

Utilising the Event Calculus for Policy Driven Adaptation on Mobile Systems

Christos Efstratiou¹, Adrian Friday¹, Nigel Davies^{1,2} and Keith Cheverst¹

¹Computing Department
Lancaster University
Lancaster, LA1 4YR
United Kingdom

{efstrati,adrian,nigel,kc}@comp.lancs.ac.uk

²Department of Computer Science
University of Arizona
Tucson, Arizona 85721
USA
nigel@cs.arizona.edu

Abstract

Adaptation is an important requirement for mobile applications due to the varying levels of resource availability that characterises mobile environments. However without proper control, multiple applications can each adapt independently in response to a range of different adaptive stimuli, causing conflicts or suboptimal performance. In this paper we present a policy driven approach for mobile adaptive systems that can overcome the aforementioned problems. Our system is based on a policy language derived from the Event Calculus logic programming formalism. Important characteristics of our policy language are the support for explicit expressions of time dependencies and the simple and user friendly syntax.

1. Introduction

Mobile environments are characterised by sudden and dramatic changes in the availability of system and network resources including connectivity, network quality of service (QoS) and power supply [3, 8]. It is therefore a primary requirement in applications designed for such environments that they support adaptive behaviour in order to continue to function effectively [9, 12].

Moreover, supporting the development of context-aware mobile applications introduces a further set of factors based on ‘situation’ that applications are required to adapt to. These factors can be, for example the physical location of the system or the proximity of other ‘interesting’ devices.

All these characteristics require the development of applications that can react to triggers received by the system (either relating to resource availability or contextual information) and adapt accordingly.

Current approaches to mobile adaptive systems are typically based on predefined courses of action implemented

within the application or supporting subsystems (middleware or operating system). Indeed, most adaptation support mechanisms are specialised on one single adaptation domain (e.g. network QoS based adaptation) allowing modification of the adaptive behaviour of applications within the boundaries of that specific domain.

In this paper we argue that in order to successfully adapt in response to environmental or contextual changes, applications must respond in a coordinated ‘system-wide’ manner. Failure to consider such issues may lead interference between separate applications’ adaptation mechanisms co-existing on a given system. Moreover, the lack of a mechanism that oversees the interdependencies of adaptive applications can potentially lead to conflicts, suboptimal performance and unwanted hysteresis effects [5, 6].

In this paper we present a middleware platform that has been designed specifically to provide support for adaptation both within and between mobile applications. Coordinated behaviour is achieved on a system-wide level by separating policy from mechanism through the use of a per-host policy-driven adaptation controller. More specifically, the important characteristic of our approach is the decoupling of the adaptive mechanisms supported by each application and their adaptation policies directing when and how adaptation should take place. All adaptation policies of each host are handled by an adaptation control module based on a policy language derived from the Event Calculus [20]. Our approach allows sharing of application status information among all applications running on the system. The introduction of a policy language that handles adaptation allows the dynamic modification of the adaptive behaviour in order to overcome potential conflicts or to better satisfy the user needs.

The paper is structured as follows. In section 2 we discuss the motivations for building our platform using a range of application scenarios. The section concludes with a set of requirements that are the basis for our approach. Section 3 provides an overview of the main components of our

adaptation platform. In section 4 we describe the policy language that is used by our platform, based on the Event Calculus formal specification language. The section gives a detailed description of the design concerns in the implementation of a dynamic policy evaluation mechanism and the main issue of how time is handled within the language implementation. Section 5 presents our currently implemented prototype and section 6 considers our work in relation to other existing and related approaches. Finally, section 7 contains our concluding remarks and pointers for future work.

2. Motivation

As the need for adaptive context-aware applications increases, the case where multiple adaptive applications triggered by several adaptation and contextual triggers operating on the same end-system will be a common scenario. Our experience with the development of mobile adaptive applications has led us to believe that it is necessary to provide a certain level of control over the adaptation procedure in order to achieve a harmonious operation of the system in such environments. In this section we present an analysis of the problems that can occur in current adaptive systems and conclude with a set of requirements for our approach.

2.1. Defining the Problem

Early approaches to supporting adaptive mobile systems were mainly concerned with the impact of the variations in network QoS due to the wireless communication links [9, 11, 17]. More recent approaches consider other attributes that can become triggers for adaptation, such as power supply [7, 8]. So far all these adaptation mechanisms operate in isolation with little or no concern about their interdependencies, though the requirement for such a concern has been identified in the past [8].

Consider the following illustrative scenario: a single application is ‘triggered’ to adapt by the power adaptation mechanism in order to reduce its power consumption. The application reduces the use of the network in order to reduce the amount of power drawn by the network interface. Across the system as a whole, this reduction of the network usage is discovered by the network adaptation mechanism as a net increase in available bandwidth. As a result another application (or even the same one) is triggered to increase the use of network so as to provide higher level of service. In this scenario it is clear that the application of one adaptive mechanism (power) is the direct cause of another conflicting adaptation (use of network resources). Generalising from this behaviour, we can postulate that the isolation of adaptation mechanisms can lead to conflicting or unstable situations.

As we have stated, a characteristic of many current adaptation approaches is that adaptive behaviour is either hard-coded within the application, or implemented within a middleware platform based on predefined adaptation patterns. For instance, commercially available adaptive applications, such as video players [19], may adapt the quality of video playback in response to changes in end-to-end network QoS. Such behaviour is intrinsic to the specific video application and cannot be generalised. It is further evident that the majority of context-aware applications have been implemented in an ad-hoc manner [1, 18]. Such applications are clearly unaware of other applications operating on the system and cannot take into account any side-effects that their reaction may have on the rest of the system. In adaptive middleware designed to support mobile systems [11, 17, 3] there is a certain level of awareness of multiple applications operating on the system. The level of control over these applications however, is limited to the predefined adaptation policies built into the applications, the underlying middleware platforms or the operating system itself.

In more detail, adaptation is driven by the applications’ requirements on the available resources. The underlying adaptation mechanism is responsible for meeting these requirements or triggering the applications whenever the available resources are not enough to satisfy their demands. Whenever an application is triggered due to change in the level of available resources they have to adapt to the new situation. However, the actual decision of the appropriate reaction to such a trigger is taken by the application without any knowledge about coexisting applications. Furthermore, the hard-coded adaptation behaviour patterns built-in either the applications or the middleware does not allow dynamic modification of the systems’ behaviour according to varying situations or special user requirements. The following scenario illustrates this issue:

Let’s consider an adaptive streaming video player and an adaptive web-browser performing a download of a large file, both operating on the same host. In this case both network bandwidth and power supply are finite resources. In such a scenario there are a number of possible adaptation configurations that could be applied:

- The web-browser reduces the bandwidth usage allowing the video player to request a less compressed/higher bandwidth stream so as to reduce the power consumption due to stream decoding while the power consumption on the network card remains the same. This strategy would best suit the case where the user considers the video to be of higher importance than the timely completion of the file transfer.
- A second strategy might be for the video player to request only the soundtrack of the video sequence (reducing both network bandwidth and power consumption).

tion due to decoding the video). The bandwidth saved by the video player can be reused by the web browser to increase the speed of the file download. Such a strategy would be appropriate if the user needed the file more urgently than the full video feed.

- A third possibility would be for both applications to degrade their service in order to maintain power consumption below a specific threshold. This strategy would be the most appropriate option if extending the battery life of the system is the highest priority.

All of these options could be required depending on the user needs and the specific situation. As stated earlier, the hard-coded adaptation mechanisms implemented in many current systems does not allow this level of dynamic modification of the adaptation behaviour across the system.

2.2. Requirement for policy driven adaptation

In order to look in more detail at how the limitations of current approaches can be overcome, it is first necessary to examine the fundamental mechanisms supporting adaptation. Adaptation can be viewed as a mechanism that includes three distinct functional elements. The first element concerns the monitoring of a specific source of information that is ‘interesting’ for that adaptive mechanism. This information source could be the availability of a specific resource such as the power supply or a contextual trigger such as the system’s physical location.

The second element concerns the policy that describes how this mechanism should react according to the information being monitored. This policy could, for example, state that when the power supply drops below a specific threshold then a reaction is necessary.

The third element is the actual adaptation mechanism that performs the specific adaptive action as directed by the policy. For example, an adaptive mechanism might reduce the network bandwidth consumed by the application. This last element may also have an impact on the initial source of information, e.g. to change the rate that the available power drops. This last link between the adaptive mechanism and the initial resource being monitored does not necessarily exist in all systems. Most context-aware systems for example do not affect the initial resource that triggered their change of behaviour.

Though a simple adaptive system follows this dependency cycle, in a system where several applications or multiple triggering attributes exist, the dependencies span across several applications and information sources. As seen in the previous example, an adaptive mechanism that is triggered in order to reduce the level of power consumption may have a side effect on the level of available network bandwidth of the system. These side effects are the main

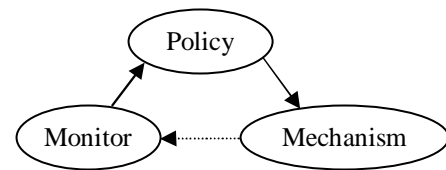


Figure 1. Basic adaptation cycle

cause of conflicts. We believe that in order to detect such conflicts it is necessary for all adaptive applications to externalise their adaptive mechanisms allowing the system to determine possible interdependencies and oversee potential conflicts.

Once a conflict or undesirable situation has been detected, either by the system or by the user, it is necessary to allow modifications to the adaptive behaviour of the system in order to resolve the situation. However, in most current adaptive applications the adaptation policies are not distinct elements within the adaptive cycle. In most cases adaptation policies are hard-coded within either the monitoring process or the adaptive mechanism, making adaptive conflicts hard to detect. To allow the necessary level of control over the behaviour of the system it is our contention that policies must be decoupled from the adaptation mechanisms themselves. Moreover, these policies should be defined in a language flexible enough to allow the specification of conditions that can include multiple triggering events that may take place over time.

In this paper we present an approach based on such a policy language. Adaptation decision mechanisms use the policy language to allow the active involvement of the user in the modification of the systems adaptive operation. Furthermore, the user can set up the system to behave according to his/her needs and dynamically modify its behaviour when these needs change.

Summarising the findings of the above analysis we can conclude with a set of requirements for a general purpose adaptation support platform that should be able to overcome the highlighted issues:

- **Application control and coordination.** Current adaptation approaches provided limited support in terms of coordinating the behaviour of multiple applications to achieve a user required goal or overall improved performance. In order to support flexible and coordinated adaptation there is a requirement for the triggering of adaptation to be handled in a cross-application level. Given this approach, the decision about when and how an application should adapt is pushed out of the applications’ boundaries while the adaptive behaviour is still part of the applications’ characteristics.

- **Support for Flexible Adaptation Policies.** A further requirement is to support the notion of policy driven adaptation. Policies can be specified externally from applications and should be described in a flexible manner to allow the incorporation of multiple adaptation triggers over time. Moreover, the externalisation of such policies allows for the system to determine the coexistence of adaptive applications that may interfere with one another. The policy language that will be used should not be bound to a specific adaptation domain – it is desired that the language specification should offer a certain level of abstraction that will allow a broad space of adaptation policies to be specified.
- **User involvement.** The user is a principle component in any adaptive systems. More specifically, the user is the only one that can judge whether the behaviour of the system is satisfactory. Given this fact, the system should allow the user to oversee the interdependencies of all adaptive applications within a system. Moreover, the user should be able to identify any possible conflicting, unstable or just undesired behaviour. Furthermore, the user should be able to configure the overall behaviour of the system according to their current needs. Generalising this requirement it is necessary to break the transparency enforced by current adaptive systems offering user awareness and involvement in respect to adaptation.

In the next section we present our prototype platform that is designed taking into account these requirements.

3. Our Adaptation Platform

The realisation of the system requirements outlined above requires the design of a platform in which adaptive mechanisms and policies are decoupled. Furthermore, mechanisms must be ‘exposed’ or externalised in order to enable coordinated control by an ‘adaptation controller’. Our adaptation platform has been designed to address these requirements. Figure 2 shows the relationship between the main components of our platform. The functionality of the platform can be split into two main areas:

1. the sharing of application state information and their available adaptive mechanisms
2. the coordination of the behaviour of the applications according to the adaptation policies

In more detail, the platform allows applications and tools monitoring changes in the systems’ environment to provide a description of their functionality including a set

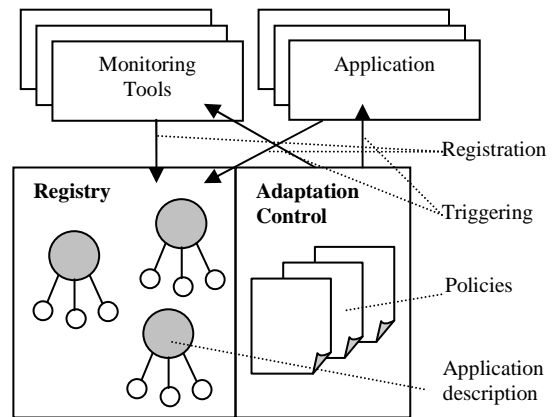


Figure 2. The overall architecture

of state attributes containing information on the environment/application and adaptive actions that they can perform. These descriptions are handled by the platform’s registry.

The adaptation control is the component that is responsible for making decisions about the appropriate adaptive actions that the applications should take. These decisions are made based on the set of policies exported by the applications or established by the user.

3.1. Registry

Each adaptive application that is running on a host must register with the platform’s registry. The application registry holds a description of each application’s available adaptation mechanisms as well as a set of state variables that provide information about each application’s current state. The registration information is provided by the application in XML¹ (a sample of which is presented in figure 3). The structure of the registration information has been influenced by the design of the UPnP protocol [16].

The registration information is divided into two sections. The first section provides a list of the available adaptation mechanisms that are supported by the applications. This description allows the platform to invoke the appropriate adaptation mechanism whenever the application is required to adapt. The description of an adaptation method may specify relationships between the attributes passed to the method when invoked and the state variables within the application. The semantics of such a relationship denotes the effect an adaptive mechanism may have on the state variables of the application. More specifically, each of the attributes passed to an adaptation mechanism may relate to the new value a

¹The XML schema for the application registration can be found in <http://www.comp.lancs.ac.uk/~efstrati/platform/api-schema.html>

```

<application>
  <name>WebBrowser</name>
  <uniqueId>1234</uniqueId>
  <methodList>
    <method>
      <name>SetBand</name>
      <attributeList>
        <attribute>
          <name>bandLimit</name>
          <relatedVariable>netBandwidth
            </relatedVariable>
        </attribute>
      </attributeList>
    </method>
  </methodList>
  <stateVariableList>
    <stateVariable>
      <name>netBandwidth</name>
    </stateVariable>
  </stateVariableList>
</application>

```

Figure 3. Sample of XML description for an adaptive application

state variable will take after the invocation. Multiple affected state variables by an adaptation mechanism can be specified through the definition of multiple attributes passed to the adaptation mechanism.

The second section provides a list of ‘state variables’ that represent the current state of the application. Based on this information the registry creates a copy of these variables within the platform. The values of the variables are updated by the application by sending events to the platform whenever a state variable changes. This provision of state information by the applications allows the platform to monitor all applications within the system and consider their state before taking adaptation decisions.

As described in section 2.2, the general adaptation cycle includes the monitoring of a set of information sources that may trigger adaptation. The registration protocol described here provides a general mechanism for the introduction of such information sources to the platform. Any information source acting as an adaptation trigger can be served by a monitoring tool registered with the platform. The architecture is thus extensible; information sources may include attributes relating to the state of system hardware (network, power etc.), but also application specific information sources (such as context monitors). The specific information that the platform requires for triggering adaptation can be accessed through the state variables that each monitoring tool defines. In our current prototype implementation two monitoring tools have been defined that act as wrappers for the network interface and the battery level of the host. Both of these tools register with the platform and provide details

about the state of these devices through their state variables.

3.2. Adaptation Control

The adaptation control module is responsible for monitoring the behaviour of all applications on a single host and triggering them to adapt according to adaptation policies registered with the platform.

The behaviour of the adaptation control module is controlled by events fired by the applications when the values of their state variables change. These events act as triggers for the adaptation control module in order to apply an adaptive mechanism as described by the adaptation policies. Applications register ‘default policies’ with the platform that specify their default adaptive behaviours. In practice however, the set of active policies may include modifications or additional policies provided by the user.

Part of the adaptation control module’s functionality is to provide a dynamic policy evaluator for the adaptation policies. The details of the policy language and the evaluation mechanism used are presented in the following sections.

4. The Event Calculus as a Policy Language

In this section we will provide a detailed description of the policy language used in our platform. This policy language is based on the event calculus formal specification.

4.1. The Event Calculus

The event calculus was introduced by Kowalski and Sergot [13] as a logic programming formalism for reasoning about events and change. The work presented here is based on a simplified version of the event calculus that was presented later by Kowalski [14].

The event calculus provides a theoretical framework where it is possible to reason about events and their effects in an event-driven system. The event calculus is defined over a set of entities, namely events that take place at specific time points and fluents that represent the effects of the events. A fluent represents a specific situation that has a timed duration, for example a state like ‘battery is low’. When the system under consideration gets into that specific condition the fluent is considered to be valid (it holds). The state of fluents are defined according to events that can initiate or terminate them.

Along with the basic entities of events and fluents, the event calculus defines a set of predicates that allow the specification of propositions about when specific events take place and what the state of fluents are. More specifically the basic predicates defined in event calculus are:

$\text{Initiates}(e, f, t)$: Fluent f is initiated by event e at

time t .

$\text{Terminates}(e, f, t)$: Fluent f is terminated by event e at time t .

$\text{Happens}(e, t)$: Event e occurs at time t .

By using these predicates we can ask about the validity of some fluents at particular time points. More specifically, the simplified event calculus defines the following additional predicates:

$$\begin{aligned} \text{HoldsAt}(f, t) \Leftarrow & \exists e, t_1 [\text{Happens}(e, t_1) \wedge t_1 < t \wedge \\ & \text{Initiates}(e, f, t_1) \wedge \\ & \neg \text{Clipped}(t_1, f, t)] \end{aligned}$$

$$\text{Clipped}(t_1, f, t_2) \Leftarrow \exists e, t [\text{Happens}(e, t) \wedge t_1 < t < t_2 \wedge \text{Terminates}(e, f, t)]$$

$$\text{Declipped}(t_1, f, t_2) \Leftarrow \exists e, t [\text{Happens}(e, t) \wedge t_1 < t < t_2 \wedge \text{Initiates}(e, f, t)]$$

The *HoldsAt* rule states that a fluent is valid at a specific time point t if an event e exists that initiated this fluent at an earlier time and this fluent has not been terminated during this time. The *Clipped* and *Declipped* rules state that a fluent has been terminated or initiated respectively by an event that took place within a time period.

Based on this small set of rules the event calculus allows us to define an event based system that changes as events take place. In addition, we can use the available rules to ask about the validity of specific conditions of the system and the times that these conditions are valid.

4.2. The Policy Language

In order to understand the suitability of the event calculus for our platform we have to consider some of its basic characteristics. Firstly, the operation of the platform is driven by events. The input given to the adaptation control module is the set of state variables reported by the applications. As described in section 3.1 as the values of these state variables change, the corresponding applications generate events that notify the platform about their new values. The event calculus, by definition, embodies the eventing mechanism within its specification.

Secondly, the platform is required to handle the conditions under which an adaptive reaction should take place in a uniform manner, irrespective of the type of adaptation. The decision mechanism should be a general purpose mechanism that will handle adaptation policies relating to a variety of adaptation types such as network based adaptation or adaptation related to physical context. The event calculus is general enough to allow the specification of rules for *any* type of event based system.

Thirdly, the policy language must be flexible enough to allow policy specifications that consider multiple events and time relationships among these events. Time relationships in particular, are important for the fine-tuning of the system and for overcoming instabilities or conflicts. Time is a fundamental element of the event calculus, allowing a high level of flexibility in the specification of time relationships.

Therefore, deriving from the specifications of the event calculus, we define the event calculus policy language in which policy rules are formulated as event-fluent-condition-action sets in a form similar to policies specified in *PDC* [15].

Specifically, each policy rule is comprised of a set of system specific event definitions, a set of fluents controlled by the events, a condition body and an action body. The basic operation of a rule is to perform the actions defined in the action part if the condition part evaluates to true. The condition part consists of a logical expression involving the occurrence of events or the current state of fluents. Each fluent expresses a specific situation that the rule is interested in. The situations expressed by fluents are directly controlled by the defined events.

For example, let's consider a policy specifying that the network connection should switch to GSM when the user is outdoors. An informal way to describe this is:

```

Events LeftHome, LeftOffice, InHome, InOffice
Fluent Outdoors :
    initiated by events: LeftHome, LeftOffice
    terminated by events: InHome, InOffice
Condition :
    Initiated(Outdoors)
Action:
    Switch network to GSM
  
```

As described in this example, the fluent *Outdoors* is controlled by the events denoting when the user leaves or enters areas that the network connection should not be GSM. The condition part evaluates to true at the time the fluent is initiated and the action part is executed.

Formally speaking a policy rule is an expression of the form:

$$\begin{aligned} & \text{event definition}_1 \\ & \dots \\ & \text{event definition}_n \\ & \text{fluent definition}_1 \\ & \dots \\ & \text{fluent definition}_m \\ & \text{condition } \{ \text{condition} \} \\ & \text{action } \{ \\ & \quad \text{action}_1 \\ & \quad \dots \\ & \quad \text{action}_k \\ & \} \end{aligned}$$

```

event lowBand :- NetworkInterface.availableBandwidth < 19200
event normBand:- NetworkInterface.availableBandwidth >= 19200
fluent inLowBand {
  initiates(lowBand)
  terminates(normBand)
}
condition {
  initiates(lowBand, inLowBand, t1) and
  not clipped(t1, inLowBand, t2) and
  t2 > t1 + 30
}
action {
  WebBrowser.LowBand()
}

```

Figure 4. A sample policy rule

Definition 1 An event symbol e represents the occurrence of an event as described by the event definition. The event definition is an expression of the form:

event e :- l

where e is an event symbol and l is a system specific logical expression. The logical expression is of the form $p_1\theta p_2$ where

1. θ is a Boolean operator from the set $\{\mathbf{and}, \mathbf{or}\}$ and p_1, p_2 are logical expressions as well, or
2. θ is a relation operator from the set $\{=, !=, <, <=, >, =>\}$, p_1 is a system specific attribute and p_2 is a constants of the same type. It is assumed that the user has access to the set of available system attributes that can be used for the definition of the logical expression.

As highlighted in definition 1, the user is assumed to have access to the set of system attributes that can be used for the definition of events. In our system these attributes are the application state variables reported by the adaptive applications running on the system during registration with the platform (described in section 3.1). The specification of such an attribute is represented by an expression of the form:

$a.v$

where a represents the application running on the system and v one of its state variables. An event, for example, specified to mark the time the network bandwidth is between 19.2Kbps and 64Kbps is defined as:

```

event normBand:- (NetworkInterface.Bandwidth > 19.2)
and (NetworkInterface.Bandwidth < 64)

```

Definition 2 The occurrence of an event is defined through the predicate $\mathit{happens}(e, t) \rightarrow \{\mathbf{true}, \mathbf{false}\}$ where e is an event symbol and t is a time point. Predicate $\mathit{happens}$

evaluates to true iff t is the time point the logical expression l specified by the event definition transits from false to true.

The $\mathit{happens}$ predicate should be interpreted as “the logical expression defined for event e has changed its value from false to true at time point t causing the event to take place”.

Definition 3 A time point is a positive integer that represents a specific point in time.

In our system, time points are considered to represent time in seconds. However, the granularity for the representation of time within a policy system is an issue that depends on the requirements of each implementation.

Definition 4 A fluent symbol f represents the state of a fluent as described by the fluent definition. The fluent definition is an expression of the form:

```

fluent  $f$  {
   $init_1$ 
  ...
   $init_n$ 
   $term_1$ 
  ...
   $term_m$ 
}

```

where f is a fluent symbol and each $init_i$ is an expressions of the form $\mathit{initiates}(e)$ where e is an event symbol representing the event that initiates the specific fluent; and each $term_i$ is an expressions of the form $\mathit{terminates}(e)$ where e is an event symbol representing the event that terminates the specific fluent. The order in which the initiates/terminates expressions appear is not significant.

A fluent is considered to hold for the time period between its initiation and termination including the termination time and it does not hold for the time period between termination and initiation including the initiation time.

A fluent in the policy language does not relate to any value within the platform itself. It is an abstract entity that can be defined according to the policy author’s requirements. The purpose of a fluent is to represent entities that have time duration and their state changes according to the occurrence of events. In practice a fluent usually represents a real situation of the system’s behaviour (like for example operating in a low bandwidth state as shown in figure 4).

As Definition 4 describes, the state of a fluent is controlled by the events that initiate or terminate the fluent. We have to make clear the distinction between the statements $\mathit{initiates}(e)$ and $\mathit{terminates}(e)$ defining a fluent from the predicates $\mathit{initiates}(e, f, t)$ and $\mathit{terminates}(e, f, t)$ that are defined later on.

Definition 5 The condition is a logical expression of the form

1. $p_1\theta p_2$ where θ is a Boolean operator from the set $\{\mathbf{and}, \mathbf{or}\}$ and p_1, p_2 are condition expressions as well, or

2. a predicate proposition of *initiates*, *terminates*, *holdsat*, *happens*, *clipped*, *declipped* and their negations, or
3. a logical expression of the form $t_1 \theta t_2$ where θ is a relation operator from the set $\{=, \neq, <, \leq, >, \geq\}$ and t_1, t_2 are time variables or expressions representing time points.

The body of a condition specifies the logical expression that should be evaluated in order for the action part to be executed. Within the condition body a policy rule may include combinations of predicate propositions and time relationships.

Definition 6 The *initiates/terminates* proposition is an expression of the form:

$$\mathbf{initiates}(e, f, t) / \mathbf{terminates}(e, f, t)$$

where e is an event symbol, f is a fluent symbol and t is a time variable. This proposition is true iff **initiates**(e)/**terminates**(e) is part of fluent's f definition, **happens**(e, t) is true and the fluent does not hold (hold for terminates) at time t .

These predicates allow the specification of queries in relation to the initiation/termination of fluent. They should be interpreted as “the event e initiated/terminated fluent f at time t ”.

Definition 7 The *holdsat* proposition is an expression of the form:

$$\mathbf{holdsat}(f, t)$$

where f is a fluent symbol and t is a time variable. This proposition is true iff there is an event e_1 for which **initiates**(e, f, t_1) is true and $t_1 < t$ and for every event e_2 and time point t_2 , $t_1 < t_2 < t$, **terminates**(e_2, f, t_2) is false.

The *holdsat* predicate allows the specification of queries in relation to the actual state of a fluent. The predicate should be interpreted as “fluent f holds at time t ”.

Definition 8 The *clipped/declipped* proposition is an expression of the form:

$$\mathbf{clipped}(t_1, f, t_2) / \mathbf{declipped}(t_1, f, t_2)$$

where f is a fluent symbol and t_1, t_2 are time points and $t_1 < t_2$. This proposition is true iff there is an event e for which **happens**(e, t) is true and $t_1 < t < t_2$ and **terminates**(e, f, t)/**initiates**(e, f, t) is true.

The *clipped/declipped* predicates are used for specifying queries about the initiation or termination of a fluent within a specific time range. The predicates should be interpreted as “fluent f has been terminated/initiated sometime within (t_1, t_2)”

Definition 9 An action is a statement of the form:

$$a(t_1, \dots, t_n)$$

where a is an action symbol with n arguments and each t_i is a parameter of the appropriate type.

An action statement represents a call to a specific adaptation method of an application as defined by the applications by their registration. An action call triggers an application to adapt when the condition part of the policy evaluates to true.

In a more informal way, each rule of the policy language consists of two main parts: a condition and an action. The condition is a logical expression that can evaluate to true or false. The action is a list of calls to adaptation methods that should be performed only if the condition evaluates to true.

Each condition is further divided into two parts: the declaration part and the condition body. The declaration part defines the events and fluents that participate within the body of the condition. The body itself consists of a logical expression combining Boolean operations (and, or, not) and the predicates specified by the event calculus.

The declaration of an event specifies when an event is considered to have occurred in relation to the values of specific application state variables. As shown in figure 4 the event *lowBand* is considered to have taken place when the state variable *availableBandwidth* of the application *NetworkInterface* has taken a value below 19.6Kbps. A fluent declaration is done by specifying all the events that can be initiated and terminated by.

The condition body consists of a logical expression using the event calculus predicates. This logical expression can use predicates to evaluate the time specific events take place or whether a fluent holds or does not hold. The condition body can include time relationships between time variables that correspond to the time at which specific predicates evaluate to true. The policy author can thus specify not only the events and fluents that affect a condition, but also the time relationships between these predicates. As presented in figure 4 the body of that condition specifies that it will evaluate to true only if the fluent *inLowBand* has been initiated at a time t_1 and has remained valid until time $t_2 > t_1 + 30$. In essence, this rule specifies that it evaluates to true if the systems' available bandwidth has remained below 19.6Kbps for more than 30 seconds.

The last part of a policy rule is the list of actions. Within the list of actions the policy author has to specify a sequence of adaptation methods that should be invoked by the platform when the condition of the rule evaluates to true.

4.3. Policy Evaluation

An important requirement for the evaluation of the policies is that the evaluation should take place progressively as the values of state variables change over time. Specifically, there is no easy way to discover when all necessary information is available in order to evaluate a rule in

one step. Rather, the evaluation of a rule must take place in stages as information about the state variables become available. More specifically, as events take place some predicates within the body of the condition may evaluate to true, whilst others are false, awaiting future events that may change their value. Even while all predicates may have evaluated to true at specific time points, the time relationships may still not be satisfied. As a consequence, the policy evaluation mechanism should progress incrementally as events take place and allow the execution of the actions only at the time that the whole condition body has been satisfied.

One of the important characteristics of the policy language is the fact that time variables receive their values implicitly through the evaluation of the predicates in which they participate. A predicate such as **happens**(e, t) evaluates to true when event e occurs. When a predicate becomes true, the time variables participating receive their values according to the semantics of the predicate. In the **happens**(e, t) example, variable t will take as its value the time that event e took place.

Even though a time variable within the policy language represents a single time point, there are some predicates that can evaluate to true when the related time variables take any value from a range of available time points. Predicate **holdsat**(f, t) for example can be true for any value of t within the range of values from the time the fluent f has been initiated until it was terminated:

$$\forall t, \mathbf{holdsat}(f, t) \Leftrightarrow \exists t_1, t_2 [t_1 < t_2 \wedge \mathbf{initiates}(e_1, f, t_1) \wedge \mathbf{terminates}(e_2, f, t_2) \wedge \neg \mathbf{clipped}(t_1, f, t_2) \wedge t \in (t_1, t_2)]$$

This fact forces us to represent all time variables within the evaluation mechanism as *ranges of values* that can evaluate a specific predicate to true. The case of a single time point value, as produced by predicates like **happens**(e, t), are special cases where the time range has the same start and end point. Table 1 offers a list of the rules that define the values that time variables receive during the evaluation of predicates.

The representation of each time variable as a time range, has a direct impact on the way time relationships are evaluated. Though a logical expression between two variables should result in a comparison of the values that can either be true or false, in our case, any logical expression among time variables results in a modification of the ranges that a time variable covers in order to make this expression valid. Let's consider the common case of equality between two time variables. The result of such an expression will be to assign to both variables a new value range that includes the common sections of the two initial time ranges of the two variables:

$$t_1 \in (a_1, a_2) \wedge t_2 \in (b_1, b_2) \wedge t_1 = t_2 \Rightarrow t_1 \in (a_1, a_2) \cap (b_1, b_2) \wedge$$

Table 1. Time value assignment through the evaluation of predicate.

Predicate	Variable	Description
initiates (e, f, t)	$t = a$	a : the time that e took place
terminates (e, f, t)	$t = a$	a : the time that e took place
happens (e, t)	$t = a$	a : the time that e took place
holdsat (f, t)	$t \in (a, b)$	a : the time that f was initiated b : the time that f was terminated or the current time if it still holds
clipped (t_1, f, t_2)	$t_1 \in (0, a)$ $t_2 \in (a, b)$	a : the time that f was terminated b : the current time
declipped (t_1, f, t_2)	$t_1 \in (0, a)$ $t_2 \in (a, b)$	a : the time that f was initiated b : the current time

$$t_2 \in (a_1, a_2) \cap (b_1, b_2)$$

Similar rules can be defined for operations like “less than” and “greater than”.

Therefore the algorithm for the evaluation of the policy language can be summarised in the following steps:

1. As state variable values change, the evaluator determines which of the event expressions in the policy are to be marked true or false.
2. When the occurrence of an event has been identified, the state of all the related fluents is determined, based on the fluent definitions (initiate/terminate lists).
3. The state of the predicates within the rule's body is evaluated progressively as events take place and fluents modify their state.
4. For each step where a predicate receives its value, the corresponding time variable is assigned a time range that makes this predicate true.
5. The time relations defined within the policy rule are evaluated so as to find whether there are any possible values for the time variables that can evaluate both the predicates and the time relationships to true.

6. If all conditions have been evaluated to true, the action list is executed.
7. If at any stage the values of the time variables do not satisfy the time relationships, the incremental evaluation of the rule is canceled and all variable values are reset to null.

4.4. Efficiency

An important issue in the evaluation of the event calculus policy language is the efficiency of the evaluation mechanism. Since the evaluation procedure is driven by the update of state variables reported by the adaptive applications, it is important to reduce the number of times that false evaluation attempts take place due to a state variable update. Significantly, as the definition of policy conditions may include time relationships between predicates, these time relationships can provide a means for scheduling the incremental evaluation of a policy rule at time points that are derived by the policy condition.

As expressed in definition 1, each policy event is defined through a logical expression evaluating over the state variables of applications. This expression represents a range of values the given state variables may take, causing the policy event to take place. However, an application may update the values of the state variables manipulated by the platform even when these updates do not cause any policy events to take place. Therefore, in order to reduce the number of state variable updates performed by the applications, these ranges of values specified by the policy definitions are passed over to the application. This way the applications will update the values of their state variables only when the new values change the state of the logical expressions defined by the policy events. For example, in the sample presented in figure 4, the NetworkInterface monitoring tool will only update the values of the availableBandwidth state variable when the value of the variable changes from less than 19200bps to more than 19200bps or vice versa.

As described in definition 5, the condition part of a policy rule may include expressions that define relations among time variables. The evaluation of such expressions can only be performed when all time variables involved have been assigned a value or a range of possible values through the evaluation of the predicate in which they participate. However, there may be cases where it is possible to infer a possible value for an undefined time variable. For example:

$$t_1 = a \wedge t_2 = t_1 + 10 \Rightarrow t_2 = a + 10$$

where a is the value t_1 has received by the evaluation of a predicate.

This conjecture can be used for the scheduling of the evaluation of a predicate that relates to this time variable. For example, let's consider the following condition body:

happens(e, t_1) and
holdsat(f, t_2) and
 $t_2 = t_1 + 10$

Let's suppose that event e takes place at time point a . Through the evaluation of the relation between time variables we can infer that $t_2 = a + 10$. Using this value the evaluation of the rest of the condition body can be scheduled for time $a + 10$. At that time point the **holdsat** predicate will be evaluated as either true or false. The current implementation of the policy evaluation mechanism is only capable of scheduling evaluation when the inferred time variable is assigned a single value. However, we believe that time relationships that specify a range of possible values for an undefined variable (such as $t_1 > t_2$) can be useful for the efficient scheduling of the policy evaluation procedure. As we expect a policy based adaptive system to have an intrinsically higher overhead with respect to traditional "hard-coded" adaptive solutions, we see the efficiency of the rule evaluation as being an important area for us to explore in our future work.

5. Prototype

We have developed a prototype of the platform presented in section 3 in order to evaluate our ideas. The prototype consists of a full implementation of the registry and adaptation control modules and an initial version of the interpreter/evaluator of the event calculus policy language. The policy evaluator accepts the full policy language as described in section 4, and implements the policy evaluation algorithm presented in section 4.3. One aspect of the evaluator that we have not yet implemented, is provision for mechanisms which detect conflicts between the policy definitions, this is an key aspect of our approach an important area for future work.

In addition, we plan to provide a set of tools that enable the system to more easily integrate the user in the conflict resolution process. These tools will promote awareness of potential adaptive conflicts and allow the user to assist in their resolution by adjusting or augmenting the existing policies and evaluation strategies.

6. Related Work

Most mobile systems supporting adaptation are concerned with network related adaptation [9, 11, 17]. As presented in section 2.1, efficient adaptation requires the decoupling of adaptation mechanism and policies so as to allow the dynamic modification of the latter to detect and avoid conflicts, and achieve stable and coordinated adaptation.

As a representative of these approaches we can consider the Odyssey system [17]. The Odyssey system is a platform supporting application-aware adaptation. Odyssey consists of a viceroy running on each mobile host, responsible for controlling network resources on that host, and a warden for each type of application data exchanged over the network. The viceroys in Odyssey are the system components responsible for monitoring resources and triggering application adaptation when necessary. However, viceroys are controlled by application specific requirements and do not allow any intervention from the user in modifying their behaviour. Furthermore, adaptation policies in Odyssey are hard coded within both the viceroys and wardens. The first for deciding when to trigger a particular application and the latter for deciding how to modify a particular data stream.

An approach where adaptation mechanisms are externalised through XML interfaces is used in the Puppeteer system [4]. However, Puppeteer proposes this approach as a mechanism for incorporating non-adaptation aware applications into an adaptive system. Thus no general adaptation support is provided and there is no consideration for adaptation policies specified/ modified by the user.

The Smiley system [10] provides an example of an adaptive web browsing architecture where user awareness is a prime requirement. In Smiley, the GUI allows the user to identify the quality of a hyperlink before deciding to follow it.

In relation to policy management and policy languages, our event calculus policy language follows a common approach to policy specification with the *PDL* language [15]. However, as *PDL* does not define any entities representing time it does not allow the explicit specification of time relationships. It is possible to express time through the specification of time events, however we believe that the explicit specification of time variables and relationships between time variables is much closer to the user's notion of time and is thus a key consideration. Moreover, the specification of the fluent in the event calculus policy language allows the user to map policy rules to real life conditions that have a 'duration', such as "I am at home", "network is down", etc.

The Ponder policy language [2] defines a full-scale policy management system. It supports the specification of authorisation, delegation, information filtering, refrain and obligation policies over system specific policy domains. The event calculus policy language described here does not provide a complete policy management system, but it does allow the specification of policies similar to the 'obligation policies' described in Ponder. In our approach, the expression of time relationships in the event calculus policy language provides more flexibility. Time relationships as defined in Ponder are quite limited; used mainly for the specification of constraints over the time a policy should be considered active.

7. Conclusions

In this paper we have presented the requirements for coordinated adaptation of multiple applications, which characterise modern mobile applications. We describe a platform which aims to facilitate the development of such applications through the use of policy driven adaptation. In our approach, policies are described using an event calculus based policy language and exported by applications to our platform. We believe that by thus externalising adaptation policies to an entity with a system-wide purview, coordinated adaptive behaviour is possible. Significantly, our platform and simply policy formalism allows the active involvement of the user in defining the system's behaviour. We have presented formal syntax definitions for the event calculus policy language used by our platform and in describing our current implementation work, have demonstrated how policies are constructed and evaluated efficiently. Moreover, while this language has been chosen as an appropriate solution for our requirements, we believe that it can prove a useful basis for other policy based systems with specific timed dependent behaviour.

References

- [1] K. Cheverst, K. Mitchell, and N. Davies. Design of an object model for a context sensitive tourist GUIDE. *Computers and Graphics*, 23(6):883–891, Dec. 1999.
- [2] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *Proceedings of Policy Workshop*, Bristol, UK, January 2001.
- [3] N. Davies, S. Wade, A. Friday, and G. Blair. L²imbo: a tuple space based platform for adaptive mobile applications. *ACM Mobile Networks and Applications (MONET): Special Issue on Protocols and Software Paradigms of Mobile Networks*, 3(2):143–156, 1998.
- [4] E. de Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, California, March 2001.
- [5] C. Efstratiou, K. Cheverst, N. Davies, and A. Friday. Architectural requirements for the effective support of adaptive mobile applications. Work in progress paper presented in *Middleware2000*, (USA:New York), April 2000.
- [6] C. Efstratiou, K. Cheverst, N. Davies, and A. Friday. An architecture for the support of adaptive context-aware applications. In *Proceedings of Mobile Data Management (MDM'01)*, Hong Kong, January 2001.
- [7] C. Ellis. The case for higher-level power management. In *Proceedings of HotOS'99*, 1999.
- [8] J. Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Proc. of the Second IEEE Workshop on Mobile Computing Systems and Applications*, 1999.

- [9] A. Friday, N. Davies, G. Blair, and K. Cheverst. Developing adaptive applications: The MOST experience. *Journal of Integrated Computer-Aided Engineering*, 6(2):143–157, 1996.
- [10] Z. Jiang and L. Kleinrock. An adaptive pre-fetching scheme. *IEEE Journal on Selected Areas in Communication*, 16:358–368, 1996.
- [11] A. Joseph, J. Tauber, and F. Kaashoek. Mobile computing with the Rover toolkit. *IEEE Transactions on Computers: Special issue on Mobile Computing*, 43(3), 1997.
- [12] R. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1(1):6–17, 1994.
- [13] R. Kowalsky. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.
- [14] R. Kowalsky. Database updates in event calculus. *Journal of Logic Programming*, 12:121–146, 1992.
- [15] J. Lobo, R. Bhatia, and S. Naqvi. A policy description language. In *Proceedings of AAAI*, Orlando, FL, July 1999.
- [16] Microsoft Corporation. Universal Plug and Play device architecture, version 0.91, March 2000. <http://www.upnp.org>.
- [17] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Sixteen ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, Oct. 1997.
- [18] J. Pascoe. The Stick-e note architecture: Extending the interface beyond the user. In *Proceedings of the 1997 International Conference on Intelligent User Interfaces*, Short Papers, pages 261–264, 1997.
- [19] Real Networks Inc. The Real Player. <http://www.real.com>.
- [20] M. Shanahan. The event calculus explained. In M. J. Wooldridge and M. Veloso, editors, *Artificial Intelligence Today, Vol. 1600 of LNCS*, pages 409–430. Springer, 1999.