Rapid prototyping of formally modelled distributed systems

Buchs, Didier; Buffo, Mathieu

Rapid Prototyping of Formally Modelled Distributed Systems

Didier Buchs
Software Engineering Laboratory
Swiss Federal Institute of Technology
1015 Lausanne SWITZERLAND
Didier.Buchs@epfl.ch

Mathieu Buffo
Software Engineering Laboratory
Swiss Federal Institute of Technology
1015 Lausanne SWITZERLAND
Mathieu.Buffo@epfl.ch

Abstract

This paper presents various kinds of prototypes, used in the prototyping of formally modelled distributed systems. It presents the notions of prototyping techniques and prototype evolution, and shows how to relate them to the software life-cycle. It is illustrated through the use of the formal modelling language for distributed systems CO-OPN$_{/2}$.

1 Introduction

Application prototyping is a technique that helps software developers in understanding the functionalities proposed by a given system, and in tuning their realizations. Rapid prototyping (i.e. prototyping from the very first steps of software development) of formally modelled systems is a very useful idea when dealing with the development of critical systems. In this case, various techniques were proposed for the prototyping of formal models, such as algebraic data types or Petri nets. However, concrete prototyping techniques are rarely proposed for incremental development in which prototyping must be performed at various level of the development life-cycle.

This paper identifies various kinds of prototypes used during the development of formally modelled distributed systems. These various kinds of prototypes are definitely fruitful as they cover the whole process of software development, from the first specification to the final implementation. The aim of this paper is exhibited through the use of the formal modelling language CO-OPN$_{/2}$, typically used during the development of critical distributed systems [7], as well as the use of Java as the target implementation language.

Section 2 briefly introduces CO-OPN$_{/2}$, allowing the reader to understand the formal basis of this paper. Then, Section 3 presents various kinds of prototypes dealing with the formal modelling of distributed systems, while Section 4 explains how to use them in the life-cycle promoted by CO-OPN$_{/2}$. Section 5 illustrates our aim through a typical scenario. Finally, Section 6 concludes this paper.

2 CO-OPN

CO-OPN$_{/2}$ is an object-oriented modelling language, based on (ADT) Algebraic Data Types, Petri nets, and IWIM coordination models [4]. Hence, CO-OPN$_{/2}$ concrete specifications are collections of ADT, class and coordination *modules* [1] [2]. Syntactically, each module has the same overall structure; it includes an *interface section* defining all elements accessible from the outside, and a *body section* including the local aspects private to the module. Moreover, class and context modules have convenient graphical representations, showing their underlying Petri net model.

Low-level mechanisms and other features dealing specifically with object-orientation, such as genericity, subclassing and sub-typing are out of the scope of this paper, can be found in [1].

2.1 ADT Modules

CO-OPN$_{/2}$ ADT modules define data types by means of algebraic specifications. Each module describes one or more sorts (i.e. names of data types), along with generators and operations on these sorts. The exact definition of the operations is given in the body of the module, by means of equationnal axioms. For instance, Figure 1 describes a (very simple) ADT defining one sort (the booleans) and one operation on this sort (the negation).

```
ADT SimpleBooleans;
Interface
  Sort booleans;
  Generators true, false : ->boolean;
  Operation not_ : boolean->boolean;
Body
  Axioms
    not (true) = false;
    not (false) = true;
End SimpleBooleans;
```

Figure 1 : ADT SimpleBooleans

2.2 Class Modules

CO-OPN$_{/2}$ classes are described by means of modular algebraic Petri nets with particular, parameterised, external transitions, the *methods* of the class. The behaviour of transitions are defined by so-called *behavioural axioms*, corresponding to the axioms in ADT. A method call is achieved by synchronising external transitions, according to the fusion of transitions technique.

Below is the code and the associated Petri net graphics of a class modelling an unusual storage system; it stores boolean values, but delivers the negated ones. The interface defines two methods, for the injection and the ejection of values. The body is actually a textual representation of the associated Petri net. Free variables may be defined and used in the behavioural axioms.

```
Class StrangeStorageSystem;
Interface
  Use SimpleBooleans;
  Methods put _ ,  get _ : boolean;
Body
  Place container _ : boolean;
  Axioms
      put b :: -> container b;
      get b :: container not(b) -> ;
  Where b : boolean;
End StrangeStorageSystem;
```

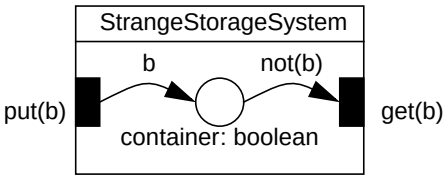

Figure 2 : Class StrangeStorageSystem

2.3 Coordination Modules

A third kind of modules is present in CO-OPN$_{/2}$, the *context* modules[4], which share the same overall structure with ADT and class modules. Basically, context modules allow the modelling of distributed systems, by means of suitable coordination mechanisms, more complex than the fusion of transitions seen above.

Due to lack of space and to the fact that COIL is clearly specific to the coordination theory, context modules are not illustrated here.

3 Prototyping of Formally Modelled Distributed Systems

The goal of this section is the identification of various kinds of prototypes related to formally modelled distributed systems using CO-OPN$_{/2}$. The next section explains how to use these kinds of prototypes in the *software life-cycle* proposed for CO-OPN$_{/2}$.

We decompose the prototyping strategy into two orthogonal dimensions. The first dimension takes into account the techniques used to prototype a system, while the second dimension takes into account about the ability for prototypes to evolve along with the developer's intentions.

3.1 Prototyping Techniques

CO-OPN specifications can be either simulated using an apposite monitor or executed by a specific executable prototype. Prototyping by simulation focuses on the validation of the system functionality with regard to the user requirements. On the other hand, prototyping by execution allows the tuning of design and implementation choices. In the case of distributed systems, an executable prototype itself may be distributed; this also allows the fine tuning of the system's configuration.

These two techniques can be easily applied on various level of abstraction of the CO-OPN$_{/2}$ formal models. In both cases, the CO-OPN$_{/2}$ code will have to be processed by a common front-end which performs lexical, syntactical and static semantic analysis.

Prototyping by simulation. Prototyping by simulation allows the developers to validate their models, by letting them explore and analyse the semantics of their specification. CO-OPN$_{/2}$ models can be simulated using standard rewriting techniques that are compatible with the denotational semantics of ADT and with the structured operational semantics of our algebraic Petri nets. For instance, the following ordered rewrite rules are used for the axioms of the ADT modelling simple booleans we saw above:

```
not (true) -> false;
not (false) -> true;
```

Axioms of the Petri nets part are evaluated in order to compute the possible evolution of the modelled system, represented as a transition system. For instance, we are able to deduce that the class modelling the strange storage system we saw above includes the following behaviour:

$$\{\} \xrightarrow{\texttt{put(true)}} \{\texttt{true}\} \xrightarrow{\texttt{get(false)}} \{\}$$

The simulation performed by using a Prolog engine, solves both the rewrite process and the symbolic manipulation [8]. Due to the versatility of the Prolog interpreter that we use (Prolog with control), it is possible to compute more abstract or more complicated behaviours. For instance, it is possible to perform a symbolic evaluation, such as:

$$\{\} \xrightarrow{\texttt{put(x)}} \{\texttt{true}\} \xrightarrow{\texttt{get(not(x))}} \{\}$$

This behaviour is more abstract, as it shows that the buffer is able to preserve any boolean value.

Prototyping by execution. Executable prototypes allow developers to tune their design and implementation choices, by letting them freely interact and play with their prototypes. When executable prototypes are desired, CO-OPN$_{/2}$ models are translated into Java programs. In the case when the abstract semantics of CO-OPN$_{/2}$ is not executable, an operational semantics is mapped to CO-OPN$_{/2}$, and it is used to perform the translation.

The operational semantics is in essence very close to the original semantics; it actually defines a heuristic for solving the non-deterministic choices found in the original semantics, and it includes explicitly a definition of the negative pre-condition (i.e. it allows the prediction of *impossible* behaviours in a given system, as well as when this behaviour is possible).

This semantics keeps intact the properties of distribution and configuration of systems found in CO-OPN$_{/2}$. Indeed, all implicit object synchronisations occurring during an event is made explicit (there are strong similarities with the semantics of distributed Prolog). Moreover, the management of complex synchronisations, which are atomic from the operational point of view, is based on general nested transaction techniques. Thus distributed prototypes are likely to be defined, for the tuning of the model in an actual environment, which can be a distributed platform.

3.2 Evolutive Prototyping

Another dimension of the prototyping process is the ability to produce prototypes that are able to evolve. This aspect is interesting in the context of the more concrete kind of prototyping method based on the distributed execution of code. It should be noted that the notion of evolution is consistent with the prototyping by simulation techniques, but it requires very specific knowledge and is out of the scope of this paper [5].

The idea is that a prototype should be able to evolve according to the intentions of the developers, by replacing parts of the distributed prototype with compatible ones. Our approach can be characterized by contributions on two levels: first, the class-based separation between the automatically generated and the hand-written code, and second, the use of prototype objects in a systematic software development process

The evolution process starts with an executable distributed prototype in Java, resulting from the prototyping technique by execution described just above. This Java code is actually main-stream object-oriented code, which provides a portable and high-level starting implementation.

The generated class hierarchy is designed so that the developer may then independently derive new sub-classes in order to make the prototype more efficient or to add new functionalities.This process is performed incrementally in order to safely validate the modifications against the semantics of the specification. The resulting prototype can finally be considered as the end-user implementation of the specified software system [3].

The originality of our scheme is that we exploit object-oriented programming techniques in the implementation of formal specifications in order to gain flexibility in the development process. The paradigm of object-orientation permits easily several implementations for a same abstraction. In usual implementation patterns, there is one parent class with several equally important sibling implementations. We propose instead a hierarchy with a privileged implementation, the one generated automatically, which will serve as reference for future hand-written class derivations.

For each CO-OPN$_{/2}$ class, two Java classes are defined, the first one modelling the algorithmic part of the original class, the second one defining an access control policy. By proper use of polymorphism, these classes can export all their operations in an implementation independent manner. The generated concrete class provides a sample internal representation to be used by default; it does not usually redefine any of the inherited routines. From here, both classes are likely to be independently extended, using Java inheritance, to let parts of the prototype evolve. Figure 3 illustrates this organization with one main evolution of the access control and two main evolution branches of the algorithmic part.

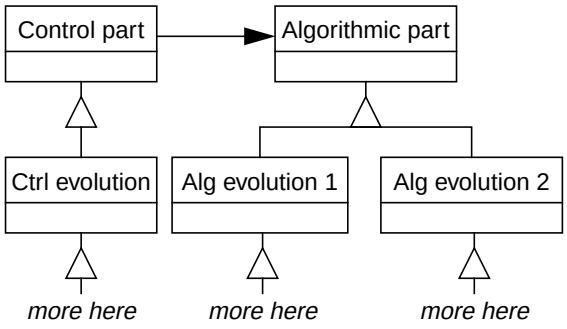

Figure 3 : Organization of Evolutive Prototypes

4 Prototyping and Software Life-Cycle

The life-cycle promoted by the development method associated to CO-OPN$_{/2}$ includes several steps of refinement (in the sense that a refinement details some aspects of the specification). These refinements bring more and more concrete models as long as we progress in the development [6]. This situation is shown in Figure 4.

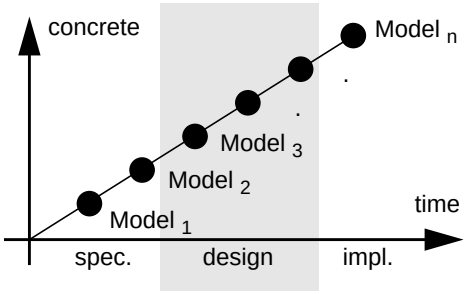

Figure 4 : Life-Cycle Promoted by CO-OPN$_{/2}$

Abstract specifications can only be symbolically interpreted while more concrete ones can be executed using a direct translation of the specification into a programming language. In addition, we remark that the prototypes are more and more incremental as long as we progress in the development. Figure 5 shows this situation. First prototypes are symbolic prototypes (prototyping by simulation) while next ones are executable prototypes. As long as the process goes on, the developer is able to increment more and more its prototypes, allowing thus more and more tuning of the resulting system.

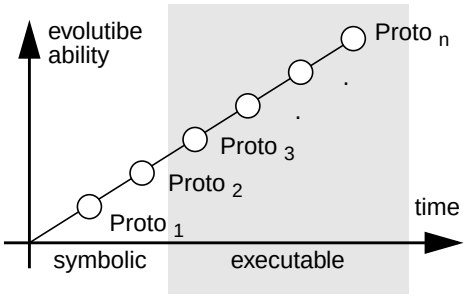

Figure 5 : Incremental Prototyping

According to both figures, we state that in the development method associated with CO-OPN$_{/2}$, the kind of prototype to use is strongly connected to the software life-cycle, and more precisely:

- prototypes used during the *specification phase* are *symbolic prototypes*, with poor abilities for evolution, with poor abilities for fine tuning, but allowing rapid validation of the system;
- *executable prototypes* with poor use of prototype evolution are used during the *design phase*, as they allow the test and the tuning of the main design choices;
- last prototypes obtained during the design phase should be incrementally modified during the *implementation phase* for the fine tuning of the software; The last *evolutive prototype* may be considered as the *resulting programme*.

5 Typical Scenario

We present now an hypothetical software development using the rapid prototyping facilities described above. The aim of this section is *not* to present the development of a realistic case study; due to lack of space, we present here *only* a *typical scenario* where the three kinds of prototypes are used.

The argument of the scenario is the following: assume we develop a model, in which a storage system is needed. We use CO-OPN$_{/2}$ to describe our model, and in particular the storage system. We select Java as the target implementation language. As usual with CO-OPN$_{/2}$, we start with a first abstract object model, and we refine it to obtain a convenient concrete solution. Due to the modularity of CO-OPN$_{/2}$, we are able to focus on a particular component during the refinement process (here the storage system), and to derive various prototypes covering the component itself as well as the whole system.

5.1 Prototypes during the Specification

We create an object model for the storage system, as depicted in Figure 6. We have a class encompassing an algebraic Petri net, the transitions of which are the methods of the class. Due to the properties of Petri nets, this class shows an abstract storage, in which the output priority is not related to the input priority.

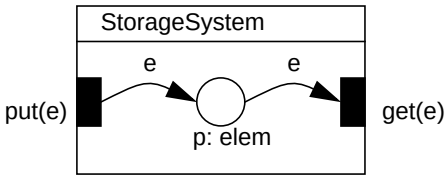

Figure 6 : Abstract Object Model

As soon as we get this class, we can prototype it by simulation. This first prototype may be used to insure the intrinsical properties of the class (in this simple example, it is obvious that the class really acts as desired, but this is not true in general). This prototype gives us confidence in the soundness of the storage system.

Now, assuming that the model of the rest of the system is available, we can prototype the whole system, by simulation again. This second prototype is used to validate the model with regards to the original (and informal) requirements. For instance, this prototype may be used to insure that no queuing strategies have to be imposed by the model to meet the requirements.

5.2 Prototypes during the Design

We refine now our model by making design choices. In the framework of CO-OPN$_{/2}$, this is performed by producing a new CO-OPN$_{/2}$ model, closer to the envisaged solution. In our scenario, we make a design choice regarding the queuing strategy of the storage: we decide to use a FIFO strategy. Figure 7 depicts our refined CO-OPN$_{/2}$ model. The place contains now a unique token, structured as a FIFO queue of elements.

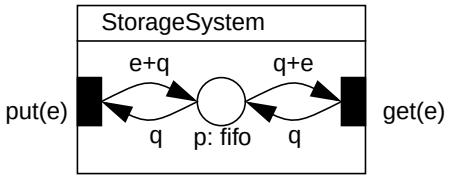

Figure 7 : Refined Object Model

We may now prototype this model, either by simulation or by execution. A prototype by simulation may be used to verify the correctness of the refinement, for instance, by analysing the behaviours of the new model with regards to the original one.

However, the most interesting thing to do now is to derive automatically an executable prototype. We create this prototype by extracting the appropriate information from the CO-OPN$_{/2}$ model. In our case, we derive two classes, one for the Fifo data type, and one for the storage itself. This situation is shown in Figure 8. This prototype can be used to gain confidence in the choice of a Fifo, for instance. In any cases, during this phase of software development, new refinements require the re-generation of new executable prototypes from the CO-OPN$_{/2}$ model.

```
public class Fifo {
  public synchronized void add
                          (Elem e) {
    // generated by the tool
  }
  public synchronized Elem remove() {
    // generated by the tool
  }
}

public class FifoStorageSystem;
  private Fifo fifo=new Fifo();
  public void put(Elem e) {
    fifo.add(e);
  }
  public Elem get() {
    return(fifo.remove());
  }
}
```

Figure 8 : Java Prototype with ad-hoc Fifo

5.3 Prototypes during the implementation

We decide now that the design of our system is complete, and we look for an implementation. We consider the last executable prototype as a starting point and we use the incremental prototyping technique to produce our software. For instance, we can replace (incrementally, without regards to the rest of the system), the ad-hoc Fifo class by a Java Vector, as Vectors are convenient means to define unbounded collection of values in Java. This situation is shown in Figure 9. Readers should note how problems dealing with threads' concurrency are handled in this class, with regards to the previous figure; synchronisations are now attached to the storage system.

```
import java.util.*;

public class FSSVector
          extends FifoStorageSystem;
  private Vector p=new Vector();
  public synchronized void put
                          (Elem e) {
    p.addElement(e);
  }
  public synchronized Elem get() {
    Elem e=p.firstElement();
    p.removeElementAt(0);
    return(e);
  }
}
```

Figure 9 : Java Prototype with a Vector

At this point, we may decide that we must bound its capacity for some good reason. We transform again our prototype, to reflect this choice, and we use a Java (fixed-length) array. This prototype is shown in Figure 10.

```
public class FSSArray
          extends FifoStorageSystem;
  private Elem[] p;
  private int pointer;
  public StorageSystem(int bound){
    p=new Elem[bound];
    pointer=0;
  }
  public synchronized void put
                          (Elem e) {
    p[pointer++]=e;
  }
  public synchronized Elem get() {
    Elem e=p[0];
    --pointer;
    for (int i=0;i<pointer;i++)
                        p[i]=p[i+1];
    return(e);
  }
}
```

Figure 10 : Java Prototype with an Array

Using the evolutive prototyping technique, we will produce many other prototypes, each of them being formally a refinement of the previous one. In this case, the last prototype is actually the desired implementation. For instance, in the case of our storage system, we may derive:

- some prototypes allowing to fix the size of the array;
- then a new prototype with an exception mechanism to prevent problems dealing with the capacity of the buffer;
- then new prototypes with better implementations of the cyclic buffer;
- and finally the last prototype, actually a distributed implementation, with remote procedure calls based on the Java RMI technique.

6 Conclusion

To summarize, in this paper, we identified various kinds of prototypes for formally modelled distributed systems, and we shown how these kinds of prototypes are used in the software development process, according to the software life-cycle promoted by the modelling language CO-OPN$_{/2}$. We shown that prototyping by simulation is useful during the first steps of software development, to validate the model against the requirements. Moreover, we exhibited the fact that - in the framework of CO-OPN$_{/2}$ - prototyping by execution is used to test various design choices, either at the level of component, or at the level of the global system. Finally, we used clearly evolutive prototyping for the fine tuning of the proposed software.

We are working currently on a new prototyping tool based on the prototyping by execution technique. This tool is derived form the work we made about evolutive prototyping [3]. The new tool is based on Java, and it must integrate CORBA *and* RMI facilities, for the integration of the generated prototypes with existing software. This integration is particularly important in our context of evolutive prototyping.

Acknowledgements

We would like to thanks Shane Sendall for his helpful comments on a first draft of this paper.

References

[1] Olivier Biberstein and Didier Buchs. Structured algebraic nets with object-orientation. In *Proceedings of the first international workshop on "Object-Oriented Programming and Models of Concurrency" within the 16th International Conference on Application and Theory of Petri Nets*, Torino, Italy, June 26-30 1995.

[2] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. In G. Agha and F. De Cindio, editors, *Advances in Petri Nets on Object-Orientation*, Lecture Notes in Computer Science. Springer-Verlag, 1998. To appear.

[3] Didier Buchs and Jarle Hulaas. Evolutive prototyping of heterogeneous distributed systems using hierarchical algebraic Petri nets. In *Proceedings of the International Conference on Systems, Man and Cybernetics*, Beijing, China, October 1996. IEEE.

[4] Mathieu Buffo. Experiences in coordination programming. In *Proceedings of the workshops of DEXA'98 (International Conference on Database and Expert Systems Applications)*. IEEE Computer Society, aug 1998.

[5] Christine Choppy and Stéphane Kaplan. Mixing abstract and concrete modules: Specification, development and prototyping. In *12th International Conference on Software Engineering*, pages 173–184, Nice, March 1990.

[6] Giovanna Di Marzo Serugendo and Nicolas Guelfi. Formal development of java based web parallel applications. In *Proceedings of the Hawai International Conference on System Sciences*, 1998.

[7] Giovanna Di Marzo Serugendo, Nicolas Guelfi, Alexander Romanovsky, and Avelino Zorzo. Formal development and validation of the dsgamma system based on CO-OPN/2 and coordinated atomic actions. Technical Report to appear as 1997 - Technical Report of the Esprit Long Term Research Project 20072 "Design For Validation", University of Newcastle Upon Tyne, England, Department of Computing Science, 1997.

[8] Peter Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. sv, Berlin, 1988.