Implementing a Data Distribution Variant with a Metamodel, Some Models and a Transformation

Eveline Kaboré and Antoine Beugnard

Department of Computer Sciences, TELECOM Bretagne, Technopôle Brest-Iroise
CS 83818 – 29238 Brest Cedex 3, France
{eveline.kabore,antoine.beugnard}@enst-bretagne.fr

Abstract. In this paper, we show how model transformations can be used to implement data distribution features in the software design process of a component. This approach is based on a single metamodel that defines data distribution abstractions and on the design of alternatives that are used to implement each data distribution variant. A model transformation is associated with the metamodel and the component metamodel we consider as the target. We show that this approach facilitates the derivation of different implementation strategies from the model of a component. We illustrate our approach with the example of distributed communication component software that implements one centralized and two peer-to-peer variants and we demonstrate the reusability of the transformation.

1 Introduction and Motivation

Models are widely used in science and have become an essential tool for software designers and programmers. Models have been used in many development methods such as SADT [1], JSD [2], etc. They allow the description of different aspects of a system: structural, functional, behavioural, temporal, etc. Models also allow the description of the system to be developed at different stages with various levels of detail.

The Unified Modeling Language(UML) is the last avatar of a standard modelling notation. The way models are produced and elaborated is mainly beyond the scope of modelling, which relies on good-practice, know-how and more or less formalized methods. One of the latest great advances in software engineering has been the introduction of patterns (especially design patterns) as a semi-formalization of good (or bad) practice.

The formalization and the clarification of the process of elaborating models are the next challenges. Considering the processes of elaborating and renaming models as an activity that can be described with a dedicated language is, in our view, a revolution.

We show in this article how models, metamodels (that can be defined as model types) and model transformations can be used to automate the design and implementation process of a distributed software component.

R. Meier and S. Terzis (Eds.): DAIS 2008, LNCS 5053, pp. 224–237, 2008.
© IFIP International Federation for Information Processing 2008

We are working on a specific component model dedicated to communication [3,4] and the way to derive an implementation thanks to a process based on model transformations [5,6]. Many models and metamodels have been developed and a design process (Fig.1.) has been implemented as a set of model transformations. The first transformations that were defined introduced the general architecture of the implementation. In this paper, we describe the way we have automated the design choice related to data placement and distribution.

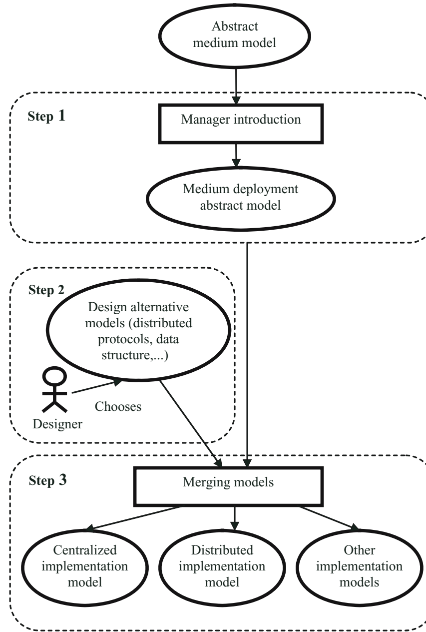

Fig. 1. A view of the full design process of a medium

We argue that if models can describe *the product* to develop and metamodels the *abstractions and constraints* that are used and reused to define models, model transformations can be used to describe *the process* to follow. Hence, we have analysed the different design choices that are available when implementing communication components. Among them, we have isolated data placement strategies. We then looked for the abstractions required to describe data placement and elaborated a dedicated metamodel. The validation of this metamodel was made thanks to the definition of some data placement models such as "centralized placement" "peer-to-peer Pastry placement strategy" or "peer-to-peer Chord placement strategy".

In order to apply these design choices we had to define a model transformation that was compatible with our target: communication components. The transformation was hence defined using the two metamodels: the communication component and the data placement metamodels. This choice guarantees

that the transformation is reusable since applicable to all models that conforms: any communication component model or any data placement model. Finally, this approach also ensures extensibility since new data placement models - if conforming - could be added.

The paper is organized as follows. The next section summarizes the definition and the deployment target of communication components in order to ensure a better understanding of metamodels and transformation. Section 3 presents our approach defining the implementation parts of data distribution problems as a sequence of transformations in a communication component design process. Section 4 presents some related work. We conclude the paper in section 5 with some perspectives of this work.

2 Communication Component: Medium

Definition. A medium is a special component which implements any level communication protocol or system. A medium can implement, for example, a consensus protocol, a multimedia stream broadcast or a voting system. A medium includes classical component properties such as explicit interface specification, reusability or replaceability, but a medium is not a unit of deployment. A communication component is a *logical* architectural entity built to be *distributed*. An application is the result of inter-connecting a set of components and mediums. This is particularly interesting as it can allow the separation of two concerns: local concerns described by components and communication concerns described by mediums.

Example. As an illustration, we reuse the example published in [5] of an airline company with travel agencies located worldwide. A medium can implement the reservation system and offer services to initialize information on seats, to reserve seats and to cancel reservations. A reservation application can then be built by inter-connecting the reservation medium and components representing the company and the agencies as illustrated in Fig.2.

Deployment Target. In the previous section, we saw that, at the abstract level, the medium is represented by a single software component. The goal of the design process is to make the distribution of this abstraction possible. The

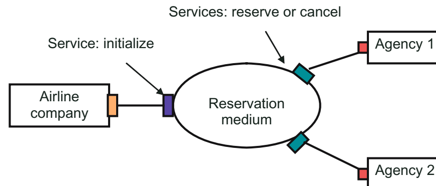

Fig. 2. An example of a communication component: reservation medium

single software component which represents the medium at the abstract level is split into small implementation components called *role managers*. Each *role manager* is locally associated with a local component and the medium becomes a logical unit composed of all the *role managers*. From a local point of view, each *role manager* implements the services used by its associated component. From a global point of view all the *role managers* communicate through middleware and cooperate to realize all the medium services.

Thus, at the deployment level the single software communication component which represents the medium at the abstract level disappears completely and the medium becomes an aggregation of distributed *role managers*. The data manipulated by the medium at the abstract level are distributed between role managers.

The next section presents our approach to implement data distribution features as a sequence of transformations. We note that the definition of technical details related to elements which are used to ensure data distribution and access services is beyond the scope of this paper.

3 Our Approach

3.1 Analysis

Identifying the Source and the Target of the Transformation

Identifying the Source. The introduction of managers is beyond the scope (step 1 in Fig.1.) of this paper. Thus, in the context of this paper, the source of the transformation is a medium deployment abstract model in which:

1. A *manager* is associated with each role
2. Each *manager* implements all the services offered by the medium to its associated role
3. Each role is separate from the other elements constituting the medium
4. The medium is defined by the aggregation of *managers* as illustrated in Fig.3. for the reservation medium.

Identifying the Target. The target of the transformation is a medium implementation model in which:

1. Items (1) - (4) in the source specification are verified
2. The entity representing the medium in the source specification is deleted
3. Each data managed by the medium in the source specification is distributed between *managers*

The description of the target does not provide design alternatives that will be used to implement data distribution features. It just specifies the set of constraints that each final implementation model of the medium should satisfy. Both source and target descriptions are detailed in [7].

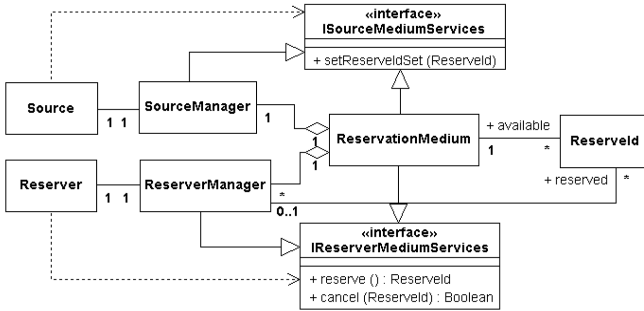

Fig. 3. Structure of the reservation medium after the introduction of managers

Identifying and Separating Design Alternatives. We decompose the problem of implementing data distribution issues into three design alternatives at the moment. Other design alternatives related to data distribution such as synchronization or context adaptation aspects can be added in future work.

First Design Alternative: Data Distribution Topology Choice. This involves in specifying the set of managers that can participate in the distribution of each data and those that can only have access to it.

Second Design Alternative: Distributed Protocol Choice. This specifies the distributed protocol (*Chord [8]*, *Pastry [9]*, etc.) that will be used to implement the distribution strategy of each data.

Third Design Alternative: Distributed Protocol Implementation Algorithm. The last step is the choice of algorithm that will be used to implement each distributed protocol services. As an illustration, in the case of the *Chord* protocol, the designer can choose the algorithm proposed by *MIT* [10] or the algorithm proposed by the *MACEDON* [11] framework to implement the protocol.

3.2 Automation

In this section, we sketch out metamodels and transformations that we use to describe and automate the introduction of design alternatives identified in the previous section in the medium deployment abstract model.

Metamodelling. We define a different metamodel for the source, the target and distributed protocols in order to ensure a better understanding of metamodels. Each metamodel is specified with two elements: a UML class diagram describing the generic structure of the concept and a set of OCL specifications describing the properties of the concept which cannot be expressed in the class diagram. For the sake of brevity, we only show the generic structure of each metamodel in this paper. The full definition of all the metamodels is available in [12].

Medium Deployment Specification Metamodel. Figure 4 shows the generic structure of a medium during deployment. A *manager* (*</RoleName>Manager*) is associated with each role (</RoleName>). Each *manager* implements the interface of the services offered by the medium (*I<RoleName>MediumServices*) to its associated role and the medium (*<MediumName>Medium*) is defined by the aggregation of *managers*.

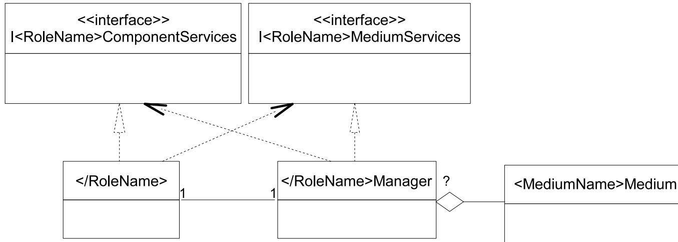

Fig. 4. Generic structure of a medium at deployment level

Medium Implementation Specification Metamodel. Figure 5 shows the generic structure of a medium during implementation. The medium class disappears completely and the medium data are distributed between managers. The gray colour delimits our metamodel of a distributed data. Each distributed data is represented by two elements. The first element (*DataManager*) ensures the data distribution services and the second (*DataObject*) element ensures the data access services.

Distributed Protocol Metamodel. We define a distributed protocol by a set of objects called *ProtocolObject* (Fig.6.). A *ProtocolObject* is an object that can

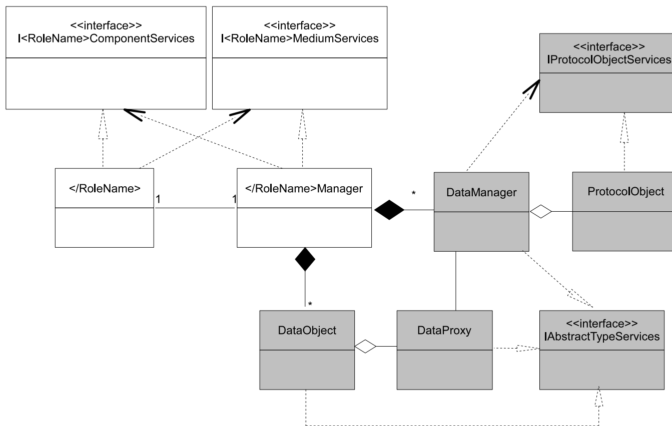

Fig. 5. Generic structure of a medium at implementation level

execute the behaviour of a distributed protocol. Each *ProtocolObject* is implemented by a specific algorithm (*ProtocolObjectAlgorithm*). The main goal of the distributed protocol metamodel involves defining a common interface for all the distributed protocols that will be used in the context of mediums. Such interfaces are proposed in [13,11,9]. The *IProtocolObjectServices* interface exported by the distributed protocol definition metamodel is similar to the interface defined in [13]. This interface defines services for three main distributed application abstractions: DHT (Distributed Hash Tables), DOLR (Decentralized Object Location and Routing) and CAST (group anycast/multicast). The *IProtocolObjectServices* interface offers the following services: *route* (to route a message), *forward* (to forward a message), *deliver* (to deliver a message), *join* (to join the distributed application) and *leave* (to leave the distributed protocol).

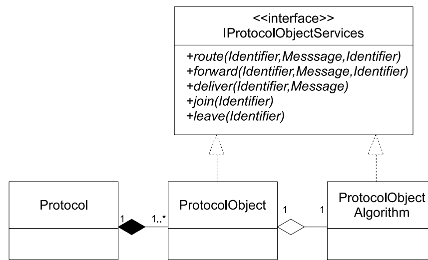

Fig. 6. A view of the distributed protocol specification metamodel

A definition of a distributed protocol model which conforms to the distributed protocol metamodel involves:

1. The description of each *ProtocolObject* and at least one of its implementation algorithms
2. The implementation of each service offered by the *IProtocolObjectServices*.

We illustrate the definition of a *Chord* protocol model in [14].

Model Transformation

Principle. The entry point of the transformation is a well defined medium deployment abstract model (Fig.1. and Fig.3.) in this paper. In the full design process of the medium, we perform a first transformation in order to transform this model into an abstract implementation model in which all the UML associations between the medium class and the medium data classes are replaced by the appropriate abstract types specified by the designer [15]. In the case of the reservation medium for example, the available reservation identifiers (**available** property in Fig.3.) can be represented by a **list**. Since we do not discuss abstract types choice in this paper, this transformation will not be described in this section. After the introduction of abstract types, we perform five successive

transformations in order to introduce data distribution topologies, distributed protocols and distributed protocol algorithms into the abstract implementation model of the medium. These transformations leads to a generic implementation model implementing the actual data distribution variant. The designer completes this generic implementation model by defining each offered service's actual implementation algorithm to produce the final implementation model of the medium. The following example describes a data distribution variant that will be used to illustrate transformations in the remainder of this section.

An Example of a Data Distribution Variant. We will use the reservation medium to illustrate transformations in this section. We suppose that the available reservation identifiers (**available** property) are represented by a list. The goal is to distribute this list between managers associated with agencies (*ReserverManager*) using the *Chord* protocol. The *manager* associated with the airline company (*SourceManager*) can only access the list. The Chord protocol will be implemented by the *MIT* algorithm. These information are defined in a medium decision model in the full implementation process in order to allow the automatic execution of transformations [15].

T_1. Introducing each Data Distribution Topology into the Abstract Implementation Model of the Medium. The input model of this step is the medium abstract implementation model obtained after the introduction of abstract types. We aim at introducing each data distribution topology into this model. We define a transformation based on the medium deployment and implementation metamodels and the distributed data metamodel for this purpose. This transformation leads to an abstract implementation model of the medium in which : a *DataManager* is associated with each *manager* participating in each data distribution and a *DataObject* is associated with each *manager* accessing each distributed data. Here is an informal summary of its main operations.

```
Preconditions:
 1.Verify if each distribution node is defined in the medium model.
Actions:
  For each data managed by the medium class:
   1.Create and associate a generic DataManager object with each manager
     participating to data distribution.
   2.Create and associate a generic DataAccess object with each manager
      using the data
Postcondition:
  Verify if a DataManage and/or a DataObject is associated with each
  manager according to the implementation variant.
```

As an illustration, in the example of the reservation medium, the transformation associates a *ListDataManager* to *ReserverManager* and a *ListObject* with both *SourceManager* and *ReserverManager* (Fig.7.).

T_2. Introducing Distributed Protocol in the Abstract Implementation Model of the Medium In this step we define another transformation based on the same

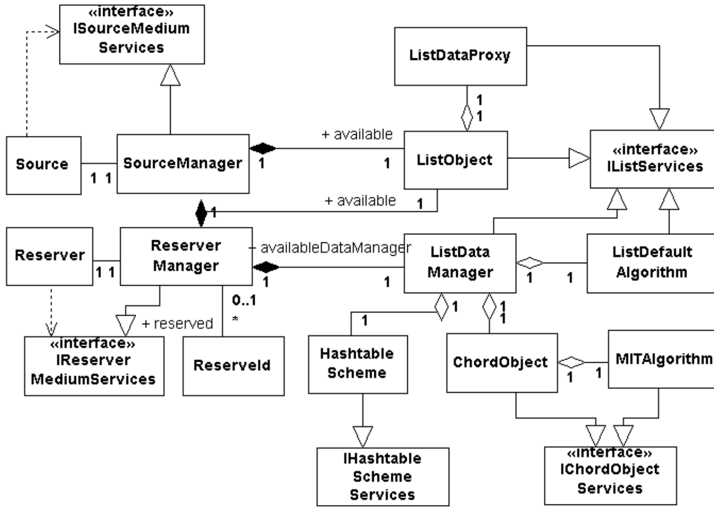

Fig. 7. A view of the reservation medium after T_1, T_2 and T_3

metamodels as T_1 to introduce the distributed protocol that will be used to ensure each data distribution strategy in the abstract implementation model generated by T_1. Its operations can be summarized as follows:

```
Preconditions:
 Verify if the model of each distributed protocol conforms to
 the distributed protocol metamodel
Actions:
1. Create and associate a generic ProtocolObject with each DataManager
   according to the distribution variant.
Postcondition:
 Verify if a generic ProtocolObject is associated with each DataManager
   according to the distribution variant.
```

T_3. Introducing Distributed Protocol Algorithms into the Abstract Implementation Model of the Medium This transformation perform the following operations in order to introduce the implementation algorithm of each distributed protocol object into the model generated by T_2 as illustrated in Fig 7.

```
Preconditions:
  Verify if each distributed ProtocolObject implementation algorithm is
  well defined according to the distributed protocol metamodel.
Actions:
 Create and associate a generic ProtocolObjectAlgorithm with
 each protocol object according to the distribution variant.
Postcondition:
  Verify if a generic ProtocolObjectAlgorithm is associated
  with each ProtocolObject according to the distribution variant.
```

T_4. Generating Abstract Methods Implementation Algorithms in the Abstract Implementation Model of the Medium In this step, we define a transformation to generate a default algorithm in order to implement each abstract method contained in the abstract implementation model produced by T_3. Here is an example of a piece of code generated by this transformation in order to implement the *add* primitive of the **list** for the reservation medium example.

```
class ListObject inherits IListServices
{    ... method get (index : Integer ): Object from IListServices is do
        if (self.dataProxy != void) then
             then result := self.dataProxy.get(id) end
        end end  ... }
class ListDataProxy inherits IListServices
{    ... method get (index : Integer ): Object from IListServices is do
        var dataManager : ListDataManager init getListDataManager()
        if (dataManager != void) then
             then result := dataManager.get(id) end
        end end  ...}
```

T_5. Configuring the Medium The previous step leads to an abstract implementation model of the medium in which all the generic elements needed to provide each data distribution and access service are well defined. In this step, we define a last transformation based on the same metamodels as the previous transformation in order to instantiate and associate the appropriate value with each generic element. Four generic operations called *managerConnection, managerDisconnection, initialization and termination* are defined in [7] in order to associate a specific behaviour with a manager during its connection, disconnection, initialization and termination. The last transformation redefines these operations in order to reach its goal. Here is an informal summary of the main operations performed by the last transformation as an example of generated code for the reservation medium.

```
Precondition:
 Verify if all the abstract methods are implemented in the input model.
Actions:
 1.Redefine the managerConnection, managerDisconnection, initialization
   and termination operations in each Manager class.
 2.For each distributed data:
   2.1.Generate instructions in the managerConnection operations to
       instantiate the appropriate protocol objects and protocol object
       algorithms according to the design choice
   2.2.Set the data manager and the data object values
   2.3.Generate instructions in the managerDisconnection operations to
       disconnect protocol objects
   2.4.Generate instructions in the initialization operations to
       initialize protocol objects
   2.5.Generate instructions in the termination operations to terminate
       protocol objects.
Postcondition:
```

```
 Verify if the output is a good medium implementation specification
 model according to the medium decision model and the medium
 implementation specification metamodel

class ReserverManager inherits IReserverMediumServices
{...operation managerConnection() is do ....
     available := ListObject.new
     available.dataProxy := ListDataProxy.new
     availableProtocolObject := ChordProtocolObject.new
     availableProtocolObject.protocolObjectAlgorithm := MITAlgorithm.new
     availableDataManager := ListDataManager.new
     availableDataManager.protocolObject := availableProtocolObject
     availableDataManager.listDefaultAlgorithm :=
                  ListDefaultAlgorithm.new // other instructions end }
```

Transformation Definition Platform. Transformations are implemented, tested and executed on the Kermeta [16] platform. Each metamodel is implemented by two Kermeta files. The first file implements all the structural aspects of the metamodel. The second file implements all the properties of the metamodel. It is then possible to check if a specific model is conform to the metamodel in which it is defined. Each transformation is implemented by three Kermeta files. The first file implements the preconditions, the second file implements the operations and last file implements the postconditions of the transformation. A full definition of metamodels and transformations in Kermeta is available in [12].

4 Related Works

Most methodologies are informally described. They suggest a process which, in most formalized cases, rely on contracts [17] or mathematical refinements like the B-method [18]. B defines a language and a refinement methodology. It is an algebraic specification language that is supported by tools that help refine specification safely. Each step of the process generates the proof requirement that the developer has to demonstrate, either manually or automatically. Some critical systems have been developed in B (in 1998 the control system of line 14 of the Parisian subway was fully developed and proved in B). Our approach is more empirical and uses the so-called "semi-formal" approach. It may be easier to learn and may tackle different kinds of design problems such as distribution. We do not try to prove design steps, but just to automatize them and give enough confidence in the transformations thanks to pre and post-conditions.

In a recent paper, H. Sneed [19] criticizes the model driven approach. He argues that model-driven tools magnify the mistakes made in the problem definition; create an additional semantic level to be maintained; distort the image of what the program is really like; complicate the maintenance process by creating redundant descriptions which have to be maintained in parallel; are designed for top-down development that creates well-known maintenance problems. These drawbacks are mainly associated with tools. All these criticisms have already

been raised when assembler was replaced by high level programming languages. We agree that tools are not mature. Our experiment shows that transformations may help make explicit the process and simplify the maintenance, if models are well defined enough.

Other experiments [20] tend to prove that model composition (hence a bottom-up approach) is possible. This compositional approach resembles Aspect Oriented Modelling [21]. This approach recommends separating concerns and offers an operation of weaving that composes/weaves each concern with the functional specification. Our approach differs since the "weaving" operation we use is a transformation that is adapted to the kind of concern composed. Instead of using a universal weaving operation we propose a more flexible (but less re-usable) approach in which a balance may be found between the metamodel definition of the concern and its composition operation implemented as a transformation.

Model transformations are widely used in UML models. Most of them cover a small part of the development life cycle. Some transformations are dedicated to code generation. They usually produce the skeleton (structural part) of the source code that has to be completed manually. Another current use is in applying design patterns [22]. Once again, the structural part is rather well implemented[1], but the collaboration part is still research in progress.

5 Conclusion

This paper shows how model transformations can be used to describe the implementation process of data distribution issues in a distributed software component. To do this we have defined metamodels that capture the required concepts of data distribution. We have also realized a sequence of model transformations that weave a variant of the data distribution design choice into the model of the distributed component.

The transformations describe the process of introducing actual design alternative models in the specification of the component. The approach makes explicit the data distribution implementation process. We argue that it is of great interest in the sense that it facilitates traceability (the sequence of transformations), reuse (applicability of transformations to many different models) and the evolution of the full process (adding more variant models).

As an illustration, we have applied our approach to implementing data distribution in the context of mediums. We have described a set of transformations and metamodels that can be used to introduce distributed protocols.

But the concern of data distribution is only one step in a larger design process. We have described an approach based on the definition of a sequence of design concerns. As an example we have selected the choice of an abstract type for the

[1] Pattern purists would say that patterns are not dedicated to be automatically applied. In the absolute, we agree, but why not consider applying patterns in well defined contexts?

collection of data, the choice of distribution strategies (described in this paper) and the choice of data representation format.

Transformations are implemented, tested and executed with the Kermeta platform. The implementation of the full design process relies on 6 metamodels and 5 main transformations. Each metamodel is implemented by two Kermeta files. The first file implements all the structural aspects of the metamodel. The second file implements all the properties of the metamodel. It is then possible to check if a specific model is conformed to the metamodel in which it is defined. Each transformation is implemented by three Kermeta files. The first file implements the preconditions, the second the operations and the last file the transformation postconditions. We also provide a library containing some abstract types (list, set, bag, etc.), distributed protocols (chord, pastry, etc.) and data representation format models (hashtable, array, matrix, etc.).

Transformations can be used to implement any abstract type, distributed protocol and data representation format model which conforms to our metamodels in any well defined medium initial specification model. As an illustration, we have used transformations to automatically derive various centralized and distributed implementation variants of the reservation medium presented in this paper and two other mediums: a voting medium and a message broadcast medium.

The actual generated implementation models of mediums are not fully executable in the sense that they do not provide a full executable code of distributed protocols. They just call protocol APIs to ensure that all distributed feature are well implemented. Thus, in the short term, our main perspective is to build middleware in order to allow the execution of the generated models in conjunction with existing executable distributed protocol frameworks such as MACEDON. We also aim to define other design alternatives metamodels and models to enrich our library. After that, we aim to extend transformations to define auto-adaptable mediums that embed many variants and that could change their internal deployed structures according to environment evolutions.

References

1. Connor, M.: Sadt - structured analysis and design technique. Technical Report 9595-7, Softech (May 1980)
2. Jackson, M.: System Development. Prentice-Hall, Englewood Cliffs (1983)
3. Cariou, E., Beugnard, A.: The specification of UML collaboration as interaction component. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 352–367. Springer, Heidelberg (2002)
4. Matougui, S., Beugnard, A.: Two ways of implementing software connections among distributed components. In: International Symposium on Distributed Objects and Applications, Agia Napa, Cyprus (October 31 - November 2, 2005)
5. Cariou, E., Beugnard, A., Jézéquel, J.M.: An archictecture and a process for implementing distributed collaborations. In: The 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2002), Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland (September 17 - 20, 2002)

6. Kaboré, E., Beugnard, A.: Conception de composants répartis par transformations de modèle. In: Journées de l'Ingénierie Dirigée par les Modèles, Toulouse, France, pp. 117–131 (March 29–30, 2007)
7. Cariou, E.: Contribution à un processus de réification d'abstraction de communication. Thèse de doctorat, Université de Rennes 1 (June 2003)
8. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: ACM SIGCOMM Conference, San Diego (2001)
9. Rowstron, A., Drusche, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
10. Massachusetts Institute of Technology: lsd (2004), `http://www.pdos.lcs.mit.edu/chord/`
11. Rodriguez, A., Killian, C., Bhat, S., Kostic, D., Vahdat, A.: Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In: USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004) (2004)
12. Kaboré, E.: Metamodel definitions (2008), `http://stockage.univ-brest.fr/~kabore/`
13. Dabek, F., Zhao, B., Drushcel, P., Kubiatowicz, J., Stoica, I.: Towards a common api for structured peer-to-peer overlays. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, Springer, Heidelberg (2003)
14. Kaboré, E., Beugnard, A.: On the benefits of using model transformations to describe components design process. In: Twelfth International Workshop on Component-Oriented Programming (WCOP 2007), at ECOOP 2007, Berlin, Germany (July 2007)
15. Kaboré, E., Beugnard, A.: Automatisation d'un processus de conception par transformations de modèles. L'Objet 13(4), 105 (2007)
16. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: Briand, S.K.L. (ed.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
17. D'Souza, D., Wills, A.C.: Objects, Components and Framework with UML: The Catalysis Approach. Addison-Wesley, Reading (1998)
18. Abrial, J.R.: The B-Book, Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
19. Sneed, H.M.: The drawbacks of model-driven software evolution. In: Workshop on Model-Driven Software Evolution, IEEE - CSMR 2007 11th European Conference on Software Maintenance and Reengineering "Software Evolution in Complex Software Intensive Systems", Amsterdam, the Netherlands, March 20-23 (2007)
20. Muller, A., Caron, O., Carré, B., Vanwormhoudt, G.: On some properties of parametrized model application. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 130–140. Springer, Heidelberg (2005)
21. Mens, K., Lopes, C., Tekinerdogan, B., Kiczales, G.: Aspect-oriented programming. In: Bosch, J., Mitchell, S. (eds.) ECOOP 1997 Workshops. LNCS, vol. 1357, Springer, Heidelberg (1998)
22. Sunyé, G., Guennec, A.L., Jézéquel, J.-M.: Design pattern application in UML. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 44–62. Springer, Heidelberg (2000)