On the benefits of using model transformations to describe components design process

Eveline Kaboré, Antoine Beugnard
ENST Bretagne
Technopôle Brest-Iroise
CS 83818 - 29238 Brest Cedex 3 - France

Abstract

In this paper we show how model transformations can be used to describe and automatize component design process through an example of a communication component. Automatizing parts of the design with transformations allows to trace the design, defend such or such design choice and build non functional variants of the same component.

1 Introduction

Models are widely used in sciences and have become an unavoidable tool for software designers and implementors. Models were used in many development methods such as SADT [8], JSD [6], etc. They allow to describe different aspects of a system: structural, functional, behavioral, temporal, etc. Models allow the description of the system to be developed at different stages with various levels of details. The Unified Modelling 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; it is mainly good-practices, know-how and methods more or less formalized. One of the last great advances in software engineering was the introduction of patterns (especially design patterns) as a semi-formalization of good (or bad) practices as catalog of patterns expressed in a language of pattern. The formalization and the clarification of the process of elaborating models are the next challenge. Considering the processes of elaborating and refining models as an activity that can be described with a dedicated language is in our point of view, a revolution.

We show in this article how model transformations can be used to automatize the design and implementation process of a software component. Model transformation languages can hence be considered as a language dedicated to process modelling. In order to confront an idea to a real development process we have developed communication components using transformation techniques. Design choices are left to the designer, but the refinement process is implemented as transformations that are automatically applied once selected.

Communication components are components that are dedicated to communication and hence are naturally distributed. In [3], E. Cariou presented a manual transformation process that shows how the abstract specification can produce many implementation variants: one can be centralized, another be distributed without replication of data, and another with replication. The interest of this approach is that all variants are functionally substitutable but offer different non functional features. Communication components appear to be an interesting candidate to study the design process as a sequence of transformations. Due to its distributed deployed structure, many algorithmic variants may be chosen.

The rest of the article is organized as follows. The next section introduces the starting point of the process: the abstract specification of a communication component. Section 3 suggests the whole design process with models, meta-models and transformations. Section 4 evaluate the benefits of using model transformation to describe components design process. Some related works are presented before the conclusion.

2 Communication component: medium

In order to elaborate a better understanding of model transformations as design choices implementation, we have restricted their use to special communication abstractions called mediums [2].
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 would allow the separation of two concerns: functional concern described by components and communication concern described by mediums.

Example. As an illustration, let us consider the example 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 figure 1.

![Figure 1: An example of communication component: reservation medium](image)

Specification. The specification attempts to describe the medium contract from its user’s point of view as illustrated in figure 1. As a ‘classical’ component, a medium is specified through a set of offered and required services. For each role, a medium defines an interface for offered services and an interface for required services. Each offered and required service should be specified.

Mediums are specified in UML. The specification of a medium in UML contains two aspects: structural aspect described by a class diagram and behavioral aspect described through other UML features such as interaction messages, statecharts and OCL. Figure 2 presents a specification of the reservation medium in a UML class diagram.

![Figure 2: Reservation medium abstract specification](image)

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 distribution of this abstraction possible. The single software component which represents the medium at the abstract level is split into small implementation components called role managers. A different role manager is locally associated with each 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. This architecture presents two main advantages: it allows several implementations of the same abstract medium model, depending on how role managers cooperate, and a good separation of functional and interactions concerns from specification to implementation.

The next section sketches out the full development process and its implementation with models and transformations.

3 Outline of the design process

The entry point of the design process is the medium abstract specification model (figure 3). Our aim is to produce a distributed medium implementation model from this abstract model. This distributed implementation model should preserve all the functionalities of the medium and match all the deployment constraints of the medium. In order to reach this goal, we define a medium refinement process
containing the three following steps.

3.1 Step 1: Introducing the distributed architecture in the medium model

The first step consists in transforming the single UML class which represents the medium in its abstract specification model into an aggregation of Managers in order to match the deployment architecture. This transformation is clearly described in [2]. Figure 4 illustrates the result of this transformation in the reservation medium abstract model (figure 2). In short, the transformation:

- associates a Manager to each role (Reserver-Manager for the role Reserver and SourceManager for the role Source);

- implements in each Manager all the services offered by the medium to its associated role (setReservationIdSet in SourceManager, reserve and cancel in ReserverManager);

- translates all the references of each role on its associated Manager (property reserved on ReserverManager).

Figure 4: Managers introduction into the reservation medium

3.2 Step 2: Specifying the medium actual design alternatives

3.2.1 Objective and principle

We propose a set of design alternatives to the medium designer. In the case of the reservation medium data management for example, the designer will have to choose among data structures (that will be used to express the communication semantic), distributed protocols (that will be used to define the repartition strategy of the medium data) and data representation formats (that will be used to represent the distributed data structure in each storage node). This is illustrated in the following example.

Example: design alternatives for the reservation medium

- Example of a centralized implementation. For a small regional company, reservation management can be concentrated on a single role manager: the manager which is associated to the company (SourceManager in figure 4) for example. The other role managers associated to reservers act as proxies. In this case, the designer can represent the identifiers of reservation seats (property available in figure 2) by a list that will be stored in the SourceManager.

- Example of a distributed implementation. For a big company with hundreds of flights and thousands of travel agencies located worldwide, each agency can locally manage its reservations in order to increase the performance and the reliability of the application. In that case, the designer can use a set to represent the identifiers of reservation seats. It can distributed this set between reservers using the protocol Chord [16] and implements it using the algorithm defined by the MIT [9]. Finally,
the designer can use a hashtable to locally represent the identifiers stored on each server manager.

Other design alternatives can be proposed to deal with non-functional properties such as security, reliability, etc. Thus the designer chooses each design alternative according to its application features and constitutes the actual medium implementation strategy.

### 3.2.2 Modelling design alternatives

In order to facilitate the definition and the reuse of transformations, each chosen design alternative is defined in a model which is injected into the medium abstract model to produce the distributed medium implementation model. Hence, we define a metamodel for each design alternative. For the sake of brevity we only show in figure 5 a view of the distributed protocol definition metamodel.

**Distributed protocol metamodel.** We define a distributed protocol by a set of objects called ProtocolObject (figure 5). A ProtocolObject is an object that can execute the behaviour of a distributed protocol. Each ProtocolObject is implemented by a specific algorithm (ProtocolObjectAlgorithm). The main goal of the distributed protocol metamodel consists in defining a common interface for all distributed protocols that will be used in the context of mediums. Such interfaces are proposed in [4, 13, 14]. The IProtocolObjectServices interface exported by the distributed protocol definition metamodel is similar to the interface defined in [4]. This interface defines services for three main distributed application abstractions: the DHT (Distributed Hash Tables), the DOLR (Decentralized Object Location and Routing) and the 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).

**Use of the distributed protocol metamodel.** To define a distributed protocol model, we describe each ProtocolObject and its implementation algorithms and we implement the interface IProtocolObjectServices for this protocol by defining its services. This is illustrated in the following example.

**Example: a view of the Chord protocol model.** In short, the initial specification of the Chord protocol defined in [16] exports one role and provides five services described in table 1.

<table>
<thead>
<tr>
<th>Services</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert (key, value)</td>
<td>inserts a key/value binding at r distinct nodes.</td>
</tr>
<tr>
<td>lookup(key)</td>
<td>returns a value associated with the key</td>
</tr>
<tr>
<td>update(key, newval)</td>
<td>inserts the key/newval binding at r nodes</td>
</tr>
<tr>
<td>join (n)</td>
<td>add a node to the Chord system</td>
</tr>
<tr>
<td>leave()</td>
<td>leave the Chord system</td>
</tr>
</tbody>
</table>

Table 1: A view of the Chord protocol API [16]
the class `MACEDONAlgorithm`. The class `MITAlgorithm` encapsulates the Chord protocol implementation algorithm proposed by the MIT. The class `MACEDONAlgorithm` encapsulates the Chord protocol implementation algorithm proposed by the MACEDON framework [13]. Other implementation algorithms of the Chord protocol can be added to the model.

In order to encapsulate each implementation algorithm of the Chord protocol, we implement the `insert`, `lookup`, `update`, `join` and `leave` services (described in table 1) on top of the `IProtocolObjectServices` interface. As an illustration, a simple implementation of `insert` routes an `INSERT` message containing `value` and the local node’s nodehandle\(^1\), \(S\), using `route(key, [INSERT, value, S], NULL)`. The key’s root, upon receiving the message, stores the (key, value) pair in its local storage. In this definition of `route(key, [INSERT, value, S], NULL)`,

- the first parameter `key` represents the identifier of the local node;
- the second parameter `[INSERT, value, S]` represents the message. This message contains three information: the type of the message `INSERT`, the `value` to be inserted and the local node’s nodehandle, \(S\).
- the last parameter is an optional argument which is used to specify a node that should be used as a first hop in routing the message. No first hop node is specified in this example: the value is `NULL`.

The `lookup`, `update`, `join` and `leave` services can be implemented in the same way. We note that each implementation algorithm of the Chord protocol has its own definitions of information contained in the messages, algorithms used in routing messages, etc.

### 3.3 Step 3: Merging the actual design alternatives in the medium model

The third step consists in merging the actual design alternatives defined in the previous step into the medium deployment abstract model obtained in the first step. We perform two kind of transformations to reach this goal: a structural transformation and a behavioral transformation. In short, each transformation is specified using a pre-condition, a post-condition and a set of actions. All the transformations are defined using the model tranformation language *kermeta* [12]. The technical details of the specification, the implementation and the validation of transformations are out of the scope of this paper. These details are clearly described in [7]. Here we only show the objective and the principle of each kind of transformation.

#### 3.3.1 Structural transformation

The structural transformation introduces elements\(^2\) which will be used to ensure each data distribution services. Two kinds of elements are created for this purpose: elements which ensure distributed data access services (small eclipse in figure 7) and elements which ensure data distribution services (big circle in figure 7). Figure 7 illustrates the introduction of these elements in the reservation medium model for the distributed implementation strategy illustrated in the previous section. For the sake of simplicity, we only present the new structure of the `ReserverManager`.

![Outline of data managers introduction in the reservation medium](image)

- the class `SetDataManager` implements all the functionalities that are necessary to distribute a set. The distributed protocol functionalities are encapsulated in the class `ProtocolObject`. The set data access primitive definitions are encapsulated in the class `SetDefaultAlgorithm`. The definition of primitives that are used to read and/or write a piece of data stored in memory is encapsulated in the class `DataFormat`.
- The class `SetObject` implements all the functionalities that are necessary to access the distributed set.

\(^1\)A nodehandle encapsulates the transport address and identifier of a node in the system. The transport address might be, for example, an IP address and port (see [4] for more details).

\(^2\)The technical details of these element definitions will not be given in this paper.
Since the identifiers of reservation seats are distributed between reservers, the new structure of the SourceManager contains only elements which ensure data access services. No modification is performed on all the other classes of the medium model.

### 3.3.2 Behavioral transformation

The behavioral transformation implements all abstract operations in the medium model obtained in the second step. Two kinds of transformations are performed for this purpose: algorithmic transformations and configuration transformations.

**Algorithmic transformations** denote transformations that are used to generate abstract operation implementation algorithms. As an illustration, the following example show a view of the code which is automatically generated by an algorithmic transformation in order to implement the add operation of the type set.

```plaintext
class SetDefaultAlgorithm inherits ISetServices { ...
    method add( element : Element ) : Boolean
        from ISetServices is
        do
            if (self.dataManager.lookup(element) == void)
                then
                    result := self.dataManager.place(element)
                else result := false
            end
        end
}
```

In this example, the add operation invokes the lookup service defined in its associated data manager to check the existence of the current element in the set. If the element is already in the set, it returns false, otherwise, it adds the element in the set using the service place defined in its associated data manager.

**Configuration transformations** denote transformations that are used to instanciate the appropriate objects according to the designer choices. Here is an outline of the result of the configuration transformation which is used to instanciate the appropriate objects for the reservation medium according to the distributed implementation strategy specified in the previous the section.

```plaintext
class IReserverManager { ...
    operation connect() is
        do
(1) available := SetDataObject.new
(2) availableDataManager := SetDataManager.new
(3) availableDataManager.protocolObject := ChordProtocolObject.new
(4) availableDataManager.protocolObject.algorithm := MITAlgorithm.new
(5) availableDataManager.setAlgorithm := SetDefaultAlgorithm.new
end}
```

In the connect operation, instruction (1) instantiates a new SetObject and assigns it to available (figure 7). Instruction (2) instantiates a new SetDataManager and assigns it to availableDataManager (figure 7). Instruction (3) instantiates a new ChordProtocolObject and assigns it to the property representing the protocol object in availableDataManager. Instruction (4) instantiates a newMITAlgorithm and assigns it to the property which represents the chord and instruction (5).

### 4 Benefits of the approach

The benefits of using model transformation to describe components design process is 6 fold.

1. The process can produce many implementation variants from the same abstract specification of a medium.
2. The process generates automatically the appropriate code, structure and configuration of the medium according to its actual implementation strategy.
3. The process is explicit; it is described by the transformations and all models and meta models used.
4. The process is traceable. Being explicit, transformations could be memorized in a tool and be undone.
5. The process is reusable. Applying the process to other initial models is direct. We applied our transformations to abstract data types considered as distributed memories with no difficulties.
6. The process is extensible. New models conforming to the meta models used can be added. We added new data type to implement association links very simply.

---

3 The method connect is invoked at the connexion of each manager.
However, this approach presents a limitation in the sense that we should define each design alternative according to its metamodel.

5 Related works

Most methodologies are informally described. They suggest a process which, in the most formalized cases, rely on contracts [5] or mathematical refinements like the B-method [1]. B defines a language and a refinement methodology. It is an algebraic specification language that is supported by tools that help refining specification safely. Each step of the process generates proof requirement the developer has to demonstrate, either manually or automatically. Some critical systems were 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 [15] criticizes the model driven approach. He argues that model-driven tools (1) magnify the mistakes made in the problem definition, (2) create an additional semantic level to be maintained, (3) distort the image of what the program is really like, (4) complicate the maintenance process by creating redundant descriptions which have to be maintained in parallel, (5) 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 assembly was replaced by high level programming languages. We agree tools are not mature. Our experiment shows that transformations may help explicit the process and simplify the maintenance, if models are defined well enough. Other experiments [11] tend to prove that model composition (hence a bottom-up approach) is possible.

This compositional approach looks like Aspect Oriented Modelling [10]. This approach recommends to separate 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 approach (but less re-usable) were a balance may be found between the meta-model definition of the concern and its composition operation implemented as a transformation.

Model transformations are widely used on 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 applying design patterns [17]. Once again, the structural part is rather well implemented, but the collaboration one is still research in progress. Our experiment relies on all those works on model transformations and tries to demonstrate how (under which conditions) all these steps could be integrated in a full design process.

6 Conclusion

We have shown in this article how model transformations can be used to automatize the design and implementation process of a software component. The application context was strongly constrained. The specification rules of communication components are used as preconditions of our first transformations. Models and metamodels used are also defined to work together. We wanted to demonstrate that when the domain is well defined and when the design process is well understood, it should be possible to automatize the whole design process with a set of model transformations. The example used is large enough and the deployment target complex enough to give a good indication of what transformation driven design process would look like.

Our experiment focuses on design, implementing design choices as transformations. We believe transformations should be considered in metrics computation, quality evaluation, test generation, ... in all the development life cycle in short. Systems (viewed as a product) have their own dedicated languages: modelling or programming languages. We argue that transformation languages should be considered as dedicated process languages.

References


4Patterns 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?


