

Architecture-based Refinement Process to Support Distributed Dynamic Adaptation

An Phung-Khac, Maria-Teresa Segarra, Jean-Marie Gilliot,
and Antoine Beugnard

Department of Computer Science, TELECOM Bretagne
Technopôle Brest-Iroise - CS 83818 - 29238 Brest Cedex 3 - France
{an.phungkhac,mt.segarra,jm.gilliot,antoine.beugnard}
@telecom-bretagne.eu

Abstract

Distributed autonomous applications are generally composed of a set of distributed objects (components) that collaborate to offer some particular functions. These applications execute in environments in which operational context changes occur frequently. Allowing such applications to use dynamic adaptation mechanisms becomes necessary in order to meet requirements. Adaptation actions performed by such mechanisms may affect collaborations among several distributed components. Therefore, planning such distributed adaptations is a complex task for developers. This paper presents Adapt-Medium, an architecture structuring adaptation mechanisms for adaptive distributed components as well as a model-based methodology that automatically generates the components with adaptation plans executed on context changes.

1 Introduction

Many applications are composed of a set of distributed components collaborating in order to offer some particular functions. Increasingly, such applications execute on dynamically changing environments and to perform their task the use of adaptation mechanisms is required. Applications must therefore be able to dynamically change their behav-

ior or even their internal structure [13]. However, adapting the behavior or architecture of a distributed component-based application may require adaptation actions on several distributed collaborating components which can be challenging. In particular, runtime architecture-based adaptation [7] requires:

- *Specifying consistent architecture variants:* Through an adaptation action, an application moves from a consistent architecture variant to another consistent architecture variant. Specifying such consistent variants is thus critical to ensure the correctness of the application after executing the adaptation action.
- *Supporting runtime transitions:* Runtime transitions between architecture variants are also critical in order to preserve states and data through adaptation actions. Such transitions involve simultaneous distributed processes having dependencies between them. These dependencies make the transitions more difficult.

In order to help developers on both tasks, we propose Adapt-Medium, an architecture-based approach that introduces dynamic adaptation mechanisms on distributed component-based applications. From a collaboration abstraction called “medium”, we specify a refinement process that builds consistent architecture variants of the medium and embeds these variants into a platform that can select the proper

one at runtime. Thanks to the refinement process all the architecture variants are consistent. Moreover, because all the variants and their *structures* are embedded in the platform at design time, planning runtime transitions including data transfer can be performed easily.

The remainder of this paper is organized as follows. Section 2 presents the original collaboration abstraction called *medium* proposed in our project by Eric Cariou *et al* [5]. Section 3 introduces the design principles of the Adapt-Medium architecture based on the *medium*. Section 4 illustrates how our refinement process can build consistent architecture variants and how it is possible to automatically generate adaptation plans by an example. Section 5 presents related work. Finally, Section 6 summarizes the paper and discusses future work.

2 Medium

A medium is a collaboration abstraction represented as a software entity. A medium-based application is built by interconnecting functional components with a particular medium that models their collaboration [5]. A medium is a logical component that is composed of a set of distributed sub-components. A medium is reusable.

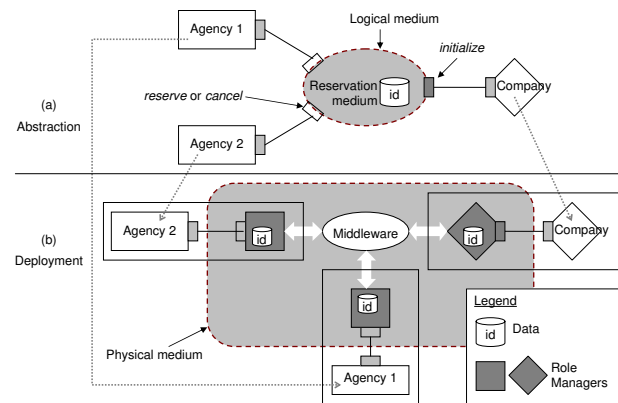


Figure 1: Medium deployment architecture

As an example, consider an airplane seats reservation application of an airline company with travel agencies located worldwide. As shown in Figure 1 (a),

we can specify a *reservation medium* managing seats' identifiers (IDs) and offering *medium services* to initialize information about seats, to reserve seats and to cancel reservations. The reservation application can then be built by interconnecting the reservation medium and functional components representing the airline company and the agencies.

This medium can be reused in similar applications, e.g., in an application for a parking lot that has many places (IDs), cars come in or go out via several entries.

Figure 1 (b) shows the deployment architecture of the reservation medium. At this level, the medium is splitted into physical *role managers*. Each role manager is associated with a functional component and implements the medium services used by the functional component. As shown in the figure, the seats' IDs set may be distributed among role managers. Depending on the data management strategy (e.g., distributed, centralized) or the data type (e.g. list, tree), the medium at the deployment time may be different.

3 Adapt-Medium

3.1 Design Principle

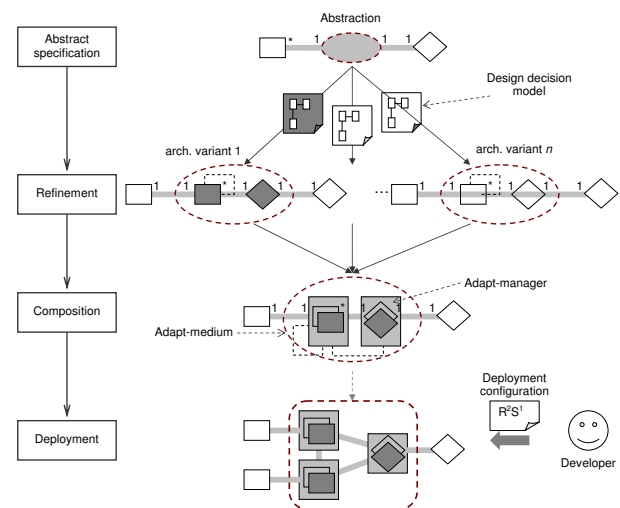


Figure 2: Adapt-Medium design principle

The design principle followed by our approach is to obtain all planned architecture variants from a collab-

oration abstraction, then compose these variants into a new medium, the adapt-medium, and add the necessary machinery to select a proper running variant at runtime.

Figure 2 briefly shows the utilization of our development approach to build an adaptive reservation medium (called reservation adapt-medium). This adapt-medium can be used in reservation applications that dynamically switch the data type and the data management algorithm used for managing seats' IDs when the execution context changes (e.g., the number of agencies increases, evolution of database systems). An adapt-medium is a medium, it is reusable in other similar applications.

- *Refinement.* From the collaboration abstraction, we specify design decision models. Each of these models contains a sequence of design alternatives that lead the refinement of the abstraction to an architecture variant. All the architecture variants conform to the abstraction.
- *Composition.* All the role managers corresponding to a functional component are composed into an *adapt-manager*. A generic implementation of adapt-managers is introduced in this step. Models of the architecture variants, the adapt-medium model, and design decision models are preserved in the adapt-medium.

3.2 Adapt-Medium Architecture

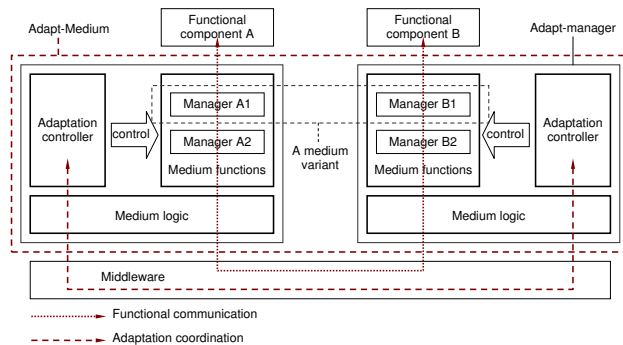


Figure 3: Adapt-Medium architecture

Figure 3 shows the global view of a distributed application using an adapt-medium. This application is deployed on two sites, on each site, a functional component is associated with an adapt-manager. The adapt-medium is then the logical aggregation of two adapt-managers, one per site. As shown in Figure 4, each adapt-manager consists of a *composite manager*, an *adaptation controller* and a *medium logic* component. The composite manager contains all the *manager variants* of the corresponding role manager and an *adaptor*.

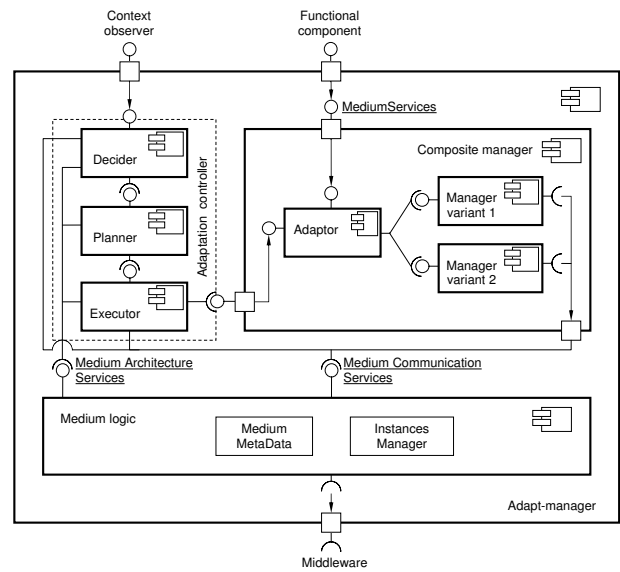


Figure 4: Adapt-Manager architecture

The adaptor calls the medium services of the running variant. The adaptation controller receives context information, makes adaptation decisions, selects a proper running variant, generates adaptation plans and executes them. The medium logic sub-component manages medium's meta-data, i.e. information about structures of all variants and design decision models and adapt-managers' instances information. Functional collaboration between composite managers and adaptation coordination between adaptation controllers are performed through the medium logic layer. In addition, adaptation controllers use medium information managed by medium logic sub-components to schedule adaptations.

3.3 Planning Transitions

Because an adapt-medium contains all the architecture variants at runtime, data of a replaced architecture variant can be transferred to the new one.

Transition plans can be built by analyzing the two design decision models corresponding to the current running variant and the target variant to determine which data from which managers of the current variant should be read, and then, should be write in which managers of the target variant. The result is then a plan of *Read* and *Write* actions. Each of these actions is refined into some coordinated distributed actions by top-down goal decompositions [11].

4 Example: Reservation Adapt-Medium

This section illustrates the refinement process in the adapt-medium design principle presented in the previous section by the example of the reservation adapt-medium.

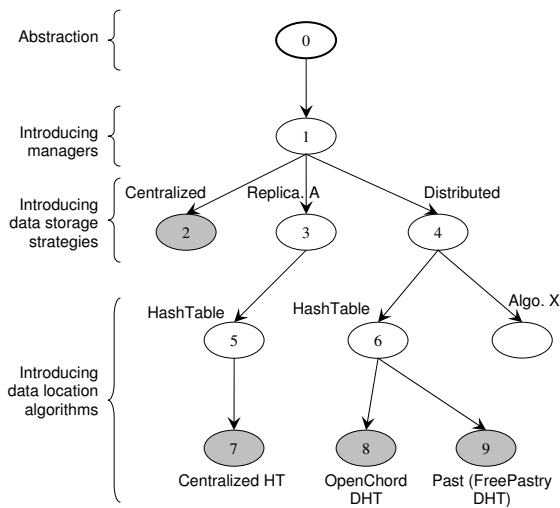


Figure 5: Refinement process

As shown in Figure 5, the refinement process is organized as a tree, the root node corresponds to

the medium abstraction, the end-nodes (gray) correspond to architecture variants, and the inner nodes correspond to refinement stages. The following subsections explain in detail the process.

4.1 Node 0: Abstraction

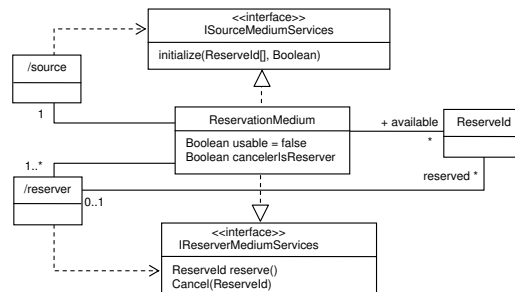


Figure 6: Abstract specification of the reservation medium (Node 0)

Figure 6 shows the class diagram of the reservation medium at the abstract level. The class **ReservationMedium** represents the logical medium that implements the **ISourceMediumServices** interface used by the **Source** class and the **IReserverMediumServices** interface used by the **Reserver** class. The class **ReserveId** represents a seat's ID. The medium manages the (**available**) set of available seats' IDs. Each instance of the **Reserver** class has a (**reserved**) set of seats' IDs reserved by the corresponding agency.

4.2 Node 1: Managers

In order to introduce distributed implementations into the medium, role managers are introduced, one per functional component. Figure 7 shows the class diagram of the medium after role managers introduction. The **SourceManager** and **ReserverManager** classes are introduced. Both classes implement the medium services, but the **available** data are still managed by the **ReservationMedium** class representing the logical medium. The **reserved** data of the

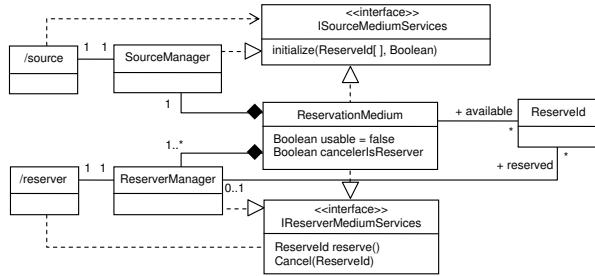


Figure 7: Managers introduction (Node 1)

`Reserver` class in the abstract specification is now managed by the `ReserverManager` class.

4.3 Nodes 2-4: Storage Strategy Alternatives

`ReservationMedium` of the previous stage is still “abstract” and its `Available` data needs to be distributed. The data can be stored by many strategies:

- Centralized: All IDs are stored on a site, e.g., on a *reserver*.
- Replication: n replicas of an ID are stored on n *reservers*. For example, with $n = 2$, we have replication strategy A.
- Distributed: An ID is stored on a *reserver*.

4.4 Nodes 5-9: Data Location Algorithm Alternatives

In order to locate IDs, we can use many algorithms. For example, when an ID is stored on a site, we can insert a pair (ID, Manager’s ID) in a hash table. Then we can use distributed hash table algorithms (e.g., Chord [18], Pastry [15]) or centralized hash table to manage the hash table. Figure 8 shows the medium variant corresponding to Node 6.

4.5 Modeling Design Alternatives

Abstract types used to represent data are also alternatives. Data can be structured in tables, lists, etc.

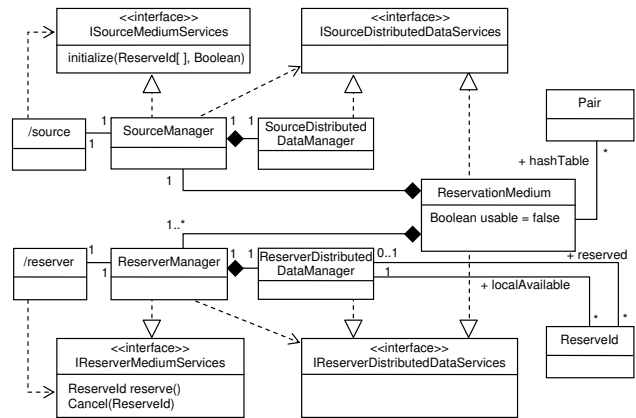


Figure 8: Medium variant corresponding to Node 6

Likewise, corresponding to a data location algorithm, there may be several implementation alternatives.

Design concerns. In this example, we have identified several design concerns of the seats’ IDs distribution: the *abstract type* used to represent distributed data, the *abstract type implementation*, the *data format* used to represent local data, the *data distribution topology* specifying role managers that can participate in the distribution (e.g., client, server, peer), the *distributed protocol* used to implement the data distributed strategy (e.g., Chord [18]), the *distributed protocol implementation algorithm* that specifies the protocol implementation (e.g., OpenChord [1] or MIT Chord implementation [12]). For each design concern, there are several design alternatives (e.g., Chord, Pastry design alternatives for the distributed protocol design concern).

Design decision models. From the identified design concerns, we create design decision models. Each design decision model guide a refinement step that transforms the abstraction into some medium architecture variants. With each design decision model, a medium architecture variant’s model is built.

Figure 9 shows the generic design decision model (a) and a design decision model for the distribution of seats’ IDs (b). For example, the `available` set can be distributed on `ReserverManager` role man-

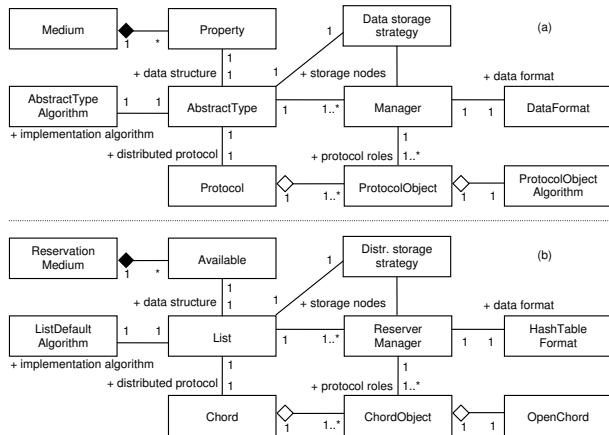


Figure 9: (a) Generic design decision model and (b) a design decision model

agers by using the OpenChord implementation. The `available` distributed data can be accessed via proxies as `List` data. Primitives of the `List` data are implemented by `ListDefaultAlgorithm`. With this design decision model, the abstract medium is transformed into the variant corresponding to Node 8.

The refinement process presented below are automated by model-transformations [14]. We have implemented a transformation program in Kermeta [10] that transforms a UML diagram of the medium abstraction into UML class diagrams of medium architecture variants at the implementation level.

Our refinement process ensures that all the medium architecture variants conform to the medium abstraction. Thereby, they are consistent from the point of view of distributed functional collaborations.

4.6 Composition

The architecture variants from the refinement process are then composed into adapt-managers (Figure 4). This step has been also automated by a transformation program in Java.

Figure 10 shows three component models of three manager variants. These variants 1, 2, 3 correspond to the nodes 8, 2, 7 of the tree in Figure 5.

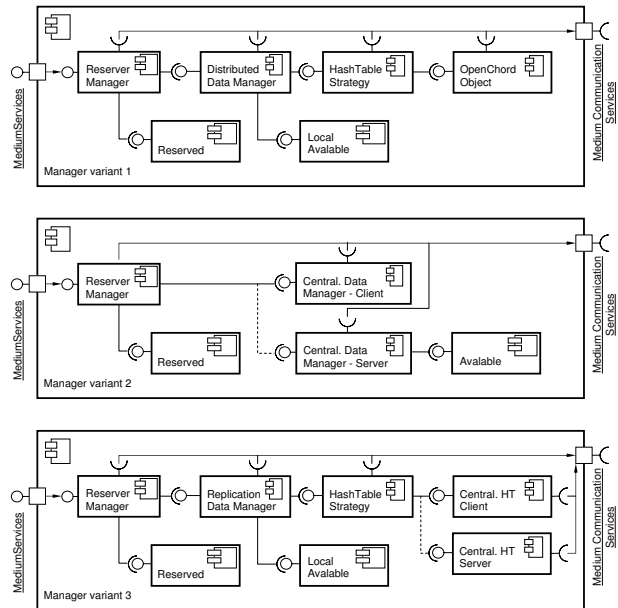


Figure 10: Component models of three reserver manager variants

4.7 Generating Transition Plans

In the refinement process, a sequence of design alternatives forms a design decision model. For example, the sequence of design alternatives corresponding to nodes $\{(0), (1), (4), (6), (8)\}$ in the refinement tree corresponds to manager variant 1 in Figure 10.

In order to automatically generate adaptation plans, we specify transition actions within each step of the refinement process. Therefore, for each step on the refinement process, we specify 1) actions that need to be executed to transfer data from one step to the next one and 2) actions that are needed to restore data of this step from the next one.

For example, with the sequence $\{(0), (1), (4), (6), (8)\}$, the actions can be described as follows:

```

From (0) to (1):
// No data to be transferred
From (1) to (4):
// distribution of Available
for ID in (1).Available

```

```

{
  (4).SourceManager.insert(ID)
  // insert function is implemented
  // in SourceManager
}
Restore (1) from (4):
// Restore the Available data
(1).Available = null
for m in {all managers}
{
  (1).Available.add(m.localAvailable)
}
From (4) to (6):
// No data to be transferred
From (4) to (8):
// build OpenChord HashTable
for m in {all managers}
{
  for ID in m.localAvailable
  {
    m.ChordObject.add(ID,m.name)
  }
}
From (8) to (4):
// No data to be restored

```

Another design decision model corresponds to an architecture variant in which the `available` set is organized in a centralized way. It corresponds to the sequence: $\{(0),(1),(2)\}$ in the refinement process of Figure 5. With this sequence, we can specify the following transition actions:

```

From (0) to (1):
// No data to be transferred
From (1) to (2):
(2).serverNode.available = (1).Available
Restore (2) from (1):
(1).Available = (2).serverNode.available

```

Using these actions, we can automatically generate adaptation plans for switching between both architecture variants. For example, from variant 2 (Centralized) to variant 1 (OpenChord), the path is $\{(2) \text{ to } (1) \text{ to } (4) \text{ to } (6) \text{ to } (8)\}$, then actions need to be performed to transfer data are:

```

variant 2 to variant 1: {
//Restore (2) from (1):
(1).Available = (2).serverNode.available

```

```

// From (1) to (4):
for ID in (1).Available
{
  (4).SourceManager.insert(ID)
}
// From (4) to (8):
for m in {all managers}
{
  for ID in m.localAvailable
  {
    m.ChordObject.add(ID,m.name)
  }
}
}

```

We have implemented a transformation program that generates the reservation adapt-medium. In our program, some functions of the reservation medium are implemented beforehand, then integrated in the adapt-medium by model transformations with the OpenChord implementation and a centralized hash table implementation.

5 Related Work

Many research projects have been investigating techniques to support runtime adaptation of distributed applications. But currently, to the best of our knowledge, there does not exist an approach that supports automatically planning runtime adaptations of applications having distributed functional collaboration.

In the field of robotic, some work supported automatically planning adaptation. For example, in [19], Daniel Sykes *et al* proposed a three-layer model in which adaptation plans are generated from goal models expressed in temporal logic. The plans are executed by selecting alternative components. In the context of distributed collaboration, e.g., two robots collaborate to perform a task, this work does not ensure the correctness of the collaboration between alternatives components of the robots.

A number of approaches supports adaptation mechanisms by replacing or rebinding components [8] or by customizable frameworks to developing adaptable component-based applications [16, 3]. In these approaches, the authors did not focus on the distributed functional logic of applications.

In [2], Gautier Bastide *et al* proposed an approach to create composite components from monolithic ones by restructuring the latter. A composite component consists of sub-components that are deployed on distributed hosts in order to adapt to deployment policies (e.g., when the monolithic component cannot be deployed on a host). Compared with Adapt-Medium, the monolithic component corresponds to the abstraction and a composite component corresponds to an architecture variant. However, because the goal of [2] is to adapt the application deployment, this approach does not support mechanism to switch composite components. Moreover, as concluded in [2], this work does not allow runtime adaptations.

A few approaches support multiple distributed adaptations. In ACEEL [6], an adaptive distributed application has some distributed coordinators that coordinate multiple distributed adaptation in order to maintain the cooperation of distributed components. The coordinators collaborate by using an *adaptation policy* provided by developers. In [9], Kurt Geihs *et al* proposed an approach to develop component-based distributed applications that includes a framework for selecting proper variants based on the current state of the execution context. In this work, the creation of the application variants is also based on some *component plans* describing the components composition defined by developers. By allowing developers define the adaptation policy [6] and the component plans [9], these approaches support a large class of applications, but the capability to maintain distributed collaboration thus depends on developers.

In [4], Nelly Bencomo *et al* proposed an approach to modeling structural variants of component-based applications. The such variants correspond to configurations that are executed by a reflective component framework. Because this work does not particularly focus on distributed collaborations, the presented configurations are not “distributed”.

From the viewpoint of distributed components connection, mediums have a similarity to explicit software connectors [17] used in ArchStudio [13] to supporting runtime evolution. But they differ in many aspects: In contrary to mediums being reusable components, connectors are built by compilers that an-

alyze interfaces specifications of distributed components that need to be connected. Moreover, mediums implement functional collaboration, but connectors implement non-functional interaction of distributed components.

6 Conclusion

In this paper, we presented Adapt-Medium, an architecture to support adaptive distributed components. In the architecture, adaptations are realized by performing dynamic compositions of distributed components. We introduced a model-based process for 1) specifying architecture variants of such distributed components and 2) automatically generating adaptation plans that are performed at runtime to switch running architecture variant. The context includes applications having distributed functional collaborations. In this class of applications, adaptations involving distributed processes may affect the collaborations, planning adaptations is thus a complex task for developers.

In our approach, a distributed application is firstly specified using a collaboration abstraction called medium. Then we presented a refinement process that transforms this abstraction into many architecture variants. These architecture variants are then composed into an adapt-medium that can select a proper running variant and dynamically switch between variants in order to adapt to context changes. We proposed to specify adaptation actions within the refinement process, thus automatically generate plans for performing adaptations.

Our current architecture does not support continuous availability [13]. An adapt-medium enables the application using it to move from a consistent architecture to another consistent architecture at runtime without loss of data, but during the data transfer, the medium services must be stopped. Our ongoing work includes specifying local data as shared objects between manager variants by analyzing common design alternatives of the design decision models. Thereby, we could replace the Read and Write actions in transitions by rebinding components, thus make applications available during transitions.

Our future work includes supporting runtime model transformations in adapt-mediums in order to allow integrating new medium architecture variants at runtime, thus supporting better runtime evolution of adapt-mediums.

References

- [1] Bamberg University, Distributed System Group. Openchord. <http://www.uni-bamberg.de/projects/openchord>, 2007.
- [2] G. Bastide, A. Serial, and M. Oussalah. Adaptation of Monolithic Software Components by Their Transformation into Composite Configurations Based on Refactoring. In *Proceedings of The 9th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, Lecture Notes in Computer Science, pages 368–375. Springer-Verlag, 2006.
- [3] I. Ben-Shaul, O. Holder, and B. Lavva. Dynamic adaptation and deployment of distributed components in hadas. *IEEE Trans. Softw. Eng.*, 27(9):769–787, 2001.
- [4] N. Bencomo, G. Blair, C. Flores, and P. Sawyer. Reflective Component-based Technologies to Support Dynamic Variability. In *2nd Workshop on Variability Modelling of Software-intensive Systems (VaMoS08)*, Germany, 2008.
- [5] E. Cariou, A. Beugnard, and J.-M. Jézéquel. An architecture and a process for implementing distributed collaborations. In *Proceedings of the 6th IEEE International Enterprise Distributed Object (EDOC 2002)*, pages 132–143, Lausanne, Switzerland, September 2002. IEEE Computer Society.
- [6] D. Chefrour. Developing component-based adaptive applications in mobile environments. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1146–1150, New York, NY, USA, 2005. ACM Press.
- [7] B. H. C. Cheng and et al. Software Engineering for Self-Adaptive Systems: A Research Road Map. In *Dagstuhl Seminar*, <http://drops.dagstuhl.de/opus/volltexte/2008/1500/>.
- [8] P.-C. David and T. Ledoux. Towards a framework for self-adaptive component-based applications. In J.-B. Stefani, I. Demeure, and D. Hagimont, editors, *Proceedings of Distributed Applications and Interoperable Systems 2003 DAIS2003*, volume 2893 of *Lecture Notes in Computer Science*, pages 1–14, Paris, 2003. Federated Conferences, Springer-Verlag.
- [9] K. Geihs, M. U. Khan, R. Reichle, A. Solberg, S. Hallsteinsen, and S. Merral. Modeling of component-based adaptive distributed applications. In *Proceedings of the 2006 ACM symposium on Applied Computing (SAC'06)*, pages 718–722. ACM Press, 2006.
- [10] IRISA Triskell Team. Kermeta. <http://www.kermeta.org/>.
- [11] T. W. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.
- [12] Massachusetts Institute of Technology. lsd. <http://www.pdos.lcs.mit.edu/chord/>, 2004.
- [13] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.
- [14] A. Phung-Khac, A. Beugnard, J.-M. Gilliot, and M.-T. Segarra. Model-Driven Development of Component-based Adaptive Distributed Applications. In *Proceeding of the 23rd ACM Symposium on Applied Computing (SAC'2008), track on Dependable and Adaptive Distributed Systems (DADS)*, Fortaleza, Ceará, Brazil, March 2008. ACM Press.
- [15] A. Rowstron and P. Drusche. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware*, novembre 2001.
- [16] M.-T. Segarra and F. André. A framework for dynamic adaptation in wireless environments. In *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, page 336, Washington, DC, USA, 2000. IEEE Computer Society.
- [17] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM Conference*, San Diego, 2001.
- [19] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components: a combined approach to self-management. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 1–8. ACM, 2008.