Modelling Adaptive Services for Distributed Systems

Liliana Rosa
IST/INESC-ID
lrosa@gsd.inesc-id.pt

Antónia Lopes
University of Lisbon
mal@di.fc.ul.pt

Luís Rodrigues
IST/INESC-ID
ler@ist.utl.pt

ABSTRACT

There exists a growing class of distributed applications that require adaptive middleware services, i.e., services that are able to monitor changes in the execution environment and in the user requirements, reacting to these changes by adapting their behaviour. This paper proposes modelling primitives that allow to describe the adaptation logic of distributed applications that use reconfigurable service compositions.

Categories and Subject Descriptors

D.2.m [**Software Engineering**]: Miscellaneous

General Terms

Adaptive systems, context-awareness, distributed systems, service composition

1. INTRODUCTION

Today, more and more distributed systems are required to have the ability to operate in highly dynamic environments. Such systems must cope with aspects such as resource variability, changes in the user requirements, faults and intrusions, etc, by dynamically self-adapting their behaviour.

Unfortunately, building adaptive distributed applications, that monitor changes in their operational context and are able react to those changes, is an inherently complex task. Our work aims at reducing this complexity for distributed systems whose adaptiveness can be achieved through the use of adaptive middleware services. To achieve this, we developed an architecture to support the construction of such systems. The architecture comprises general-purpose and application-specific components. General-purpose components are responsible for tasks such as the management and dissemination of contextual information as well as the adaptation management, and can be used in the construction of different adaptive applications. They interact with application-specific components that encode the application's *adaptation logic* and address issues such as which contextual information needs to be monitored and how, when adaptation must take place, and what kind of adaptation must be performed.

In this paper, our focus is on the modelling of application adaptability. We present modelling primitives that allow to describe the *adaptation logic* of adaptive distributed applications. The role of these primitives is twofold. First, they provide means for expressing adaptation explicitly, at a high-level of abstraction, separated from the description of the individual middleware services in use. This facilitates the conception and validation of a design solution, and the comparison between different design alternatives. Second, they support the definition of models that can be used as the basis for automatically obtain part of the implementation of the architecture's application-specific components. This facilitates significantly the construction of adaptive systems as it contributes to the reduction of the development effort and eases system's modification and evolution.

The modelling approach that we developed considers that the core part of an application uses a set of channels, each offering a given quality of service (QoS). Each channel is realized through a composition of middleware services that can be dynamically reconfigured whenever the current composition is not accomplishing what it is intended to do, or better functionality is possible. Support for describing different aspects of the adaptation logic, such as the user's preferences, the properties of the individual services, and the relevant context information, is provided by five complementary elements: *service*, *channel*, *sensor*, *context*, and *application* models. Additionally, the adaptation logic modelling includes a high-level *policy* defining when and how the service compositions in place should be reconfigured.

In previous work, we have developed a service composition framework that supports the implementation and execution of such reconfigurable service compositions [16] and a concretization of the architecture generic components [15]. This infrastructure was used on the construction of prototypes of some case studies and we applied our modelling approach to these examples. In this paper, for illustration purpose, we use a simplified version of a middleware application for database replication.

This paper is organized as follows. Section 2 gives an approach overview, focusing the architecture and the example used for illustration purposes thorough the paper. Section 3 describes the modelling approach that was developed, introducing the different kinds of models that are used. Section 4 contains a discussion of the models described in the previous section. Section 5 addresses the related work and, finally,

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

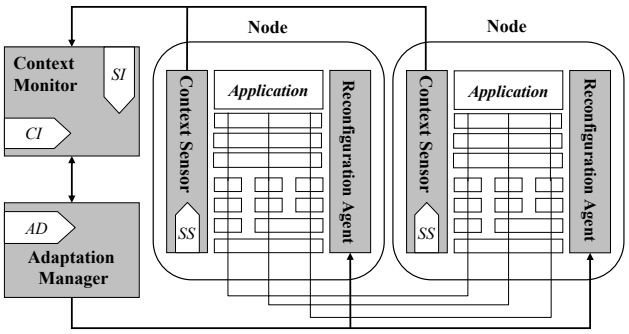

Figure 1: System architecture

Section 6 concludes this paper.

2. APPROACH OVERVIEW

Our approach addresses the construction of adaptive distributed systems whose *adaptation logic* can be separated from the *core application logic*, and achieved through the use of adaptive middleware services [11]. As argued in [9], when an application needs to adapt it is usually not because the core problem domain of the application has changed but rather a non-functional requirement or behaviour of some service within the application, such as the network communication protocol, needs to change.

An application is considered to be structured in terms of a core application layer (or just application, for short) that uses a fixed set of channels. Each channel is realized through a reconfigurable composition of domain-specific and general-purpose middleware services, as depicted in Figure 2. Adaptation may happen in reaction to changes in the user's requirements, which are assumed to be collected by the core application layer, or to changes in the system's operational envelope.

2.1 Architecture

To support the construction of such adaptive systems, we have developed the architecture sketched in Figure 1, where we have represented the general-purpose components by grey boxes.

In this architecture, the system has two types of components involved in the adaptation management — local *Reconfiguration Agents* and a central *Adaptation Manager*. Adaptation is controlled by the *Adaptation Manager*, enforcing the adaptation decisions communicated by an application specific component (*AD*). The adaptation manager is also responsible for guiding the nodes during the adaptation process, either preparing them for reconfiguration, coordinating them, or ordering specific reconfigurations. At each node, the *Reconfiguration Agent* is responsible for performing the necessary reconfigurations of the local service compositions, as ordered by the manager. These reconfigurations can be achieved through the addition, removal, and exchange of services, as well as the fine-tuning of service parameters. Additionally, the architecture has two types of components involved in the gathering, management, and dissemination of contextual information — local *Context Sensors* and a central *Context Monitor*. Context information comprises all relevant information whose evolution can trigger adaptation. This information can have different sources [2], such as user's preferences or devices characteristics, and is captured by the local sensors. All captured information is concentrated in the context monitor. The *Context Monitor* keeps all information and makes it available to an application-specific component (*CI*) that interprets this information and is able to detect the relevant changes. These changes need to be communicated to the adaptation manager. The context monitor is also responsible for informing the generic sensor of each node about the context information that it has to gather locally. It is an application-specific component (*SI*) that defines which context information should be collected by the generic *Context Sensor* in each node and sent to the context monitor. The generic context sensor may obtain this information from the local service compositions and also from application-specific sensors (*SS*) locally deployed. The need for such specific sensors depends on the situation. For example, keeping information on the error rates of different services, can be achieved simply through the use of a generic sensor that collects information from all the target services, through a request/reply approach. On the other hand, to keep information on the available bandwidth at specific intervals or CPU usage, we would need to deploy specific sensors.

2.2 Example

For illustration purpose, we shall use a middleware solution for database replication based in the results of the *GORDA* project [4]. The system is composed by a set of database servers. The database is fully replicated, i.e., a replica of the entire database exists at each server. At each server, an application service is in charge of serving multiple remote clients; it includes an interface with remote clients through which SQL queries are routed. Additionally, the application service also includes the interface with a management console which can be used, for instance, to activate auditing services and other management operations.

The application service uses a JDBC [13] interface to perform queries to the database in the *Data* channel. The queries are processed by a replication service that executes locally the transaction and, at commit time, communicates with the remaining replicas to ensure serializability. For that purpose, the replication uses a group communication service implementing totally ordered atomic broadcast [12]. A description of the database consistency algorithm is outside the scope of this paper (the interested reader may refer to [4]). The service composition of the *Data* channel may include an optional auditing service, that can be present at each node for performance monitoring and management. This service role is to keep track of information regarding the queries processed at the local server.

The application uses another channel— *Control* channel, for administrative purposes, for instance, to temporarily shutdown a replica for maintenance. The channel composition includes a management service that, among other tasks, executes the reintegration procedure when a replica recovers from a crash. The service composition is illustrated in Figure 2.

Two different total order services are available to be used in the service composition, namely a sequencer based total order service and a rotating-token based total order service. The first service offers better performance in systems with unbalanced load, when most queries are performed by a single node. The second service performs best in highly loaded, well balanced systems given that it reduces the contention

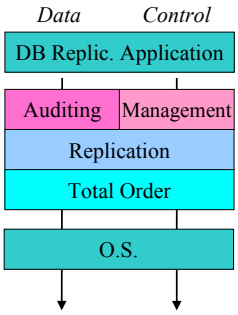

Figure 2: A possible configuration of the database replication application

in low-level resources [12].

The adaptable behaviour of the application results from the reconfiguration of the service composition associated to the channels used by the application. More concretely, its adaptation logic includes rules defining that (i) the *Auditing* service is added or removed from the service composition according to the user preferences and (ii) the *sequencer* based total order service may be replaced by the *rotating-token* based total order service (and vice-versa) depending on the observed load pattern. The next section is concerned with the modelling of this kind of adaptive systems.

3. MODELLING ADAPTATION LOGIC

In our approach, the adaptable behaviour of a system results from the dynamic reconfiguration of the service composition associated to the channels used by the core layer. To support the definition of the runtime adaptation of these service compositions at a high-level of abstraction, we have developed primitives for the specification of high-level adaptation policies. These policies allow to specify when and how the service composition associated with each channel has to be reconfigured in terms of a logical view of channels, services, and service compositions. The choice of these policies was driven by the adaptation requirements of previous systems built using the protocol composition framework [16], as described in [14].

As shown in Figure 3, the specification of an adaptation policy uses elements described in other models: the *service model*, that provides a logical description of the services that are available and can be used in service compositions; the *channel model*, that provides a logical description of the channels whose service compositions can be adapted; the *context model*, that describes the context information required to define the situations in which adaptation is needed. Additionally, modelling the adaptation logic of an application also involves the definition of an *application model* and a *sensor model*, as explained in the next sections.

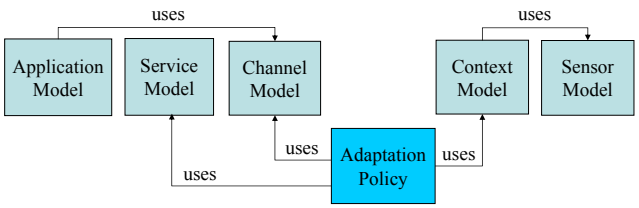

Figure 3: Models for describing adaptation

3.1 Service Model

A *service model* describes the services that are available for composition in terms of a type hierarchy, reflecting the functionality provided by those services. As usual, this notion of sub-typing subsumes the *is a* relationship. Moreover, all the super-type characteristics also apply to the elements of the subtype. Service types can be concrete, designating a specific service for which an implementation is available, or abstract, representing simply the characteristics of a group of other service types. Naturally, the service type hierarchy can have multiple levels. Figure 4 depicts part of the service type hierarchy for the database replication application. In this model, *TokenTOService* and *SequencerTOService* are concrete services, sub-types of the abstract type *TotalOrderService*.

The service type hierarchy supports the specification of adaptation policies in an abstract and flexible way. For instance, it is possible to specify a rule that applies to any service of a given abstract type, without concerns regarding the concrete service used in the composition at a particular moment in time. This is particularly important because the concrete service may change as a result of a reconfiguration. When an application uses multiple service compositions simultaneously, the type architecture also allows to specify reconfiguration rules that apply to all services of a given type, without requiring a specific enumeration of these services.

In addition to the type hierarchy, the service model also describes the configuration parameters of each service type and the context information that it provides. It is considered that a service can provide context information in two manners. One consists in maintaining context information in its local state that can be queried, for instance, the average message load at each node. Another consists in having the service raising an alarm event when some exception condition occurs, for instance, when a replica failure is detected. The full description of services in the service model has the following structure:

```
{abstract} service serviceType is
  subtypeOf [serviceType]*
  parameters [parameterSignature]*
  queries [querySignature]*
  traps [trapName [attributeName; type]*]*
```

For instance, the *Total Order* is an abstract service that could be described as follows:

```
abstract service TotalOrderService is
  subtypeOf Service
  parameters failureDetectionTimeout:long,
      retransmissionTimeout:long
  traps noConnection
service ReplicationService is
  subtypeOf Service
  queries serviceLoad:int
```

This service model fragment describes that the *TotalOrder* abstract service has two configurable parameters – *failureDetectionTimeout* and *retransmissionTimeout*, and throws the *noConnection* trap when connectivity is lost. From this incurs that both *SequencerTO* and *TokenTO* services will also have these parameters and trap. In our example, the service model also describes that the *Replication* service offers a possible query to the sensed information *serviceLoad*, given in terms of the number of *SQL* queries processed in a fixed time interval.

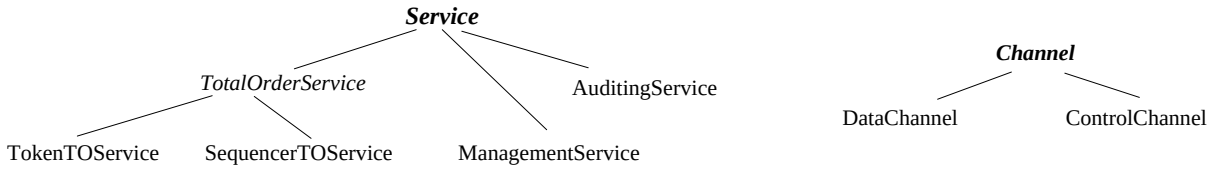

Figure 4: Service and channel hierarchies for the database replication application

3.2 Channel Model

As discussed in Section 2, in our approach, the connection between the application and service layers relies uniquely on the notion of channel. At runtime, a channel is associated with a stack of service instances that process the information sent by the application, and produce and deliver information to the application. Typically, as illustrated in Figure 2, at the bottom of the stack, there is an interface to the operating system level services. For instance, a stack of services may send and receive messages using a socket interface, or save data in the persistent store using the file system interface.

As illustrated by our example, an application may use multiple channels simultaneously, each one for a different purpose, i.e., a different QoS. In order to support the specification of the reconfiguration actions scope, in a flexible manner, channels are described in a *channel model* also in terms of a hierarchy of types defining a subtype relationship. To some extend, the channel types in this hierarchy reflect the types of QoS that are expected from the channel instances.

Figure 4 depicts the hierarchy defined by the channel model developed for the example. The possibility of defining QoS at different levels of abstraction, allows to describe adaptation policies that are reusable across different systems. Naturally, the importance of this model becomes more clear with examples that have richer hierarchies.

3.3 Application Model

The *application model* is quite simple and mainly describes the concrete channels that are used by the application and their type. If the application captures user-defined preferences and these preferences need to be passed to the service layer, the application model also defines how this context information is provided and where. Changes in the relevant user-defined preferences are modelled as context traps, raised by the application, and sent through the channels where they are relevant. The description of an application in the application model has the following structure:

```
use channel channelName:channelType
  traps [trapname [attributeName: type]*]*
```

Our database replication application, which uses two different channels, is described by the application model presented below. The channel *Data* is of type *DataChannel* and the channel *Control* of type *ControlChannel*. The activation/deactivation of the *auditing* service is triggered by the user's preferences that are passed to the service layer by the traps *auditingOn* and *auditingOff* in the channel *Data*. This is described as follows:

```
use channel Data:DataChannel
  traps
    auditingOn
    auditingOff
use channel Control:ControlChannel
```

3.4 Sensor and Context Model

Another important element of the adaptation logic is the context definition, i.e., the surrounding environment characteristics, determinant to the system behaviour. These characteristics need to be sensed and, to be used as context information, the captured data has to be abstracted. The role of the sensor and context models is, precisely, to define these two aspects.

In our approach, the description of context information relies on two types of mechanisms: *observables* and *events*. Observables model the context information part that is kept in the context monitor state, while events are indications of asynchronous changes in the context. Events can carry extra information, as the identity of the node that raised the event.

Sensor Model. This model consists of a set of observables and events, modelling context information that needs to be provided to the context monitor by appropriated sensors. This context information is often obtained from the services in compositions, through queries and traps. Observables and events are described as follows:

```
observable returnType accessName([parameter]*)
  [periodically: number]*
event eventName
  [attributeName: returnType]*
```

Observables have a return value and may also have one or more parameters. The observable definition may indicate that periodical capture is required, with a certain sampling time (for instance, through the query of a sensor that has that information). Events may have attributes, carrying different types of information. In the database replication example, the sensor model declares events representing the change in the user's preferences regarding the auditing service and defines an observable representing the service load. Recall that this information is declared to be sensed by the *Replication* service, in a periodic manner. Hence, if we ensure that this service is always present in the service composition, we can be sure that the solution does not require any specific sensor for sensing this information.

```
observable int serviceLoad(nodeId)
  periodically: TIME
```

Context Model. This model consists of a set of observables and events modelling context information that is provided by the context monitor to the adaptation manager and that can be used in the definition of the adaptation policy. Typically, this kind of information is obtained by interpreting, combining, and/or constraining information obtained from different sources. This information is produced from

the sensed context information through a number of computations. It can be described as follows:

```
observable returnType accessName([parameter
    ]*)
  [periodically: number]*
  expressionOfReturnType
event eventName
  [attributeName: type]*
  [when [condition]
   with [attributeName=expressionOfType]*]*
```

The definition of an observable includes an expression describing how its value is calculated from other observables and/or events, namely, those defined in the sensor model. An event definition includes at least one pair of *when* and *with* clauses. The *when* clause allows to express what is the condition, expressed in terms of other events or changes in other observables, that once evaluated to true, triggers the event publishing. Through the *with* clause it is possible to express the values of each event's attribute. These values can, for instance, be inherited from other events or calculated using observables. Below, we present a fragment of the context model of our running example declaring two observables and three events.

```
observable int AverageLoad()
  periodically: TIME
  (SUM i:Nodes() serviceLoad(i))/
    Nodes().length()

observable int NumbOverloadedNodes()
  periodically: 2xTIME
  COUNT i:Nodes() | serviceLoad(i)>
    (AverageLoad()+LIMIT)

event SingleNodeOverloaded
  id:nodeId
  when NumbOverloadedNodes()==1
  with id=i:Nodes() | serviceLoad(i)>
    (AverageLoad()+LIMIT)

event NoSingleNodeOverloaded
  when NumbOverloadedNodes()!=1

event AuditingPreferenceChange
  id:nodeId, withAud:bool
  when auditingOn with id=auditingOn.id
      withAud=true
  when auditingOff with id=auditingOff.id
      withAud=false
```

The logical architecture of the service compositions associated with channels is an important information that, in most of the applications, needs to be taken into account while specifying adaptation. For this reason, this information is considered to be part of any system context information and, hence, is provided by the infrastructure. In this way, in addition to application-specific sensed events and observables, every context model includes some built-in observables capturing the logical structure of the current service compositions. For instance, it includes *observable bool hasService(ServiceType,ChannelType,nodeId)*, that allows to know if a certain service type is present in the service composition of a channel type in a given node. Every service composition is associated to a specific node. The list of nodes with service compositions is also a built-in observable, accessed as *Nodes()*.

3.5 Adaptation Policy

An adaptation policy defines *when* adaptation should be performed, *how* the application should be adapted, and *what* to adapt, i.e., which are the adaptation targets [5]. This description can be achieved by a set of rules, described using a policy specification language [14]. Each rule follows an event-condition-action (ECA) [10] style, specifying the events that trigger the rule, the conditions that must apply to activate the rule, and the reconfiguration actions to be applied. All the elements needed to specify these rules are defined by the previous models.

More concretely, in each rule, the conditions that trigger the adaptation are expressed in terms of context information produced by the context monitor. The elements defined in the context model are used to refer context changes. The adaptation to be carried out, when the rule is triggered, is expressed in terms of a number of actions that can be performed on the current service compositions. The reconfiguration actions available are: tuning parameters that change the behaviour of a service and add, remove, or replace an ordered set of services by another one. Finally, the adaptation target expresses which nodes, services, and channels of the distributed application should be affected by the reconfiguration actions, in what is called *action scope*. This scope is specified in terms of the target elements defined in service and channel models.

An adaptation policy is a set of rules that define all circumstances that require adaptation and the corrective action to be taken. Each rule has the following general syntax:

```
When triggerCondition
[With stateCondition]
Do {reconfigurationAction
   [Where nodeScope]
   [For serviceScope]
   [Apply compositionScope]}+
```

The *triggerCondition* is a context model defined event and specifies when the rule is triggered. The *stateCondition* is a function of one or more context model defined observables that specify the conditions that need to be satisfied so that the rule can be applied. Each *reconfigurationAction* has a scope composed of a *node scope* (defining the target nodes), a *service scope* (determining the target services using the types defined in the service model), and a *channel scope* (describing the target channels using the hierarchy on the channel model). The scopes are optional and, by default, an action is considered to target all nodes/services/compositions. In the database replication example, the adaptation policy includes the following rules:

```
When SingleNodeOverloaded
With !hasService(SequencerTOService,
    DataChannel, SingleNodeOverloaded.id)
Do changeServices([SequencerTOService])
  For TotalOrderService

When SingleNodeOverloaded
With !isSequencer(SingleNodeOverloaded.id)
Do setParameter(setSequencer,
    SingleNodeOverloaded.id)
  For SequencerTOService
```

```
When AuditingPreferenceChange &&
     AuditingPreferenceChange.withAud
With !hasService(AuditingService,
     DataChannel,
     AuditingPreferenceChange.id)
Do addServices([AuditingService], above)
   Where AuditingPreferenceChange.id
   For ReplicationService
   Apply DataChannel
```

The first rule states that, when a *SingleNodeOverloaded* event occurs, if the *TotalOrderService* is in place, it must be exchanged by a *SequencerTotalOrder*. Moreover, the new service parameter *setSequencer* has to be set to the node id with highest load, carried by the trigger. The second rule states that, when a trigger *AuditingPreferenceChange* event occurs, if the node has not the *Auditing* service already active, then it has to be added to the *Data* channel, above the service of type *Replication* in place in the composition. For more details on the policy specification language, please refer to [14].

4. DISCUSSION

In this paper, we addressed the modelling of adaptive middleware services for distributed systems. Having adopted an infrastructured-centred view of adaptive systems construction, we focused on the definition of modelling primitives that allow developers to represent the adaptation logic of an adaptive distributed system at a high-level abstraction.

As shown, the adoption of appropriate abstractions of the involved elements, permits the formulation of general rules for adaptation at the structural and behavioural level of service compositions. Furthermore, these rules are amenable to different kinds of analysis useful in the validation of design solutions. For instance, the analysis of dependencies between different aspects of the adaptation may lead to the detection of conflicts that invalidate a solution. Similarly, the analysis of the set of reachable service compositions may lead to the detection of situations in which context information that is supposed to be collected from a service composition is not provided by any of its elements (for instance, because it is possible to reach a state in which the single service that may provide that information is absent).

Adaptation models are abstract representations, for which concrete implementations have to be developed. The fact that the proposed models are tailored to a specific architecture makes them particularly useful during the construction of implementations. They help developers to build implementations that are consistent with what was designed, in two different ways. On the one hand, the channel, service, and application models explicitly state what is expected from each channel (QoS), from each service (configurable parameters and sensed information) and from the application layer (channels that are used and user requirements that need to be delivered to the service layer). On the other hand, the context and sensor models and the adaptation policy can be used as the basis for automatically generating some of the infrastructure-specific code that realizes the described adaptation logic (components *CI* and *AD* of Figure 1).

Techniques for supporting this model-driven process are currently being investigated and, hence, our experience using these models in the development of case studies is limited to the first aspect. We applied our modelling approach to some case studies for which prototype implementations using *RAppia* [16] were developed and we found it significantly helped to understand the problem, conceive and validate solutions and, subsequently, it contributed to the reduction of system's development and implementation time.

5. RELATED WORK

There is a large amount of work devoted to the development of frameworks and middleware infrastructures to facilitate the implementation of adaptive systems [9, 7, 18, 6, 3, 5]. There are architecture-based adaptation frameworks such as *Rainbow* [7], whose main goal is to be general purpose, applicable to a wide variety of systems with different architecture styles. *Rainbow* incorporates mechanisms to monitor and adapt systems to surrounding changes, mainly aims at providing a reusable infrastructure, with specialization mechanisms to fill in any particular needs. Being an approach that aims at a broaden applicability, the set of tailorable parts that need to be customized or even developed from scratch still requires some effort. In contrast, our approach relies on a more tight architecture, with a strong structure, but demanding less effort to address specific application's needs in a way that is consistent with what was designed and previously validated.

In the context of composition frameworks, support for adaptation has already been addressed, namely in *Ensemble* [17] and *Cactus* [3]. However, they focus more on the mechanisms to support protocol composition and reconfiguration, than on the models that allow to capture the adaptive properties of the protocols with high-level abstractions.

In most of the existing approaches, modelling of the adaptation logic is not addressed; it is simply programmed, even with high-level models describing the components' information and interconnection [1]. As far as modelling techniques for adaptive systems are concerned, the approach presented in [8] is the most closed related to ours. In this approach, application adaptability can be addressed at high levels of abstraction through infrastructure-independent models, which can be transformed into code for a specific infrastructure using appropriated tools. Because the approach is component-based, the proposed abstractions for modelling application adaptability are not suitable for expressing the adaptation logic of systems with the structure of those that our approach targets.

6. CONCLUSIONS

This paper addresses the modelling of adaptive middleware services for distributed systems. We propose modelling primitives that aim at reducing the complexity and effort required for developing adaptive distributed systems. These primitives have several advantages: i) they allow to describe the application-specific adaptation logic using high-level constructs; ii) they provide the basis to perform the automatic generation of the application-specific components; and iii) they allow to facilitate the evaluation and validation of design solutions. We have illustrated the expressiveness of our primitives in the context of building an adaptive middleware solution for database replication.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] T. Batista and N. Rodriguez. Dynamic reconfiguration of component-based applications. In *PDSE '00: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 32–40, Washington, DC, USA, 2000. IEEE Computer Society.

[2] G. Chen and D. Kotz. A survey of context-aware mobile computing research. Technical report, Dartmouth College, Hanover, NH, USA, 2000.

[3] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting. Constructing adaptive software in distributed systems. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 635–643, Washington, DC, USA, 2001. IEEE Computer Society.

[4] A. Correia Jr., J. Pereira, L. Rodrigues, N. Carvalho, R. Vilaça, R. Oliveira, and S. Guedes. Gorda: An open architecture for database replication. In *NCA '07: Proceedings of the 6th IEEE International Symposium on Network Computing and Applications*, Los Alamitos, CA, USA, July 2007. IEEE Computer Society.

[5] F. J. da Silva e Silva, F. Kon, J. Yoder, and R. Johnson. A pattern language for adaptive distributed systems. In *SugarLoafPLoP'2005: Proceedings of the 5th Latin American Conference on Pattern Languages of Programming*, pages 19–48, 2005.

[6] P.-C. David and T. Ledoux. An infrastructure for adaptable middleware. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 773–790, London, UK, 2002. Springer-Verlag.

[7] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[8] K. Geihs, M. U. Khan, R. Reichle, A. Solberg, S. Hallsteinsen, and S. Merral. Modeling of component-based adaptive distributed applications. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 718–722, New York, NY, USA, 2006. ACM Press.

[9] 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*, pages 3–14, Washington, DC, USA, 2003. IEEE Computer Society.

[10] D. McCarthy and U. Dayal. The architecture of an active database management system. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 215–224, New York, NY, USA, 1989. ACM Press.

[11] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. A taxonomy of compositional adaptation. Technical Report MSU-CSE-04-17, Department of Computer Science, Michigan State University, East Lansing, Michigan, May 2004.

[12] J. Mocito and L. Rodrigues. Run-time switching between total order algorithms. In *Euro-Par '06: Proceedings of the European Conference in Parallel Computing*, LNCS, pages 582–591, London, UK, Aug. 2006. Springer-Verlag.

[13] G. Reese. *Database programming with JDBC and JAVA*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.

[14] L. Rosa, A. Lopes, and L. Rodrigues. Policy-driven adaptation of protocol stacks. In *ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems*, pages 5–12, Washington, DC, USA, 2006. IEEE Computer Society.

[15] L. Rosa, A. Lopes, and L. Rodrigues. A framework to support multiple reconfiguration strategies. In *Autonomics'07: Proceedings of the International Conference on Autonomic Computing and Communication Systems*, page to appear, Washington, DC, USA, 2007. IEEE Computer Society.

[16] L. Rosa, L. Rodrigues, and A. Lopes. Appia to R-Appia: Refactoring a protocol composition framework for dynamic reconfiguration. DI/FCUL TR 07–4, Department of Informatics, University of Lisbon, March 2007.

[17] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. *Softw. Pract. Exper.*, 28(9):963–979, 1998.

[18] R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, and D. Bakken. Quo's runtime support for quality of service in distributed objects. In *Middleware'98: Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 207–222, London, UK, 1998. Springer.