# Policy-Driven Adaptation of Protocol Stacks*

Liliana Rosa, Antónia Lopes, and Luís Rodrigues
University of Lisbon
lrosa@lasige.di.fc.ul.pt {mal,ler}@di.fc.ul.pt

## Abstract

*Today's mobile applications need to execute in a wide range of heterogeneous devices, that operate in different conditions. In this context, dynamic adaptation of the underlying communication support is fundamental to achieve adequate performance. We address the problem of supporting dynamic adaptation of communication protocol stacks through a policy-oriented approach, which promotes the separation of adaptation from protocol logic. In this paper, we provide an approach overview, and focus on the policy language and modeling primitives that allow to capture the adaptation requirements identified from the experience with* Appia *framework.*

## 1 Introduction

Mobile applications are required to operate in highly dynamic settings, where the resources available to the application, such as battery power, processing capacity, network bandwidth, among others, change dramatically in runtime. To offer good performance in such an environment there are several requirements: dynamic resource management, negotiation of communication protocols and the ability to change applications' behavior during runtime in response to environment changes. This cannot be achieved with static reconfiguration approaches or exclusively via offline performance tuning of the application and communication code. Dynamic adaptation succeeds in giving answer to such demands offering the possibility of applications and communications protocols react to changes in the environment, without interruption of vital services.

In these environments performance can be improved by using different protocol stacks, tailored to the current conditions. Given that, a protocol stack is a composition of protocols offering a specific quality of service, it is fundamental to have the ability to dynamically adapt the protocol stack in reaction to changes in the environment, offering the best quality of service possible.

Often, the adaptation logic is hard-coded in the implementation, entangled with the application or protocol logic. However, it is hard or even impossible to reason about the adaptation logic, reuse it in different contexts and tune it at runtime. Therefore, we adopted a policy-driven approach where the adaptation logic is described through high-level policies, decoupled from the protocol logic.

The adoption of a policy-driven approach has many advantages. It facilitates the development of adaptive software. By expressing adaptation in a high-level language, one is not required to understand every detail of the protocol implementation when defining or reading a given policy. In addition, it is easier to analyze the dependencies between different aspects of the adaptation and detect potential conflicts. Moreover, by separating adaptation from the protocol logic one opens the door to reuse of adaptation strategies in different contexts. Finally, the level of decoupling achieved with this approach makes easier to support the change in the adaptation logic during runtime, without requiring the system to be recompiled and redeployed.

We developed a framework for policy-driven adaptation of communication protocols. In this paper, we focus on the policy language and on the specific modeling primitives that allow designers to have fine-grain control on how the protocol stacks should be reconfigured.

## 2 Approach

Our approach assumes that the system is composed by a set of nodes. In each node, communication is supported by a protocol composition. Each protocol composition includes one or more communication channels. Each channel offers a quality of service that is implemented by a (potentially different) stack of protocols in each node. Our goal is to support dynamic adaptation of the protocol stacks.

In our approach, as illustrated in Figure 1, a centralized *adaptation manager* controls the adaptation, fulfilling a given adaptation policy. All the context information required by the adaptation manager to enforce the adaptation policy is provided by a *context monitor*. This monitor collects and processes context information from each node that participates in the distributed application through a set of

*This work was published in the International Conference in Autonomic Systems, 2006, IEEE

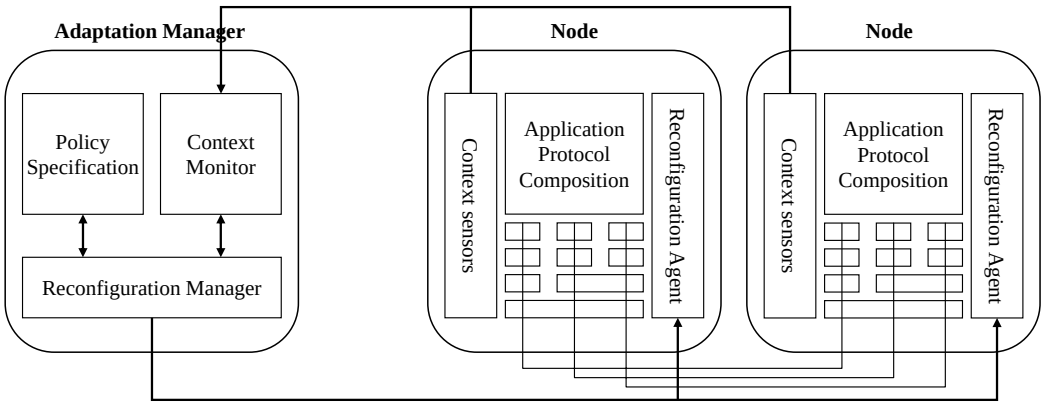


**Figure 1. System architecture**

context sensors executing at each node; sensors acquire the relevant local information and disseminate it to the monitor. The context monitor provides two complementary interfaces to the adaptation manager: an event-based interface, that allows the context monitor to notify the adaptation manager whenever a relevant context change occurs. And a query-based interface, that allows the adaptation manager to read context information on-demand.

Whenever a change in the protocol composition of one or more nodes is required, the adaptation manager controls the reconfiguration through coordination with *reconfiguration agents* executing at each node.

Naturally, our approach requires the runtime system to support dynamic reconfiguration of the protocol composition that executes at each node. For that purpose our work capitalizes on the reconfiguration capabilities of the *Appia* [8] protocol specification, composition, and execution framework that is introduced in the next paragraphs.

**Appia Framework.** The *Appia* framework supports implementation and execution of modular protocol compositions. Each module is a *protocol*, responsible for providing a particular service. A *session* is a protocol instance which maintains the protocol state.

A stack of protocols is called a quality of service (QoS) as it defines a set of properties to be enforced on the message flow. Each *channel* used by an application is associated with an instantiation of a QoS, i.e., a stack of sessions of the corresponding protocols. Each node in an *Appia* composition can offer more than one quality of service, therefore, a node can have more than one channel running. Also sessions can be shared by different channels in the same node.

Sessions interact through the exchange of events. Events are typed and each protocol is responsible for declaring which types of events the corresponding sessions require, accept and produce. This information allows the *Appia* runtime system to optimize the flow of events in the stack.

The *Appia* composition model allows the designer to define rich communication services based on a library of protocols. Currently, the *Appia* library includes more than forty protocols. However, the ability to specify multiple compositions does not, *per se*, support the dynamic reconfiguration of the protocol stacks. To make such reconfiguration possible, on the one hand, control agents are being currently developed for *Appia* [9] that perform tasks such as: i) to ensure that each channel affected by the reconfiguration reaches a quiescent state before reconfiguration takes place; ii) to capture the relevant state that needs to be carried to the new configuration; iii) to deploy the new protocols stacks; iv) to install the state carried from the previous configuration and; v) to coordinate with the adaptation manager and other nodes whenever necessary. On the other hand, we developed a language for expressing adaptation policies at a high-level of abstraction defined over a logical view of protocol stacks. Components are able to carry out the reconfiguration actions dictated by these policies are also currently being developed. The adaptation policy language is the main focus of this paper.

## 3 Adaptation Requirements

When using Appia, protocol stack reconfiguration can be achieved by changing the local state of one or more sessions (typically, protocol parameters), by adding, removing or exchanging protocols of the QoS associated with a channel, or by changing the whole QoS.

In order to identify the relevant modeling primitives for our policy language, we have made an effort to capture the adaptation requirements of previous systems built using the *Appia* framework. *Appia* has been used to support a wide range of applications: multi-user object-oriented environments, distributed real-time games, collaborative mobile application, and database replication [12, 11, 9, 5]. This experience allowed to identify the following needs.

### 3.1 Changing Protocol Parameters

The change of protocol parameters is a common way of achieving runtime reconfiguration of system's behavior.

This kind of change in the behavior of the system is useful for tuning the protocols to specific network's characteristics, for instance, workload, bandwidth, error rate, among others. Many *Appia* protocols include a set of parameters that can be adjusted at runtime. For example, with a high workload, increasing protocol's timeout parameter avoids a network overflow with retransmissions. Other examples of protocol parameters that often need to be changed at runtime are the maximum number of retransmissions, and the frame size (for message fragmentation and reassembly).

The relevant information that triggers this adaptation can be originated by either the network or other environment elements. For instance, the need to reconfigure may be due to changes perceived in a device (e.g., battery power or the use of a wired/wireless connection). Moreover, the changes that require reconfiguration of protocol parameters can be local to a node, a group of nodes, or global. Local changes usually affect only those nodes where the change was perceived. Hence, is important to provide means to define the topological scope of reconfiguration applicability.

In addition, different protocols may have common configuration parameters, being necessary to specify the affected protocols. These target protocols are defined through a protocol scope. Besides, since each node can have several channels, it is also necessary to specify the affected channels. This is done through a channel scope.

### 3.2 Changing a QoS

We identified three patterns for changing a QoS in *Appia* stacks: replace a protocol by another; add/remove a single protocol to a QoS; and replace a QoS by an alternative QoS.

The first pattern is used when one wants to replace an implementation of a given protocol by another. For instance, *Appia* offers at least four different implementations of a total order protocol [1]. Each implementation is optimized for a specific load/network topology.

The second pattern, where a single protocol is added or removed from a QoS, has been used for logging and debugging purposes. *Appia* includes a set of protocols that can log, delay, or reorder events that are exchanged in the stack, and are used for debugging communication protocols. These protocols can be added or removed from a stack without requiring any modification of the remaining ones. Their functionality naturally depends on the position in which they are placed. For instance, an event logging protocol can be added in several positions. The higher is the position in the stack, the lower is the number of events it will record, thus positioning must be defined when adding a protocol. Additional guaranties as encryption or validation can also be achieved through the addition of appropriate protocols.

The third pattern, to replace the QoS associated with a channel by a new QoS, is typically needed when there is a significant change in the operational conditions that requires changes in a large number of protocols. Similarly to changing parameters, the change of a QoS can be limited to specific target nodes.

Experience also demonstrated that triggering of this reconfiguration type is commonly due to change not only on the environment but also on the state of the application. For example, a change in a network trust level will affect all network nodes by adding message encryption service, while a change in a node's trust level will add verification guaranties to that node only. This requires the use of application-dependent sensors that monitor specific aspects of the application state. These sensors supply the required contextual information to the context monitor.

## 4 Adaptation Policies

We now describe the primitives we have developed for the specification of adaptation policies. These primitives allow to specify adaptation policies in terms of a logical view of protocol stacks. They were developed taking into account the requirements identified in the previous section and assuming the system architecture introduced in Section 2.

Policies are defined by sets of adaptation rules. The formulation of adaptation rules is inspired by Event-Condition-Action rules [7], a generic mechanism able to abstract a wide variety of reactive behaviors particularly suited to describe dynamic reconfiguration (e.g., [6, 10]). An adaptation rule has the general syntax:

```
WHEN triggerCondition
[WITH stateCondition]
DO
 {reconfigurationAction
 [WHERE topologicalScope]
 [FOR protocolScope][APPLY channelScope]}+
```

The *triggerCondition* is an event expression and specifies when the rule is triggered. The *stateCondition* determines the conditions in which the given *reconfigurationAction*s are fired. Each *reconfigurationAction* has a *topological scope*(defining the target nodes), a *protocol scope*(determining the target sessions), and a *channel scope*(describing the target channels). The scopes are optional and, by default, an action is considered to target all nodes/protocols/channels.

### 4.1 Context Models

The definition of an adaptation policy is conditioned by the kind of contextual information that is sensed and how it is made available to the rest of the system, namely to the adaptation manager. This is defined through, what we call, a *context model*. A context model defines at a high-level of abstraction what is sensed by the context monitor and how the sensed data is abstracted. This encompasses the definition of the two interfaces offered by the context monitor to the adaptation manager that were mentioned in Section 2:

an event-based interface and a query-based interface. The first interface defines which events are published by the context monitor, the kind of data carried by each event, and how we can gain access to this data. The second interface defines the observables through which is possible to gain access to contextual data and the operations through which it is possible to manipulate this data. We consider that primitive types s.a. *int* and *bool* are built-in as well as the usual operators over these types. Moreover, we also assume the existence of a primitive type *NodeId*, representing network nodes and a type *Set(NodeId)*, denoting a set of nodes.

A context model may define, for instance, that (1) the bandwidth in each node is monitored and made available as an integer through the operation *getBandwidth(nodeId)*; (2) whenever a significant change of this resource is perceived, it is published an instance of the event *BandwidthEvent* with attributes *value:double* and *node:NodeId* carrying the information about the new value of the bandwidth and the identification of the node, respectively; (3) *JoinNodeEvent* is published when a node joins the network.

### 4.2 When

The *When* part is mandatory and consists of a *triggerCondition*, describing which events trigger the rule. In fact, *triggerCondition* is an event expression of the form

```
event: condition {or event:condition}+
```

where the *condition* concerns the data carried by the corresponding event. In this way, it is possible to filter event instances that are not relevant as well as specify that instances of different events may trigger the same rule.

Consider, for instance, the following declaration.

```
WHEN BandwidthEvent: BandwidthEvent.value < MIN
     or JoinNodeEvent
```

It describes that both the occurrence of *BandwidthEvent* or *JoinNodeEvent* may trigger the underlying rule. In the first case, the rule will only be triggered if bandwidth is less than *MIN* (a constant value). In the second case no other evaluations are required.

### 4.3 With

Under *With* label it is stated in which conditions the underlying reconfiguration actions should be fired. These are boolean expressions over the state of the system and its environment made available by the context monitor as defined in the underlying context model. They complement the conditions on the event attributes expressed with the *When* primitive. Once the rule is triggered, *with*-conditions are evaluated to determine if it is necessary to react or not. When these conditions evaluate to true, we say that the rule is activated. This clause can be omitted, meaning that rule triggering always results in firing of its set of actions.

The following example illustrates the use of the *With* declaration, assuming a context model including the definition of a type *Network* with attributes *numberOfNodes* and *numberOfWiredNodes*. This means that this information is maintained by the context monitor and that their state is kept updated through relevant sensors and event filters.

```
WITH Network.numberOfNodes()>10 &&
     Network.numberOfWiredNodes()>3
```

This example describes a composite constraint on the context state. It states that the underlying action would be fired only if, in the current state, the system has at least ten nodes and at least three of them are wired.

### 4.4 Do

*Do* declaration determines the group of elementary actions that have to be performed when the rule is activated. The definition of this group is twofold. On the one hand, we can define a sequence of actions, knowing that the order will be followed by the adaptation manager. The execution of each action may require the coordination with the reconfiguration agents in one or multiple nodes of the system. On the other hand, each of these actions can be composed in the sense that it involves the execution of a set of elementary actions, for which no execution order is guaranteed.

Elementary actions available for adapting the system will be described in detail in the next subsection. Each action has a scope of applicability defined by a topological, protocol and a channel scope. The topological scope concerns the target nodes of the action and it is described under *Where* label, by an expression with type *Set(NodeId)*. The channel scope concerns the channels that can be affected by the action and is defined, under *Apply* label, by a channel expression. The protocol scope concerns the target sessions and is defined, under *For* label, by a session expression. Session expressions are defined in terms of protocol and channel expressions as described in the next paragraphs.

The specification of protocol and channel scopes relies on grouping protocols and channels according to their functionalities. To do so, we assume fixed hierarchies of protocol types and of channel types [2] defining a subtype relationship and also identifying those that are abstract types. These hierarchies are domain dependent, in the sense that applications with different domains may require different hierarchies, but are subject to some constraints.

Protocol types can represent concrete, such as *TCP communication protocol*, or abstract protocols, such as *Transport protocol* (Figure 2). The root of the hierarchy is fixed and defined by the abstract type *Protocol* (every other protocol type is a subtype of *Protocol*) and a protocol type may be a subtype of more than one type. Channel types may also represent concrete or abstract channels. A fragment of a channel hierarchy is depicted in Figure 3. The root of a

channel hierarchy is always the abstract channel type *Channel*.

Protocol and channel expressions are defined in terms of the elements of the underlying type hierarchy, according to the follow syntax:

```
TypeExp := ◇Type | Type |
           TypeExp and TypeExp |
           TypeExp or TypeExp
```

A *TypeExp* denotes a set of types. More concretely, ◇*Type* denotes the singleton set with the type *Type*; *Type* denotes those types in the hierarchy that are subtypes of *Type* (including itself); *TypeExp* **and** *TypeExp* denotes the intersection of the corresponding sets; and *TypeExp* **or** *TypeExp* denotes the union of both sets.

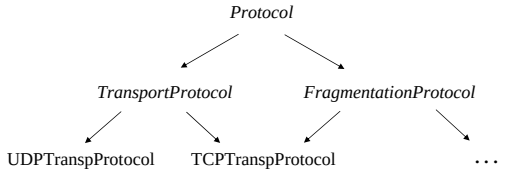


**Figure 2. A fragment of a protocol type hierarchy.**

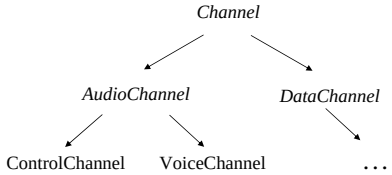


**Figure 3. A fragment of a channel type hierarchy**

Session expressions are defined in terms of protocol and channel expressions as follows.

```
SessionExp :=
  ProtocolTypeExp | ChannelTypeExp |
  SessionExp below ProtocolTypeExp |
  SessionExp above ProtocolTypeExp |
  SessionExp and SessionExp |
  SessionExp or SessionExp
```

A *SessionExp* denotes a set of sessions in use in a particular instant of time. Session expressions allow to select sessions by their protocol type, the channels to which they belong, and their relative position in a channel QoS. More concretely: (1) the first expression denotes the set of sessions that are instances of a protocol type in *ProtocolTypeExp*; (2) the second expression denotes the set of sessions that belong to a channel with a type in *ChannelTypeExp*; (3) the third expression denotes the set of sessions in *SessionExp* that, in some channel, are immediately below a session that is an instance of *ProtocolType* (and similarly for the expression with above); (4) *SessionExp* **and** *SessionExp* denotes the intersection of the corresponding sets and (5) *SessionExp* **or** *SessionExp* denotes their union.

When dealing with shared sessions, performing actions can be problematic: an action that targets a session of a given channel has side effects on the other channels that share that session. The *channel scope* of an action supports the explicit definition of the channels that can be affected by an action.

We present below an example of a rule body. When the rule is activated, the two actions are executed in sequence: *action1* targets the wired nodes and is applied to the sessions of transport protocol that belong to an audio channel; *action2* targets the mobile nodes and is applied to the sessions of transport or fragmentation protocol that belong to a data channel.

```
DO
    action1
    WHERE Network.wiredNodes()
    FOR TransportProtocol and AudioChannel
    action2
    WHERE Network.mobileNodes()
    FOR (TransportProtocol or FragmentationProtocol) and
        DataChannel
```

## 4.5 Reconfiguration Actions

### 4.5.1 Changing protocol parameters

The action *setValue(parameter,newValue)* allows to change the *parameter*'s value and set it to a *newValue* during runtime. The topological scope defines the nodes that are affected by the action. In these nodes, the set of target sessions —the sessions to which the action will be applied— is defined by the protocol scope. Given that the channel scope constrains the channels that can be affected, if a target session does not belong to any channel in the channel scope, the application of the action to that session has no visible effect. Moreover, if a target session is shared between channels $x$ and $y$ and only $x$ is in the channel scope, the session is split in two (one section for each channel) and the change of the parameter's value is only performed on the session of channel $x$. The application of the action to sessions that are instances of protocols that do not have the given protocol parameter, has no visible effect.

The *setValue* action can be used, for instance, to exchange the timeout value in all sessions of *TransportProtocol*, whenever the available bandwidth is below a specific *MINBD* value:

```
WHEN BandwidthEvent: BandwidthEvent.value < MINBD
DO
    setValue(timeoutvalue,NEWVALUE)
    FOR TransportProtocol
```

This rule is triggered by a decrease in available bandwidth, below the *MINBD* value. Since no *With* label is defined, whenever this rule is triggered it becomes immediately activated. The action does not have a topological nor a channel scope and, hence, the action affects the sessions with type *TransportProtocol* in all channels in use and all

nodes in the network. Furthermore, the sharing of sessions does not change.

#### 4.5.2 Removing/exchanging guaranties

The actions *removeProtocol()* and *changeProtocol(newProtocol)* support the removal and exchange of guaranties, respectively. In case of an exchange of protocol, the parameter *newProtocol* identifies the new protocol, which must be a concrete protocol from the underlying protocol type hierarchy.

The set of sessions that will be removed/exchanged is defined by the topological, protocol and channel scopes. The target sessions are all sessions in *protocolScope* that reside in nodes in the *topologicalScope*. As before, if a target session does not belong to any channel in the channel scope, then the application of the action to that session has no visible effect. Furthermore, if a target session is shared between channels *x* and *y* and only *x* is in the *channelScope*, the session is split in two and the removal/exchange is only performed on the session of channel *x*.

#### 4.5.3 Adding guaranties

Addition of guaranties to stacks is supported by the action *addProtocol(newProtocol,position)* where *newProtocol* is the concrete protocol that provides the guaranty to be added. The relative position of the new protocol in the stack is given in the simplest form: *below* or *above*. The *protocolScope* identifies below/above which sessions must the new service be added. The effect of this rule is the addition of new sessions of *newProtocol* immediately below/above all sessions in *protocolScope* that reside in a node in *topologicalScope*, placing these sessions on the channels that are defined by the *channelScope*.

The new sessions are not shared by different channels, even if these channels share the session that defines the insertion point. Moreover, if the insertion point session is shared, protocols can be added in all channels or only in the channels of specific types defined through *channelScope*.

To illustrate these actions, consider a scenario in which it is desirable to dynamically add and remove a debug protocol from the protocol stack of a channel used for communication. In this scenario debug is necessary whenever the system has an error related to communication but should be removed as soon as this error disappears. We assume that this information is made available by the context monitor through the publication of events *ErrorStartEvent* and *ErrorEndEvent* with the attribute *type* with the value *COMM*.

```
WHEN ErrorStartEvent:ErrorStartEvent.type==COMM
DO addProtocol(DebugProtocol,above)
   WHERE {ErrorStartEvent.NodeId}
   FOR TransportProtocol
   APPLY DataChannel

WHEN ErrorEndEvent:ErrorEndEvent.type==COMMUNICATION
DO removeProtocol()
   WHERE {ErrorEndEvent.NodeId}
   FOR DebugProtocol above TransportProtocol
   APPLY DataChannel
```

These rules will affect only the channels of type *DataChannel* residing in the node where the error was detected/solved, whose identification is carried by the trigger event. Thus, preventing the removal of sessions of *DebugProtocol* from the stack that were not inserted by the first rule. We assume that *DataChannel* is the type of channels used by the application for communication.

#### 4.5.4 Exchanging QoS of channels

The action *changeQoS(newQoS)* allows to exchange the quality of service ensured by channels of a specified type, for a whole new QoS. The new QoS is defined by *newQoS*, a sequence of concrete protocols in the stack, from bottom to top. The set of concrete channels affected by the action is defined by its channel and topological scope: all channels in use with type in *channelScope* residing in nodes in *topologicalScope* (the *protocolScope* if specified, is ignored). The instantiation of the QoS associated with a channel *x* whose QoS is required to change may include a session *s* shared with a channel *y* that should remain unaffected. The effect of the rule is to associate to *x* an instance of *newQoS*, keeping the shared session *s* in the channel *y*.

For instance, consider that we have a mobile device that can access to both wireless and wired connectivity. To make the best of each case, a different quality of service for the communication stack is used. Assuming a context model with an event *NewConnectivityEvent* that is published whenever a device has a new connection available, carrying information on the connection (wired or wireless), we can define the following reconfiguration action.

```
WHEN
  NewConnectivityEvent:!NewConnectivityEvent.wired
WITH
  Network.isLaptop(NewConnectivityEvent.NodeId)
DO changeQoS(WirelessQoS)
   WHERE {NewConnectivityEvent.NodeId}
   APPLY DataChannel
```

This rule affects the channels with type *DataChannel* residing in the node where the new connection is available, being this information carried by the trigger event. *WirelessQoS* is the stack configuration for a wireless connectivity quality of service.

## 5 Related Work

Adaptation is a challenging problem that must be addressed at all levels of a system's design and implementation; it needs to integrate many different solutions including algorithms to capture and infer common patterns in the way the context change; application and interfaces that can change their behavior to adjust to dynamic resource

changes; software architectures that support dynamic reconfiguration, among others. In this section we will mainly address the platforms that are closer to our work, namely: *Coyote* [3], *Chisel* [6] and *Poema* [10].

*Coyote* is a protocol composition and execution framework where fine-grain modules, also called microprotocols, communicate through the exchange of events. Events are processed by event handlers that are defined at compile time. Dynamic reconfiguration may be achieved by activating or deactivating handlers in runtime and, hence, Coyote does not address the problem of specifying dynamic reconfiguration policies using an high-level approach.

*Chisel* is a policy-driven, context-aware framework for dynamic adaptation based on triggering events, conditions evaluation and behavior change. Adaptation targets service objects, whose dynamic behavior change relies on metatypes, a characterization of an object's own model. Metatypes allow to add new non-functional behaviors to object, without stopping its execution. Since adaptation is achieved only by adding new behaviors, policy specification is based on *ON-DO-IF* primitives and triggering events definition based on a *NEW* primitive. Chisel's approach is oriented to adaptation of applications and, hence, the policy specification language does not offer the primitives for specifying the adaptation of protocol stacks.

*Poema* is a policy-based framework that supports runtime application reconfiguration by allowing the specification of different reconfiguration patterns. This framework relies on the *Ponder* [4] generic specification language for policies definition through declarative event-condition-action rules, that can be refined to different application areas. In our work, we have identified concrete conditions, actions, and scopes that are relevant to the area of protocol reconfiguration.

## 6 Conclusion

Adaptive protocol stacks are a key element to build efficient systems in dynamic environments such as mobile networks. However, reconfiguration of protocol stacks often involves fine-grain control over which protocol layers are used, how these layers are combined, and how the parameter of the corresponding layers are set. Since this exercise is often performed by the protocol developer, the result tends to be entangled with the algorithmic logic.

In this paper we have proposed a set of primitives that capture common requirements that emerge when reconfiguring protocol stacks. Based on our experience with the *Appia* framework, we have identified the relevant triggers, conditions, actions, and scopes to support the description of adaptation policies for protocol stacks in a high level language. Using this approach, policies can be reused and combined with other policies designed for different protocols. To illustrate our approach, we have rewritten policies from a previous *Appia* prototype using our policy language.

The rules defined in a policy are assumed to be evaluated sequentially. Therefore, we require that a single flow of control is used to evaluate the conditions and apply the policy. Furthermore, most of actions we have identified require coordination among all (target) participants. The extension of this work to cope with decentralized control where different rules can be applied concurrently at different nodes with loose or no coordination is postponed for future work.

Future work will also focus in scalability and liability concerns, analyzing different distributed management approaches. Moreover, we will validate our approach by implementing a prototype and evaluate performance aspects.

## 7 Acknowledgments

This work was partially funded by FCT project MICAS (POSI/EIA/ 60692/2004) through POSI and FEDER.

## References

[1] The *Appia* web site. http://appia.di.fc.ul.pt.

[2] B. Awerbuch and D. Peleg. Concurrent on-line tracking of mobile users. In *SIGCOMM*. ACM, September 1991.

[3] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4), 1998.

[4] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*. Springer-Verlag, 2001.

[5] Globdata. An efficient software tool for global data access. Globdata home page URL: http://globdata.iti.es.

[6] J. Keeney and V. Cahill. Chisel: A policy-driven, context-aware, dynamic adaptation framework. In *POLICY '03: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, Washington, DC, USA, 2003. IEEE Computer Society.

[7] D. R. McCarthy and U. Dayal. The architecture of an active database management system. In *ACM-SIGMOD International Conference on Management of Data*. ACM Press, 1989.

[8] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of The 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 707–710. IEEE Computer Society, 2001.

[9] J. Mocito, L. Rosa, N. Almeida, H. Miranda, L. Rodrigues, and L. A. Context adaptation of the communication stack. In *ICDCSW '05: Proceedings of the Third International Workshop on Mobile Distributed Computing (MDC) (ICDCSW'05)*. IEEE Computer Society, 2005.

[10] R. Montanari, E. Lupu, and C. Stefanelli. Policy-based dynamic reconfiguration of mobile-code applications. *Computer*, 37(7), 2004.

[11] M. J. Monteiro, J. Pereira, and L. Rodrigues. Integration of flight simulator 2002 with an epidemic multicast protocol. In *International Workshop on Large-Scale Group Communication,(in conjuction with The 22nd Symposium on Reliable Distributed Systems)*, 2003.

[12] S. Teixeira, P. Vicente, A. Pinto, H. Miranda, L. Rodrigues, J. Martins, and A. Rito-Silva. Configuring the communication mw to support multi-user object-oriented environments. In R. Meersman, Z. Tari, and et al., editors, *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE Proceedings*, Lecture Notes in Computer Science. Springer, 2002.