicit real-time needs. It does however benefit greatly from specifying several timing constraints. We present related and future work and conclude in sections ??? and ???.

2. APPROACH

In the remainder of the paper, we call the mechanisms responsible for linking constraints with adaptation actions the *adaptation logic*.

Traditional real-time systems typically reside on a dedicated system. The involved timing constraints have been extensively tested, or proven, and there is little need for ensuring 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 [?] and extensions [?] [?] or formal methods [?]) lack support for added run-time behavior and adaptation logic.

However, timing constraints tend to be re-adopted in a larger context of Quality of Service (QoS) and resource awareness. QoS management frameworks try to integrate resource management and adaptive behavior. The state of the art is to make use of QoS regions (e.g. QuO [?], CQML [?] and Agilos [?]). Typically the region transition is determined by a low level measurement mechanism or specification such as system conditions [?]. These mechanisms lack general inter-application support: if an application fails to meet its deadlines, it has no hope that other applications will adapt themselves to stabilize the situation. Only the application itself can ensure its constraints.

The CRuMB resource-aware framework [?] tries a more centralized approach where every application has to negotiate their QoS region with a contract manager, but it has not yet been extended with support for timing constraints. Other approaches, such as QRAM [?] need an exhaustive amount of information to calculate a comprehensive resource strategy. E.g. the performance results of the QRAM based approach presented in [?] suggest that it is more suited to be run occasionally (e.g. only on deployment of a component) or if the user is not distracted by the calculation cycle.

We propose an adaptation logic that acts quickly on resource changes and constraint violations and that can perform basic inter-application adaptation decisions. The logic builds on the separation of monitoring and adaptation through *reflection* and *adaptation* points. Reflection points specify end-2-end monitoring of timing constraints. Figure 1 illustrates a reflection point connecting two components in a UML based component diagram. The triangle shaped point monitors messages passing through the ports to which it is connected. Violation of those constraints cause a state change in the reflection point. These changes can be linked directly to adaptation actions by connecting the reflection point to an adaptation point. Adaptation points encompass one or more associated adaptations that are grouped in QoS regions (or adaptation recipes). The direct connection is usually made within an application: at design-time, it is known what adaptation decisions can be made to uphold certain QoS constraints.

In addition to this, the adaptation logic supports inter-application logic through indirect connections. Here, a reflection point state change is linked to a request to execute

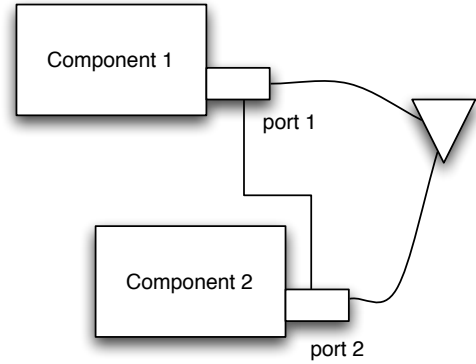


Figure 1: A reflection point connected to two components

other application's adaptation recipes in order to uphold the former's constraints. Likewise, an adaptation point can indicate that the application to which it belongs is willing to adapt to meet other application's constraints

The adaptation logic needs some heuristics to make this happen:

- a general notion of available system resources
- some measure of the utility of each application (e.g. is it actively being used)
- some measure of the utility of a QoS region (e.g. is it satisfactory or not)
- some measure of the cost of adaptations (e.g. does the user mind the adaptation)

Section ?? proposes an ad-hoc algorithm for indirect adaptation. Although indirect connections do not offer any guarantees (nor do optimal resource allocation algorithms), section ?? shows that they quickly find a suboptimal solution. First we further elaborate on the semantics of reflection and adaptation points.

3. REFLECTION POINTS

A reflection point monitors messages passing through component ports. Its graphical representation of a triangle is specified in a component-connector view such as in figure 1. The curved connections originating from the triangle determine which ports are involved in the monitoring process. Note that a reflection point is connected to ports, not a connector: the end-two-end nature of timing constraints may entail that the path between two message events is arbitrarily long. The exact semantics of the reflection point are linked to this graphical representation and are specified below. They consist of a number of QoS regions containing timing constraints and basic violation handling.

3.1 Constraints

The monitoring specification is built around timing constraints. As the discussed timing constraints are meant to be checked at runtime, we define time as it will be handled at the target platforms: a series of discrete time events

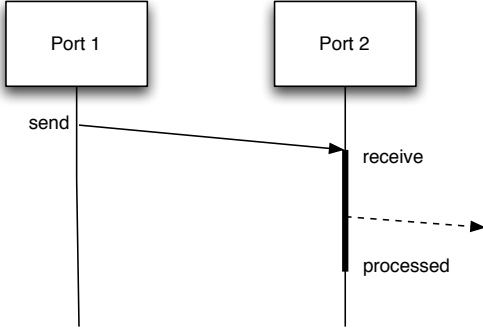


Figure 2: Message Events

(monotonously increasing) generated by a system clock. The next step is to define time-related events that can be reasoned about. We adopt the message event model that was defined in the SEESCOA project [3]. It defines 3 types of events:

send corresponds to the sending (and sending only) of a message through a port.

receive corresponds to the reception (and reception only) of a message on a port.

processed or “end of actuation”. Corresponds to the end of processing in the component. When this event is reached, there will be no more outgoing messages (send events) until a new message has been received.

Figure 2 further clarifies these events. The syntax is as follows:

$$\langle \text{message event} \rangle \longrightarrow \langle \text{port} \rangle : \langle \text{message} \rangle \text{send} \mid \text{receive} \mid \text{processed}$$

For example, `audioIn:packetreceive` would signify the arrival of the message `packet` on the port `audioIn`. Similarly, `sql:queryprocessed` means that the component to which the port `sql` belongs has processed the given query and sent out the answer to it.

In addition to these message events, we define a number of *composition events*¹:

init the composition is ready to execute (all involved components are deployed and connected)

preconf the composition is about to get reconfigured. During reconfiguration, QoS is no longer guaranteed.

postreconf the composition is reconfigured

Finally, we need a formalism that can model deadlines. For our purposes we chose a modified language based on RTL [1], mainly because this was already used in the SEESCOA [3] project that preceded this research. Our approach can be easily extended to support other formalisms. The RTL syntax uses the @ function to denote the occurrence time of

¹A composition is a collection of component instances that are usually deployed and/or reconfigured as whole. The goal of a composition is to offer a unit of functionality that is part of the workflow of the user.

a particular event. To give an example involving the earlier mentioned `sql` port, here is a constraint limiting the processing time of the request to 20ms:

$$\frac{\text{@}(\text{sql} : \text{query}_{\text{processed}})}{\text{ms}} \leq \frac{\text{@}(\text{sql} : \text{query}_{\text{receive}})}{\text{ms}} + 20$$

This particular form applies to all instances of `sql:query`. Another use of the @ function has more fine-grained control of this. E.g., the following function indicates that the time between two successive instances of `alarm:selfcheck` should be equal or less than 20ms:

$$\frac{\text{@}(\text{alarm} : \text{selfcheck}_{\text{receive}}, i + 1)}{\text{ms}} \leq \frac{\text{@}(\text{alarm} : \text{selfcheck}_{\text{receive}}, i)}{\text{ms}} + 20\text{ms}$$

Similarly, the following function indicates that the time between two successive reconfiguration instances should be equal or less than 20ms:

$$\frac{\text{@}(\text{preconf}, i + 1)}{\text{ms}} \leq \frac{\text{@}(\text{preconf}, i)}{\text{ms}} + 20\text{ms}$$

Last, we define these statistical operators:

$$\text{avg@}(e, m) = \sum_{i=1}^n \frac{\text{@}(e, i)}{n} \text{ with } n > \text{min} \quad (1)$$

$$\text{stddev@}(e, m) = \sum_{i=1}^n (\text{@}(e, i) - \text{avg@}(e, m))^2 \text{ with } n > \text{min} \quad (2)$$

3.2 Reflection Regions

Timing constraints and their violations are encapsulated in regions. A reflection point can contain an arbitrary amount of regions, of which only one is active at runtime. A reflection region declaration starts with a region block containing timing constraints. If a region is active, these constraints are monitored actively. Whenever one of the constraints is violated, the fail block specifies what to do next. Before resorting to the fail block, the QoS system may try to free up resources such as specified in section ???. This is specified by the `adapt or` phrase.

If the violation persists, the fail block provides a first step towards resolving a timing violation. First, It can try switching to another region, with other constraints and/or violation handling. possibly the application (via direct or indirect links). Second, it can alert a *delegate*, meaning the closest application instance that can handle a QoS violation. Third, the system can be notified, which may decide to stop the application altogether. Finally, a resume statement continues to evaluate the constraints in the try block. Without a switch or resume statement, the monitoring of timing constraints stops.

The syntax of this is described as follows (brackets in fixed

```

level {
  monitor
    @(selfcheck , i) < @(selfcheck , i) +
      20ms
  fail
    abort critical
}

```

Figure 3: Example level block

font represent the beginning and ending of a block):

```

⟨region block⟩ → region [⟨region name⟩] {
  ⟨monitor block⟩ [⟨fail block⟩]
}
⟨monitor block⟩ → monitor { {⟨timing constraint⟩} }
⟨fail block⟩ → [adapt_or] fail {
  [⟨switch⟩]
  [⟨alert⟩] [resume] }
⟨switch⟩ → switch [⟨region name⟩]
⟨alert⟩ → abort system | abort delegate

```

Figure 3 shows a basic region declaration that alerts the system if the timing constraint is violated.

4. ADAPTATION POINTS

Adaptation points elevate the application adaptation described in section ?? to the software architecture. It is clear that, if an application is to be mobile and adaptive, it must be developed with these characteristics in mind. Some research efforts explored the concept of QoS developer: since the application developer cannot be bothered with specifying and implementing adaptive QoS, this work must be done by another person. The Quality Objects framework calls this a person a qosketeer [?]). Adaptation points reduce the responsibilities of the qosketeer, possibly eliminating the need for such a person. Rather than linking reflection points directly with adaptation code, adaptation points encapsulate adaptation strategies that can be specified at the component-connector view. Let us first present a number of requirements for these adaptive specifications.

managed by middleware: If possible, the middleware is responsible for executing the adaptations, reducing the developer’s effort and speeding up development time.

efficiency: The user of the system should not notice adaptations. The described adaptations should therefore be easily translatable to an efficient adaptation mechanism, to keep user distraction to a minimum.

support for unanticipated evolution: Adaptations must not be limited to those that were anticipated at design-time. E.g. if a component can be replaced by others, the list of candidates must not be definite.

Being purely a non-functional construct, it is not modeled as a component and has no ports, therefore it is not depicted as a rectangle, but as a circle. Still, an adaptation

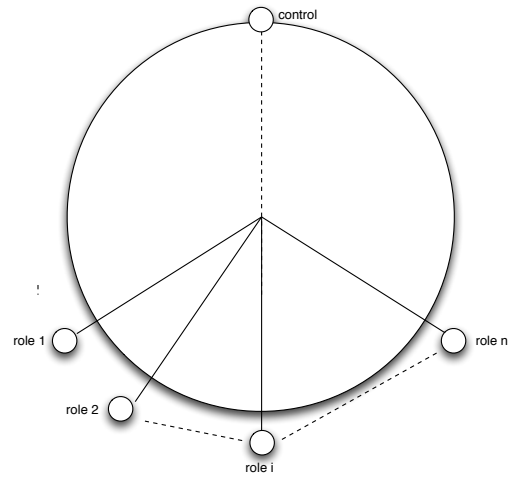


Figure 4: A standard quality adaptation point

point is meant to be connected to component port instances by means of a normal connector. The adaptation itself consists of influencing the messages sent to and from the components. The loose coupling principle of CBD ensures that the connected components are not aware of the adaptation point. It is as if the point were an n-ary connector. Figure ?? shows a basic adaptation point. Subsequent figures will further illustrate how they are to be included in component and connector instance view.

To avoid confusion with component ports, we will call the connection hooks of an adaptation point *roles*.

The actual behavior of the adaptation point is linked to this graphical representation. Like a reflection point, an adaptation point consists of a number of regions of which only one is active. A region describes a number of adaptation actions. The application adaptation types discussed in section ?? are mapped on the aforementioned adaptation actions as follows:

- component configuration or variability parameters: configure action
- runtime component optionality and variability: ignore and switch action
- semi-unanticipated changes, such as: anticipated updates of new component versions: switch action

Region declarations have a name and body and are specified as follows (curly brackets in fixed font represent the beginning and ending of the level body):

```

⟨region block⟩ → region [⟨level name⟩] {
  {⟨adaptation command⟩;}
}

```

In what follows, we describe the different adaptation point actions and how they can be used to achieve the earlier mentioned adaptations. The imperative nature of these actions makes their semantics easy to understand: the commands are executed when a region becomes active. There are many run-time issues that come with changing components and

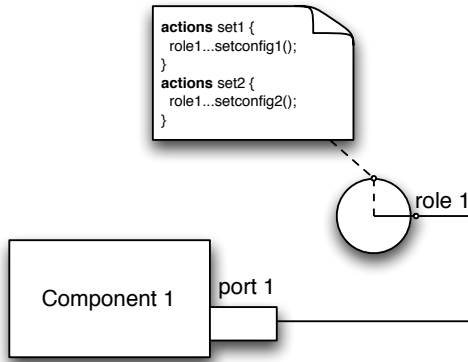


Figure 5: A configure type adaptation point connected to a component

connectors at runtime, of which adaptation points addresses a few. This will also be discussed.

4.1 Configure type

An adaptation point can be used to automatically configure one or more components. The adaptation point achieves this by sending configuration messages to regular component ports. Figure ?? shows a basic setup with 1 role.

The syntax for sending a message is rather straightforward:

$\langle \text{send clause} \rangle \rightarrow \{ \langle \text{role} \rangle \dots \langle \text{message} \rangle ([\{ \{ \langle \text{key} \rangle : \langle \text{val} \rangle \}]) \}$

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

The semantics of a configure action are quite simple as well: when a level declaration contains one or more of these DRACO send constructs and the former becomes active, the described messages are sent out through the denoted role, optionally containing primitive key-value parameter pairs. For example:

```

region {
  settings ... resolution(
    width:1280, height:768
  )
}

```

4.1.1 Pitfalls

If a region that applies to a group of adaptation points becomes active, the order in which they operate is not defined. We think it not a good idea to try to synchronize adaptation points, as this type of problem is better addressed at the functional level. That is, the application itself should be capable of the reconfiguration.

Adaptation points can be connected so that they configure more than one component. Again, caution must be used if any kind of synchronization is necessary between the configuration of the respective connected components. E.g., if the protocol between two connected components depends on their configuration, messages that rely on the older configuration may still arrive at the newly configured component (see figure ??). An application specific solution would be to

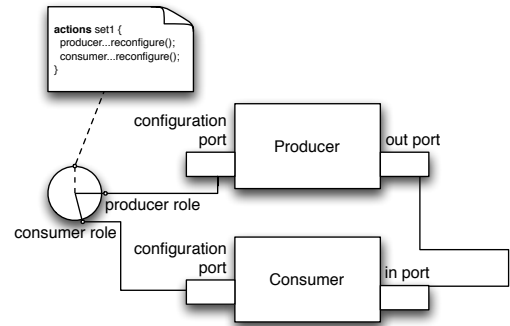


Figure 6: A dangerous use of a configure type adaptation point

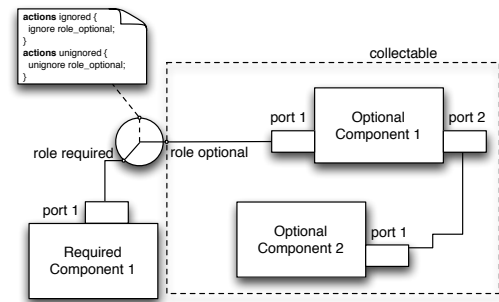


Figure 7: Ignore type adaptation point with a group of components

only configure the “producer”. The latter can then send the configuration message for the consumer in the message flow, making sure there are no miscommunications (provided that the order of the messages is guaranteed).

4.2 Ignore type

A adaptation point can be configured to ignore messages sent to and from a specific role. This behavior can be toggled as such:

$\langle \text{ignore clause} \rangle \rightarrow \text{ignore} \langle \text{role} \rangle$
 $\langle \text{unignore clause} \rangle \rightarrow \text{unignore} \langle \text{role} \rangle$

4.2.1 Pitfalls

The adaptation point general description can be annotated with a collectable role/application part. If a adaptation point role is declared as collectable, and the only link with the connected component is through the adaptation point, then the system may “garbage collect” the component if it deems it necessary to save memory. When the connection is reinstated, the component is redeployed. To extend this behavior to a group of components, the group could be identified by an enclosing rectangle, such as in figure ??.

If necessary, one could also declare a keyword to indicate that the state of the component is to be retained. The ad-

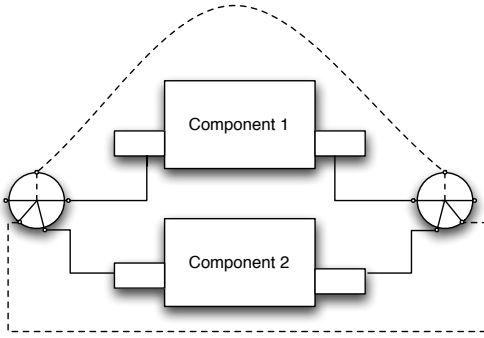


Figure 8: A twin point with 3 connected components. One component is not yet known.

vantage of this seems limited, however: remember that while the component was unreachable, the application continued to execute. The serialized state may no longer reflect the state the component would be in if it had not been disconnected.

4.3 Switch type

The behavior of an adaptation point becomes more interesting if more roles are connected. In this situation, it is useful to specify adaptation by means of changing role connections. Thus one can “switch” message flows between various components.

To enable this, we add the `link` and `unlink` clauses:

```

<link clause>  → link (<role>, <role>)
<unlink clause> → unlink (<role>, <role>)

```

A link clause ensures that all messages that are received through the first role are relayed to the second. Using this one can enable “multicast” or “router” like message flows.

4.3.1 Pitfalls

Again, care must be taken when switching between component implementations. If the message flow has a state, reconfiguration actions may need to be taken. One can solve this by combining the configure type adaptation point with the switch type.

4.4 Twin points and architectural annotations

An adaptation point can be linked directly to another adaptation point, so that a level change for one point automatically triggers the same change in the other (if a corresponding level exists). An adaptation point can be connected to a component that is not known at design-time by drawing a dotted line from it. The motives to include another component after deployment are usually to replace an existing component (e.g. for fixing bugs) or add new or improved functionality - perhaps even non-functional behavior. The adaptation point may or may not need to be updated. Because of the unclear repercussions, this construct primarily serves as an architectural annotation, and has no effects on the implementation. Both annotations are shown in figure ??.

5. ADAPTATION LOGIC

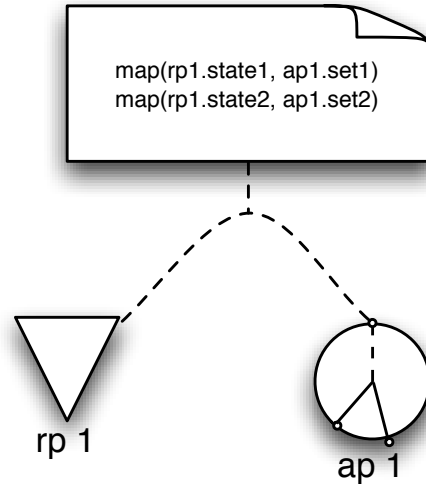


Figure 9: A direct connection mapping two region pairs.

Thusfar, we have seen how reflection and adaptation points encapsulate their behavior in regions. We chose to only link adaptation through the use of regions. This way, different types of adaptation (including adaptation that is hard-coded into the application) can be implemented without the need to extend and clutter the constraints. This section shows the actual connection mechanisms and algorithms to link the two types.

5.1 Direct connection

The semantics of a direct connection are simple: it only contains a mapping of reflection point states and adaptation point states. Figure ?? shows a simple example of mapping two region pairs.

5.2 Indirect connections

Reflection points that contain a `adapt` or `switch` statement are automatically linked indirectly. However, as explained in section ??, the indirect connection mechanism needs some extra heuristics. More specifically, we add the following adaptation point properties:

- per level: the perceived level quality. This is a value out of the enumeration { satisfactory, good, unsatisfactory }
- per adaptation point: switch cost. This is a value out of the enumeration { unnoticeable, noticeable, distracting }

Note that we have a very simple notion of utility and change cost. Indeed, while utility of a task is a very interesting benchmark for user satisfaction, it is quite hard to specify, especially in comparison with other tasks in a dynamic runtime situation². The same applies to the adaptation change cost, which has also severely been simplified.

²[2] features a global resource allocation algorithm, without going into details as to how the utility data should be obtained.

Finally, we need a global resource indicator for each type of resource.

Based on this we can create a simple adaptation algorithm such as the in listing ??.

```

//to implement..
while violation && contention {
  c = get_contention_type()
  for each s in { unnoticeable, noticeable,
    distractive }
    for each q in { good, satisfactory }
      for each adaptation point a {
        a.adaptWithin(s,q);
        if !contention(c) break to
          while;
      }

  for each adaptation point a with !a.
    partOfCurrentUserTask() {
      a.adaptWithin(s,unsatisfactory);
      if !contention(c) break to while;
    }
}

```

Listing 1: Pseudo-code for indirect adaptation

Reflection points and the above algorithm can be modified to support switching to higher QoS region in the following way. A reflection point can record the lowest resource levels for which it did not encounter the violation. If after a switch to a lower QoS region, the resource levels reach the recorded ones (minus the difference in resource usage from the two regions), an inverse adaptation may be executed.

6. IMPLEMENTATION

We have implemented reflection points and adaptation points in DRACO [?], a Java based component middleware platform.

Whereas a reflection and adaptation points conceptually link together a number of connectors, there is more than one way to implement them in the DRACO component middleware. The following implementation constructs are possible:

implementing new connectors: one could define a new type of connector with a QoRP built-in.

modifying the behavior of existing connectors: DRACO offers the possibility to influence messages as they flow through a connector. This is done by means of a chain of MessageHandlers that is processed before a message is delivered to the destination component itself.

using components: a point could be modeled as a component since a component is the basic entity that sends and receives messages.

weaving reflection and adaptation code: the functionality could be translated to application code and weaved into component code.

Weaving code reduces the reuse value of components, as modified implementations need to be stored for each use with a QoRP. We would also like to add and remove points on the fly. Implementing a new connector type requires changing the DRACO middleware implementation, an option which we did not pursue for this prototype implementation.

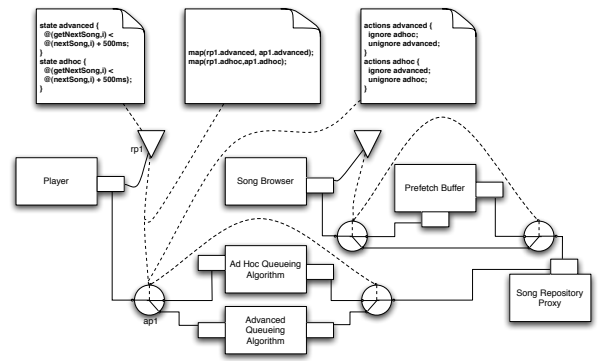


Figure 10: MusicDale component-connector view.

Reflection points can be implemented efficiently using MessageHandlers: they can be quickly inserted and removed and offer sufficient functionality to analyze message flow. However, a MessageHandler based adaptation point would have to reroute message flows and pass them from one connector to another, which is a far from elegant solution. We chose to implement reflection points using MessageHandlers and adaptation points as components. Section ?? discusses the performance repercussions of these decisions.

7. CASE STUDY

We have tested our approach on a MusicDale, a prototype application. MusicDale is a distributed music management system in which we identified several reflection and adaptation points. Basic functionality of MusicDale includes browsing, playing and sharing between a number of repositories.

To illustrate the different possible connections, we focus on a portion of MusicDale's component-connector view, given in figure ???. The repository browsing component has a reflection point constraining an information request. The point is directly connected to a prefetch buffer adaptation point. That is, if the latency of an individual request it too high, the twin adaptation point routes all messages through the prefetch component. The latter will automatically buffer subsequent requests. The second reflection point is connected directly to another twin point. The component that is responsible for finding new songs to queue can choose between an ad-hoc algorithm and a more advanced, resource intensive one. Both the reflection point and adaptation point could be indirectly connected as well.

Figure TODO: shows the behavior of this component subgroupat runtime. We have run numerous queue and browse requests on a 1.4Ghz PowerPC computer with a non-real-time Unix operating system, steadily decreasing the available CPU cycles. The graph shows the number of constraint violations and adaptations versus the relative amount of available CPU cycles. Note that the overhead is limited: the general performance hit is about 5%.

8. RELATED WORK

Table ?? summarizes the support of a number of QoS and timing mechanisms and frameworks for timing driven application adaptation.

Framework	SMIL	HQML	CQML	QuO	$2K^Q$
<i>Adaptation</i>	switching multi-media objects	reconfiguration rules	QoS transitions	QoS region transitions	component configuration transitions
<i>Mechanisms</i>	not specified	not specified	callback functions	callback functions	callback functions
<i>Efficiency</i>	good	not specified	good	good	uncertain
<i>SoC</i>	poor	poor	good	good	good
<i>Execution Support</i>	no	no	poor	poor	good
<i>Unanticipated Evolution</i>	no	no	no	no	no

Table 1: Adaptation support

Some QoS languages (e.g. QML [?] or the UML profile for Time, Performance and Scheduling [?]) do not address adaptation at all (relying on renegotiation efforts from the application's behalf). The Quality Object's Contract Description Language (CDL) and CQML specify changes by the use of callback functions. This has a reasonable expressiveness but scores badly on support for consistency and middleware support. The $2K^Q$ methodology [?] offers better middleware supported adaptation by specifying component dependencies in functional graphs. Out of 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.

SMIL, the Synchronized Multimedia Language [?] is a markup language for multimedia supporting automatic adaptation of multimedia objects. Unfortunately, its use is restricted to the specification of multimedia presentations.

9. CONCLUSION

We have defined two architectural concepts to support timing driven application. Reflection points identify timing constraints and provide a general hook that can be linked to other constructs. Adaptation points contain action recipes. Both types of points consist of regions that switch at runtime.

Our case study has shown that an application without any typical real-time constraints can still benefit significantly from identifying some basic architectural adaptations. Even more, the approach does not require expensive resource profiling.

Future work includes upward adaptation and implementing delayed instantiation and collection of alternative components. Also, more support may be given to coordinate various component reconfigurations.

10. REFERENCES

- [1] A. K. Mok and G. Liu. Efficient run-time monitoring of timing constraints. In *RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, page 252, Washington, DC, USA, 1997. IEEE Computer Society.
- [2] V. Poladian, J. P. Sousa, D. Garlan, B. Schmerl, and M. Shaw. Task-based adaptation for ubiquitous

computing. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, Special Issue on Engineering Autonomic Systems (to appear)*, 2005.

- [3] SEESCOA Consortium. Software engineering for embedded systems using a component-oriented approach, (SEESCOA). www.cs.kuleuven.be/cwis/research/distrinet/projects/SEESCOA.