View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

From Formal Specifications to Ready-to-Use Software Components: The Concurrent Object Oriented Petri Net Approach

Stanislav Chachkov
Software Engineering Laboratory
Swiss Federal Institute of Technology Lausanne
1015 Lausanne, SWITZERLAND
Stanislav.Chachkov@epfl.ch

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

Abstract

CO-OPN (Concurrent Object Oriented Petri Net) is a formal specification language for modelling distributed systems; it is based on coordinated algebraic Petri nets. In this paper we describe a method for generating an executable prototype from a CO-OPN specification. We focus our discussion on the generation of executable code for CO-OPN classes. CO-OPN classes are defined using Petri Nets. The main problems arise when implementing synchronization and non-determinism of CO-OPN classes in procedural languages. Our method proposes a solution to these problems. Another interesting aspect of our method is the easy integration of a generated prototype into any existing system. This paper focuses on the generation of Java code that fulfils the Java Beans component architecture, however our approach is also applicable to other object-oriented implementation languages with a component architecture.

1. Introduction

Developing complex reactive software systems requires modelling tools that can capture the properties of the system to develop as well as the structure of the interactions that will be necessary between the software and its environment.

In this paper, we present a formal framework for the development of distributed systems from the modelling phase to the implementation. The approach we propose has adopted the object-oriented paradigm as a structuring principle. We have devised a general formalism that can express both abstract and concrete aspects of systems, with emphasis on the description of concurrency and abstract data types. This approach, called Concurrent Object-Oriented Petri Nets (CO-OPN)[5][2], extends its object-based predecessor [11]. A coordination layer [14] has been developed on top of this formalism in order to be able to deal with a distributed architecture.

In order to produce an implementation from the CO-OPN model we will describe the automatised mapping techniques that we can use to produce programming language code. Apart from the problem of producing programs that respect the CO-OPN model and its particularities (non-determinism, atomicity of the events, modularity induced by Object-Oriented structure,...) we are also particularly interested to match the usual programming principles used in the target language. In this paper we will use the Java notion of component architecture, the JavaBeans model[13]. It is also necessary, in order to cope with incremental refinement of the automatically generated prototype by the developer, to be able to interconnect the produced components with other components or with standard libraries (for instance the Swing Java user interface libraries). We achieve this by introducing transparently the support for transactions and non-determinism, and by fulfilling the rules of the target component model.

In this paper we will first briefly explain how to start from a diagram establishing the interconnection of the application to the outside world elements and how to produce step by step a model that will be used to derive automatically a program. The elements, composing the whole system, can be for instance a lift with its components: motors, cabin, doors, and the controlling software. This example will serve to illustrate in this paper our techniques.

The paper is organized as follows. In Section 2, we discuss the model of the lift system that we can produce using CO-OPN. In section 3, we present our basic mapping method for generating OO code from CO-OPN and the problem of implementing synchronization and non-determinism in Java. In section 4, we discuss how to integrate components in classically produced software.

2. Modelling with CO-OPN

CO-OPN is an object-oriented modelling language, based on Algebraic Data Types (ADT), Petri nets, and IWIM coordination models [7]. Hence, CO-OPN specifications are collections of ADT, class and context (i.e. coordination) *modules* [11]. 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 which are used in this paper, showing their underlying Petri net model. Low-level mechanisms and other features dealing specifically with object-orientation, such as sub-classing and sub-typing, are out of the scope of this paper, and can be found in [2] [5].

2.1 The lift control problem

In order to present a concrete example of modelling, we chose to study a simple but non-trivial example of reactive system, a lift control system. The aim of this study is to elaborate a CO-OPN specification and explain how to generate the corresponding controller in Java. First of all, we will explain the steps that can lead to the CO-OPN model of the lift controller.

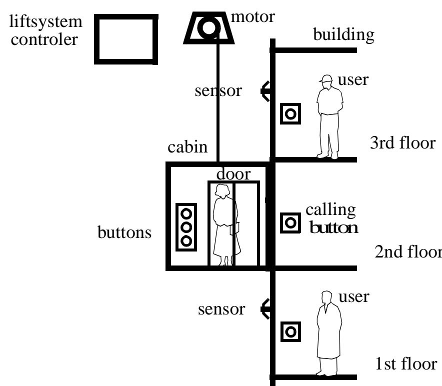

Figure 1: The actors (real life view)

The lift problem entities, composing the whole system (figure 1) can be, for instance, the users, the cabin, the doors, the motors and the controlling software. Moreover, sensors are attached to some entities. In order to simplify the system, calling buttons have a similar effect in the cabin and at the floors.

For such a system we are interested in a first approach to determine the components and their various interactions. This is, for instance, described in the UML collaboration diagram of figure 2. The main concepts that are used to express the structure and the behaviour of the system:

- a coordination model for describing the relations between the system components,
- object orientation for the structure and content of the system,
- causality relations for the dynamic aspects that must be reflected with non-deterministic and concurrent behaviours.

2.2 Introduction to coordination

In this part the various concept of CO-OPN will be introduced in the necessary order for modelling the lift control system. As we use a top-down strategy for modelling, we will first start by presenting the interface of the system given by the top-level coordination entity called LiftSystem context.

A useful approach for building systems composed of many computing entities is to use the high-level concept of *coordination programming* [15]. The term *coordination theory* refers to theories about how coordination can occur in various kinds of systems. We state that coordination is *managing dependencies* among activities.Taking a step further in this direction, it appears that *coordination patterns* are likely to be applied from the start of the *design phase* of the software development. This fact gave birth to the notion of *coordination development* [7]. This process involves the use of specific coordination models and languages, adapted to the specific needs encountered during the design phase as expressed in the lift example.

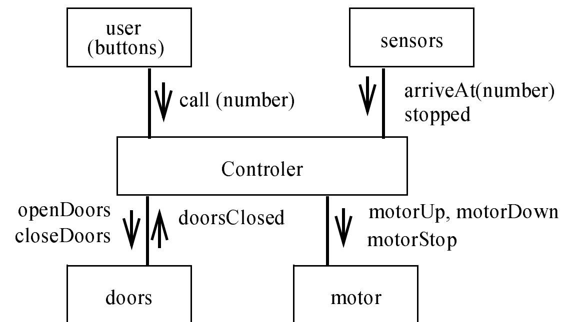

Figure 2: The actors and their interaction (controller view)

Coordination models can be categorised as either *exogenous* or *endogenous*. Exogenous coordination models separate computation and coordination tasks by devoting different modules to different concerns, while endogenous models provide coordination primitives that must be incorporated within computation tasks. Coordination models can also be categorised as either *control-* or *data-driven*. Control-driven models tend to centre around the notion of processing or flow of control, while data-driven models are essentially concerned with what happens to data. The *IWIM* (*Idealized Workers, Idealized Managers*) is a general coordination model which can be exactly characterised with the following two keywords: *exogenous* and *control-driven*. Probably the most known IWIM model is Darwin [16].

Due to their intrinsic nature, IWIM models are particularly well suited for the coordination of software elements

during the design phase [6]. The coordination layer of CO-OPN [7] [5] [6] is a coordination language based on a IWIM model, suited for the formal coordination of object-oriented systems. CO-OPN context modules define the *coordination* entities, while CO-OPN classes (and objects) define the basic *coordinated* entities of a system.

2.3 Coordination with contexts

In figure 3 we can see the interface of the `LiftSystem` controller context, with the input (black rectangle) and output events of this context (white rectangle).

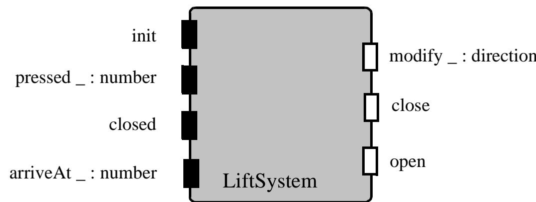

Figure 3: The Controller context

This black box contains sub-components that interact to provide the controller behaviour. The controller sub-components are two objects which are instances of the classes Cabin and Building as depicted in figure 4. In this picture the directed arcs between methods or gates are used to define strong synchronization between events. In CO-OPN, it means that the firing of synchronized events is atomic.

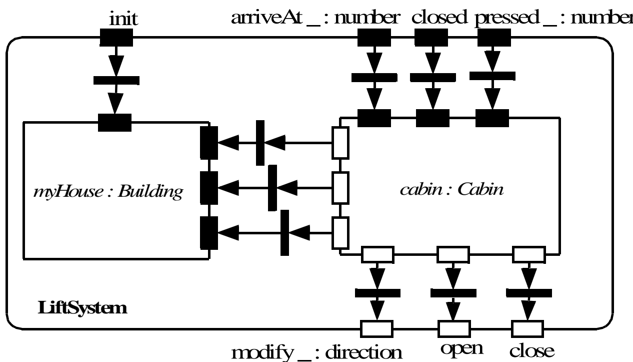

Figure 4: The static components instances inside the Controller context

The `cabin` component is devoted to managing the cabin movement, and it is also used as the main interface with the outside world through the `LiftSystem` context. The `myHouse` component collects the requests to the specific floor object. The floor objects, one object for each floor in the building, are instantiated by means of the building's `init` method that must be called before any other method.

Before explaining the components, we will quickly give an outline of the way values can be defined in CO-OPN, using algebraic data types.

2.4 ADT modules

CO-OPN ADT modules define data types by means of algebraic specifications. Each module introduces one or more sorts (i.e. names of data types), along with generators and operations on those sorts. The properties of the operations are given in the body of the module, by means of positive conditional equational axioms. Operations are partial deterministic functions.

For example, figure 5 describes the ADT module defining one sort, the `direction`, three generators: `UP`, `DOWN`, `STOP`, and two operations on this sort: `opposite _` and `way from _ to _`. The first three axioms give the definition of the opposite direction, while the last two axioms compute the direction for going from one floor to another. Having the ADT, it is possible to describe the dynamic components of a CO-OPN specification: the classes.

```
ADT Direction;
Interface
 Use  Number;
 Sort direction;
 Generators
   UP, DOWN, STOP: -> direction;
 Operations
   opposite _ : direction -> direction;
   wayFrom _ to _ : number, number-> direction;
Body
 Axioms
   (opposite STOP) = STOP;
   (opposite UP) = DOWN;
   (opposite DOWN) = UP;
   (n > m) = true => (wayFrom n to m) = DOWN;
   (n > m) = false => (wayFrom n to m) = UP;
 Where
   n ,m : number;
End Direction;
```

Figure 5: The Direction ADT

2.5 Modelling classes

In this subsection we will show more detail on the classes that compose the `LiftControl` system, and using this example explain the main elements of a CO-OPN model. The `Building` class encapsulates instances of the class `Floors`. Each floor object stores information on whether the floor is requested for a stop or not.

CO-OPN classes are described by means of modular algebraic Petri nets with particular parameterized external transitions which are the *methods* of the class. The behaviour of transitions are defined by so-called *behavioural axioms*, similar to the axioms in an ADT. A method call is

achieved by synchronizing external transitions, according to the fusion of transitions technique. The axioms have the following shape:

```
Cond => eventname With synchro : pre -> post
```

In which the terms have the following meaning:

- `Cond` is a set of equational conditions — the guard;
- `eventname` is the name of the methods with the algebraic term parameters;
- `synchro` is the synchronization expression defining the policy of transactional interaction of this event with other events, the dot notation is used to express events of specific objects and the synchronization operators are the sequence, the simultaneity and the non-determinism.
- `Pre` and `Post` are the usual Petri Net flow relations determining what is consumed and what is produced in the object state places.

CO-OPN provides tools for the management of graphical and textual representations. Figure 7 shows the partial class description net corresponding to a simple class `Floors` in a textual form, the equivalent graphical description (the Petri Net plus additional informations concerning the interface) is depicted in figure 7.

```
Class Floors;
Interface
 Use Number, Booleans;
 Type floor;
 Methods
   stopWasRequested _ : number;
   stopWasNotRequested _ : number;
(...)
 Creation new _ : number;
Body
 Place floorNumber _ _ : number, boolean;
 Axioms
   stopWasRequested n::
     floorNumber n true -> floorNumber n false;
   (n = nb) = false =>
     stopWasNotRequested n::
     floorNumber nb b -> floorNumber nb b;
        (...)
   new n:: -> floorNumber n false;
 Where
   n, nb : number; b : boolean;
End Floors;
```

Figure 6: The Floors class textual description, omissions are indicated by (...)

This floor class provides methods to modify, in different ways, the state of the floors. For instance, the method `StopWasRequested` highlighted in figure 7 is fireable if a stop is requested at the specified floor `n`, the effect of the firing of this method is to put the floor in the state not requested. The `new` method creates new instances of the `Floors` class with `FloorNumber n false`.

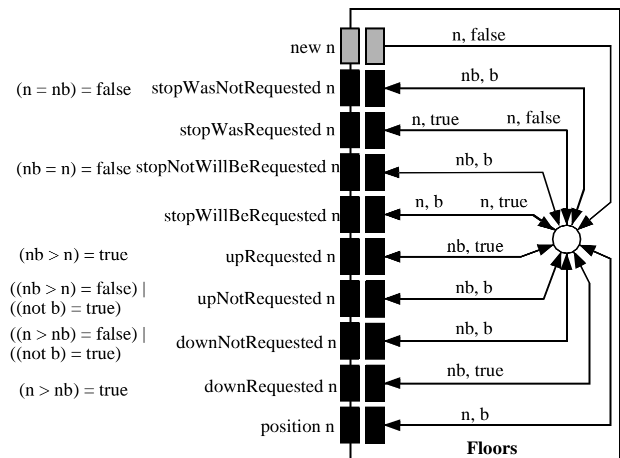

Figure 7: The Floors class graphical description

We will explain the behaviour of the cabin when the event `closed` (door closed) is received. The direction of the movement of the cabin in the lift system must be defined according to the floors that have been requested. In figure 8 two cases are formalised.

The first case is when the lift rests at the same floor. The second case is when the lift has already a destination, and therefore continues to move.

3. Translation of CO-OPN to programming languages

3.1 Introduction

The generation process takes a CO-OPN specification as a parameter and produces a set of Java classes. The object structure of a CO-OPN specification is preserved by the generated code. One of our primary goals was to find a "natural" mapping between CO-OPN and Java. In such a mapping, standard CO-OPN features like methods or gates are associated to standard Java features, methods or events, respectively. As a result, the interface part of a generated Java component is similar to the interface of the corresponding CO-OPN component, and as a consequence is also easy to understand/use by a human programmer. Because the produced Java classes satisfy the requirements of a Java component, namely JavaBeans, they are easy to use in a development tool.

Some powerful aspects of CO-OPN, such as atomic concurrent synchronizations or non-determinism, do not have a direct equivalent in Java, consequently they are non-trivial to implement. These aspects are, as much as possible, hidden in private parts of the generated code.

Finally, the structure of the code is designed so that it

```
Class Cabin;
Interface
 Use Number; Direction;
 Type  cabin;
 Gates
   modify _: direction; close;
(...)
 Methods closed; arriveAt _ : number;
(...)
Body
 Use Door;
 Places
   move _ : direction; stopped _ : boolean;
   door _ : door; localisation _ : number;
  Initial
   stopped true; move STOP;
   door open; localisation 1;
  Axioms
   ax1: closed::
      move STOP,door p -> move STOP,door closed;
   ax2: (! d = STOP) & this = Self =>
    closed With this . modify (d)::
      move d, stopped b, door p ->
      move d, stopped not (b), door closed;
(...)
  Where
    this : cabin;  p : door;
    b : boolean;  d : direction;
End Cabin;
```

Figure 8: The Cabin class textual partial description

is easy to change the implementation in a modular way as justified in [9] and initially proposed in [8]. Because of a lack of space, this aspect will not be discussed further.

3.2 Code generation for ADT

The lift specification uses some ADTs. One of them is `List` (Figure 9) which is used by `Building` class to store a list of floors. In CO-OPN types and functions are dissociated. An ADT module can declare some types and some functions. In general, there is no reason to associate a particular function to a particular type. To avoid this choice two hierarchies of classes are built for each ADT module. The first one is the class hierarchy representing sorts and terms. The purpose of this hierarchy is to build the representation of values. The second one is the class hierarchy representing the operations. The purpose of this one is the implementation of the functional part of the ADT.

Before going into a detailed explanation of the generated classes and related hierarchies, we will discuss Figure 10. It provides a synthetic view of the generated classes, and how they are related, for the example of the List ADT. We see (in a UML-like notation) the classes representing sorts and terms in the right part, and the class implementing the operations and generators in the left part. The next sub-sections are dedicated to these two parts.

```
ADT List;
Interface
 Sort list;
 Generators
   [ ] : -> list;
   _ ' _ : floor, list -> list;
 Operations
   # _ : list -> natural;
   head _ : list -> floor;
   tail _ : list -> list;
Body
 Axioms
   (# [ ]) = 0;
   (# (f ' l)) = (succ (# l));
   (...)
 Where f : floor; l : list;
End List;
```

Figure 9: List ADT (partial)

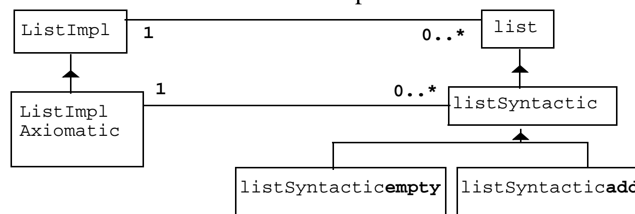

Figure 10: Synthetic view

3.3 Class hierarchy representing sorts and terms

For each sort, we define an interface representing it in the Java prototype. This interface mainly defines “test” and “inverse” methods, allowing the analysis of the syntactic structure of terms. The purpose of test methods is to find the generator used to build a term, while inverse methods allow one to retrieve the parameters of a term generator (i.e. subterms). For optimization purposes, objects that implement this interface should be immutable.

Figure 11 shows the interface corresponding to the sort `list` (figure 9). We can see two test methods, namely `isempty` and `isadd`, and two inverse methods, corresponding to the two parameters of the generator `add`. Readers should note that the original CO-OPN names are translated into valid Java indentifiers. For instance, we use the identifier `add` for the CO-OPN name `_'_` and `empty` for `[ ]`. The translation is derived from annotations bound to the original CO-OPN modules.

Along with the sort interface seen above, we produce a standard implementation based on the syntactic representation. To achieve this goal, we first define an empty marker interface for syntactic representations. For instance, Figure 12 shows such a marker for the lists.

Then, we define one class for each generator. Values

```
public interface list{
  public boolean isempty();
  public boolean isadd();
  public array getaddChild1();
  public floor getaddChild2();
}
```

Figure 11 : Interface list

```
public interface listSyntactic
                        extends list{}
```

Figure 12: Syntactic Representation of lists

are represented as syntactic trees, where nodes are instances of those "generator" classes. In the case of the Lists, we produce two classes `listSyntacticempty` and `listSyntacticadd`. Figure 13 details the class recording non-empty lists. The reader can observe how the test and inverse methods are implemented.

```
class listSyntacticadd
             implements listSyntactic {
  private list arg1;
  private floor arg2;

  public listSyntacticadd(list arg1,
                  floor arg2) {
    this.arg1=arg1;
    this.arg2=arg2;
  }
  public boolean isempty(){return false;}
  public boolean isadd(){return true;}

  public list getaddChild1(){return arg1;}
  public floor getaddChild2(){return arg2;}
}
```

Figure 13: Syntactic non-empty lists

3.4 Hierarchy of classes representing operations

Besides the class hierarchy allowing the representation of values, we must define classes implementing the various functions of each CO-OPN module. Actually, each class implementing the functional part of the ADT module includes factory methods that correspond to generators, methods implementing the functions defined in the module and methods implementing the native CO-OPN equality predicate on terms (one method per sort). Actually, for each ADT module, we produce an abstract class, that defines the signature of methods corresponding to functions and generators found in the module. Also, a default implementation of the equality method, based on a syntactic comparison, is provided. Figure 14 shows the abstract implementation

```
public abstract class ListImpl{
  public abstract list empty();
  public abstract list
      add(list arg1, floor arg2);
  public abstract floor head(list arg1);
  public abstract list tail(list arg1);
  public abstract natural
                 size(list arg1);
  public boolean COOPNEquals(list arg1,
                             list arg2){
    //syntactic equality, details omitted
  }}
```

Figure 14: Abstract implementation of lists

The default implementation of this class is based on the syntactic representation of values and on the axioms of the ADT. For each operation in ADT, we orient the axioms into a set of rewrite rules [10]. The core of the associated method is mainly a multiple choice between a set of applicable axiom. Undefined cases throw exceptions.

```
public class ListImplAxiomatic
                        extends ListImpl{
  public natural size(list arg1){
    if (arg1.isempty())
     return NaturalsImpl._0();

    if (arg1.isadd())
     return
      NaturalsImpl.succ(size(tail(arg1)));
    throw new COOPNNoSemanticsException();
  }}
```

Figure 15: Axiomatic representation

Figure 15 shows a fragment of the default axiomatic implementation of the lists highlighting the operation "`size`". We see that both of the two axioms defining this operation in the ADT (see figure 9) are represented by two if-statements in the Java method. If no axioms apply, a `COOPNNoSemanticsException` is thrown, signalling that the operation is undefined on given parameters (it never occurs for the `size` method).The reader can observe how the test method "`isadd`" is used.

3.5 Specific class principles

Syntactically and structurally CO-OPN Classes and ADT modules are rather similar (that is also true for CO-OPN contexts). Those similarities include: object structure, separation of module in public and private part, operations defined by axioms. Those similarities allow us to reuse some concepts and also code from the ADT part of the prototype generator. Nevertheless, CO-OPN classes and ADTs

are very different entities as illustrated in figure 16. Basically, this difference comes from the fact that a CO-OPN class is an encapsulated Petri Net (state based behaviour) with transitions (non-deterministic) working as a predicate, where an ADT is a definition of a set of functions (deterministic). In both modules, the meaning of the axioms are rather different, for ADTs, each axiom is a property that must be satisfied by the operations, while for classes, each axiom defines behaviour that can be followed by the method.

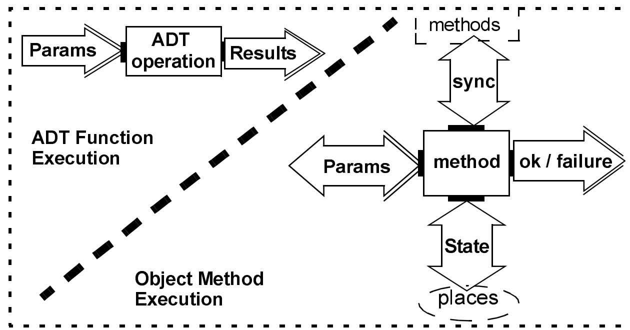

Figure 16: Methods vs. operations

3.5.1 Translation of CO-OPN elements in Java

In table 1 we summarize the relation between the CO-OPN elements and the Java concepts that are used for the translation. Basically, there is one-to-one mapping between CO-OPN and Java classes, one Java class is generated for one CO-OPN class.

Table 1: Mapping between CO-OPN and Java Classes

CO-OPN Class	Java Class/Bean
Type	Class name
Method	Method
Gate	Event
Creation Method	Static factory method
Places	Instance variable “state”
Initial State	Default Constructor
Axiom	Part of corresponding method body
Static Object	Static final variable

3.5.2 States

First of all, CO-OPN objects have an identity and associated state. Object identity is represented by the object’s reference algebra [5]. Object state is represented using the well known concept of Petri Net place. Place elements can be either ADT values or object references. Places are in the private part of a CO-OPN object. The methods of an object deal with places through their pre- and post-conditions.

3.5.3 Methods

A method of a CO-OPN object behaves as one or more transitions of the underlying Petri Net. In contrary to the ADT operations, the execution of a method of a CO-OPN object depends not only on method parameters but also on object state. This dependency is expressed by method pre/post-conditions. A successful method execution results in a change to an object’s state expressed by a method’s post-condition. The execution of a method can also depend on other methods — due to the synchronization mechanism. Synchronizations expand the method-state dependency through the set of CO-OPN objects.

3.5.4 Concurrent synchronization

Like an ADT operation can rely on other operations, a class method can use other methods in order to fulfil requested services. The semantics of those interactions differ significantly. Intuitively, a method synchronizing with another method is like the notion of fusion of transitions. Moreover a CO-OPN class method can synchronize with many other methods. In this case the composed synchronization request is built using synchronization operations. A more detailed description of this concept will be given in sub-section 3.9.

3.5.5 Non-determinism

Finally, a very important aspect of CO-OPN methods is the non-determinism. Remember that ADT operations are deterministic partial functions. CO-OPN methods, like classic PN transitions, allow non-determinism. There are two kinds of non-determinism to consider: non-deterministic choice between matching values in places and non-deterministic choice between fireable transitions. In other words: given a state and a synchronization request there may be many possible ways to fulfil it. There are some programming languages, like Prolog, that support non-determinism, but the majority do not. We use nested transactions [4] for the implementation of both non-determinism and concurrent synchronization.

3.6 Translating an interface of a class

The rules used to translate an interface of a CO-OPN class to Java are similar to those for translating from ADTs.

The name of a Java class is the name of the CO-OPN class type (and not the name of the module). This approach corresponds to the Java notion of type — identical to the class. The full name includes also the package that is identical to the CO-OPN package name. The methods of a Java class are the same as those of its CO-OPN peer.

The figure 17 represents a part of Java class corre-

```
public class cabin{
public void
  closed(CoopnTransaction T)...
public void
  arriveAt(CoopnTransaction T,
           number n)...
}
```

Figure 17: An example of an interface of a generated Java class

sponding to the CO-OPN class `cabin` (see figure 8 for comparison).

In order to interact, CO-OPN class has to provide an interface. In CO-OPN, the only way for objects (and also contexts) to communicate is by synchronization. From the operational point of view, the synchronization mechanism can be viewed as a generalization of the “rendezvous” or transaction mechanism found in other synchronous approaches. The usual way to communicate between objects in Java is the exchange of messages (also called method invocation). Although CO-OPN synchronization and Java method invocations have different semantics, it is a very attractive and natural way to represent the former by the latter.

For a Java client interacting with a CO-OPN class or context, there are two important points to deal with:

- First, a synchronization can result either in success or in failure, depending on system state and synchronization parameters. This can be easily represented by the Java exception mechanism. Successful synchronization corresponds to normal method return, and failed ones to a method execution throwing an exception. A special exception class is defined to represent such a case (`COOPNMethodNotFirableException`).
- Second, CO-OPN synchronizations are transactional. If a part of a synchronization fails then the whole synchronization fails and the system state remains unchanged. For this reason, an additional parameter is required to call a CO-OPN method, namely a transaction descriptor represented by a `CoopnTransaction` object.

Here is an example of a Java client synchronizing with the `closed` method of a CO-OPN object `Cabin`:

```
Cabin cab = new Cabin();
CoopnTransaction T = new
          CoopnTransaction();
try{      cab.closed(T);
          T.commit();
}catch(COOPNMethodNotFirableException e){}
```

In this code fragment, we first create a new `cabin` object and a new transaction descriptor T. A new `cabin` instance is initialized as its CO-OPN specification. Then we invoke the `closed` method with `T` as parameter. If the synchronization succeeds `T.commit()` commits the change of `Cabin` state. Else if the synchronization fails the exception is thrown and the state of `Cabin` remains unchanged.

The example shows some advantages of our approach: a CO-OPN object is created and used in a standard way, i.e. like any “normal” Java object. We prefer to keep the transaction descriptor visible to the client — the possibility to commit or abort the synchronization may be useful if the client is a transactional object itself. Otherwise it is easy to hide transactional aspects from the client by encapsulating them in generated object methods. The try-catch block is optional: `COOPNMethodNotFirableException` is a sub-class of `RuntimeException`.

3.7 Body

A simplified view of a method of CO-OPN class is that a method is similar to ADT operation with some additional parameters, namely values in places and synchronizations with other objects. Method code is obtained by compiling all axioms that define it, in the same way ADT axioms are compiled. A method succeeds if all of its inputs satisfy at least one of its axioms (see figure 18). Unlike in an ADT, more than one class axiom is allowed to match.

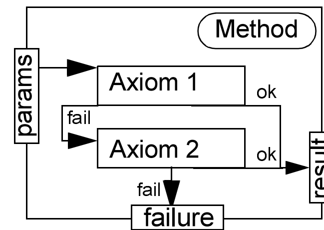

Figure 18: method evaluation scheme

If we look closer inside the axioms, this corresponds to the following ordered schema (figure 19): 1. check axiom parameters, 2. evaluate preconditions, 3. execute synchronization, 4. apply post-conditions, 5. exit the axiom with a “success” status. Some of those steps can fail. If one of those steps fails we leave the axiom with the “failure” status. If there is another axiom in the method definition, then it is tried next, otherwise method execution fails and an exception is thrown

3.8 Object states

Each CO-OPN class defines zero or more places that can contain multisets of ADT values, object references or tuples of them. In order to evaluate an axiom, a method has to search for values in places that match the preconditions and then remove matching values. Evaluating the post-conditions requires one to create new values and put them in

corresponding places. Pre/post-conditions of an axiom can be defined on one or several places, including the possibility to take/put several distinct values from the same place. An implementation of a place has to verify these two conditions:

- A token is taken/put at most once
- A pre/post-condition can take/put several tokens from/to the same place.

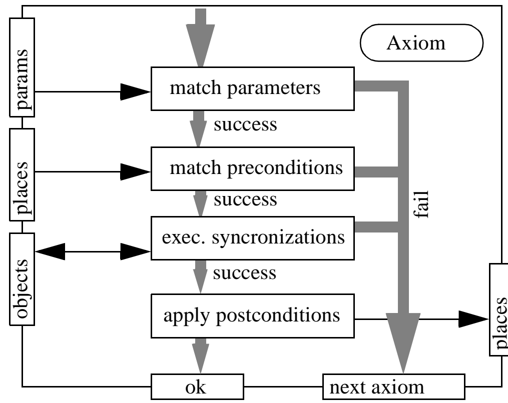

Figure 19: Axiom evaluation scheme

In our implementation the state of one CO-OPN object (all places) is maintained in one instance variable called `state`. Iterators (figure 20) are used to evaluate pre-conditions. Those iterators allow one to obtain all combinations of values that the precondition have to match. Values that are pointed to by an iterator are locked: they become unavailable for other iterators. If the current combination of values does not satisfy the pre-condition, the evaluator asks for the next combination.

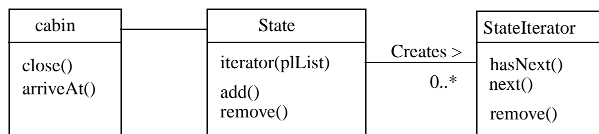

Figure 20: State and StateIterator structure

Then the iterator unlocks the values that no longer belong to the current combination, computes a new combination, locks new values, and returns them. Unlocked values become available for others iterators (at this level resources are allocated to the concurrent events). At the end of a successful synchronization, locked values are removed from places. If the synchronization is cancelled, the corresponding locks are removed and old values restored in places.

3.9 Synchronizations

Synchronizations are the most original part of the CO-OPN formalism and, perhaps, the most challenging aspect that must be managed efficiently and in a clear way.

3.9.1 Three kinds of synchronizations

Synchronizations are used by CO-OPN objects to communicate. An object can accept or refuse a synchronization. If the system has enough resources to satisfy the synchronization request then the synchronization is accepted. If the system does not have sufficient resources, the synchronization is refused. Synchronizations can be simple or they can be combined using three operators: simultaneity, sequence or alternative. A “sim” synchronization succeeds if both its left and its right part can succeed simultaneously (concurrently). A “seq” succeeds if it is possible to synchronize first with its left part then with its right part. Resources that are produced by the left part become available for the right part. “Alt” succeeds if its left or its right part succeeds.

3.9.2 Resource sharing

There are two types of resource sharing with synchronizations (figure 21). Two synchronizations occurring simultaneously have to share the same state (resources). The resources produced by the successful execution of one of many simultaneous synchronizations are not seen by others. If the whole “sim” succeeds the resources produced by its branches are merged together to form a new system state.

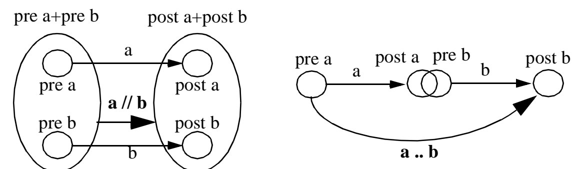

Figure 21: View of the resource sharing for simultaneity and sequence

For sequential synchronizations: If the first synchronization composing a “seq” succeeds, then the second synchronization can use produced resources. Alternative synchronizations are seen as multiple axioms.

3.9.3 Representation and management of synchronizations and states

As explained earlier, CO-OPN synchronizations are represented by method calls. An object that wants to synchronize with a method ’m’ of an object ’o’ simply invokes method ’m’ of object ’o’ with some parameters including the transaction descriptor. In order to find the correct initial state, the callee object has to know the synchronization context of the call. For example: if one object participates more that once in a composed synchronization it has to know which part of the resources can be used for the new initial

state and which part must be shared with previous invocations. This is achieved using `CoopnTransaction` — the transaction identifier that is always present in method parameters.

`CoopnTransaction`, together with `State`, play a very important role in implementation of concurrent synchronizations and non-determinism. In fact, each `CoopnTransaction` object represents a node in a nested transaction tree. `CoopnTransaction` tree structure reflects exactly the structure of a synchronization tree and describes the synchronization context of the call.

At the end of each successful synchronization, the information about the produced and the consumed resources are saved together with an associated `CoopnTransaction`. When a CO-OPN object receives a new synchronization request, it can compute the new state by comparing all previously saved `CoopnTransactions` with the new one.

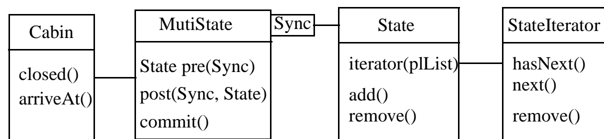

Figure 22: Relation between the class managing object states

The `State` object has to be modified accordingly. In fact, the `State` object must keep separately each intermediary change of the object state together with the associated synchronization context. The objects do not need to know in which synchronization context they will be called later, because they can always recompute initial state for any synchronization. The simple `State` is thus replaced by a multiset of states called `MultiState`. Given a synchronization context represented by a `CoopnTransaction`, the `MultiState` computes the initial `State` for the call. This `State` is then used by a pre/post condition computations. Figure 22 illustrates the structure and relation between the various representations of an object state.

3.9.4 Nested transactions for synchronizations

CO-OPN synchronizations are atomic. As we have seen earlier, the method succeeds only if all of the requested synchronizations succeed. In order to execute a method, we try to execute all of the synchronizations sequentially, one by one. If a method needs to request two synchronizations, there is the situation when the first synchronization succeeds and the second fails. In this case, in order to leave the system state unchanged, we have to cancel the results of the execution of the first synchronization. Also, in order to minimize space requirements, it is useful to notify participants of a successfully executed, complex synchronization that there is no longer any need to conserve multiple intermediary states.

Nested transactions implemented by `CoopnTransaction` together with `MultiState` brings us a solutions to these problems. Each synchronization is executed within a transaction. If a synchronization contains sub-synchronizations they are executed in nested transactions. Aborting a transaction will also abort all its nested sub-transactions, and committing a transaction commits all nested sub-transactions. To implement those commit and abort operations each node of the synchronization tree has an associated transaction manager — the corresponding `MultiState` object. To cancel the results of a synchronization it is necessary to remove the corresponding intermediary sub-state element from `MultiState`. The `abort` method of `CoopnTransaction` serves this purpose. On the other hand, the `commit` method of `CoopnTransaction` informs all participants of a composed synchronization that they have no longer need to conserve intermediary states.

3.10 Non-determinism

Methods of CO-OPN classes may be non-deterministic in data and control dimensions. Data non-determinism occurs when a precondition takes values from places. It is possible that many different values match the precondition requirements. Control non-determinism occurs when more that one method's axiom can apply (like in figure 23).

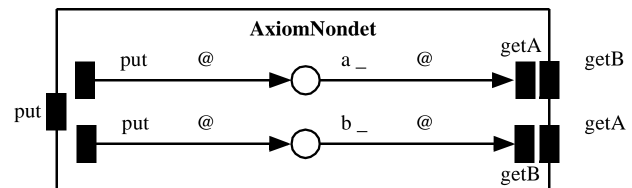

Figure 23: Non-determinism due to axiom multiplicity.

In both cases of non-determinism, it is necessary to choose one of the many matching possibilities. Sometimes, later in the execution of a synchronization, we will figure out that the choice was wrong. In that case, we have to abort any intermediary changes, return to the choice point and look for another possibility. The use of transactions allows us to abort intermediary changes, but do not solve the problem of returning to the choice point. The reason for this is simple: choice points are represented by precise locations in execution paths. Returning to a choice point means restoring the state of program execution.

There is two possible cases. First, the last choice point is in a parent synchronization. In this case it is sufficient to go back in the execution path — for example by throwing an exception. Second, the last choice point is in one of the

previously evaluated "sibling" synchronizations. In this case the execution should continue in a method that was already completed. Such an operation can be implemented using some stack manipulations, but in Java the execution stack is completely hidden from the user. We propose a pure language-based solution.

3.10.1 Non-deterministic Java

We choose to apply to Java the Prolog execution model (enter-exit-fail-redo) of Warren[17]. Suppose that you can extend Java language to have some methods with two "entry points": enter and redo, and two possibilities to leave a method: exit or fail. "Enter" is like the standard java method call. Enter a method means just execute it from the beginning. The method does some work and then successfully exits. So "exit" is like the normal return statement. Otherwise, if the work can not be done, the method fails. The "fail" can be implemented using the Java exception mechanism. The second entry point — "redo" — can be used only when a method was already entered and successfully exited in the same execution context. "Redoing" a method means execute it from the last exit point or, using Java terms, from the last return instruction. That will inform the method that the last choice was refused and it is necessary to look for another possibility to execute its work. As we have full control over the state of our objects, some simple code-rewrite rules are sufficient to implement those four primitives. The `State` class is used to save the local variables and the 'id' of the last exit point for the possible redo(s).

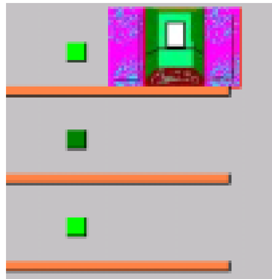

Figure 24: User interface of the Lift.

3.11 Contexts

In CO-OPN contexts are configuration entities that allows the creation of systems by connecting together objects (by the means of synchronizations) and by connecting objects with the external world. Context interfaces are like the object's ones — composed of gates and methods. Contexts have no proper state. They contain named instances and also connections represented like synchronization-only axioms: "required WITH provided". Contexts are static entities — they do not have a type nor an instantiation mechanism. They can just be reused through inheritance. One context is represented in Java by one class. The Java interface of CO-OPN contexts is similar to those of CO-OPN classes. An implementation of a context is similar to a CO-OPN class implementation, however the data non-determinism does not occur in contexts.

4. Integration of generated code into an application

In order to facilitate the integration of generated code into a larger software system, the component architecture of CO-OPN (presented in section 2.3) is translated to a Java component architecture, namely JavaBeans [13].

The CO-OPN component interaction is based on two complementary mechanisms: methods and gates.

A gate is, in some sense, the opposite of a method: a method represents a provided service, where a gate represents a required service. The role of a context is to resolve required services by provided ones that are connected by synchronization expressions. At the top-level englobing context, methods and gates represent services provided and required by the system. We choose to represent CO-OPN gates (of both classes and contexts) by JavaBean events. A synchronization with a gate (or service request) is represented by event firing. Connection axioms in a context, which route service requests to service providers, are represented by event handlers. Those handlers implement the `EventListener` interface of corresponding gate and execute synchronizations with connected methods. Mapping CO-OPN methods to Java methods and CO-OPN gates to Java events is very useful in practice: this allows one to easily integrate CO-OPN context translation in a software system using existing tools. One step further would be to map CO-OPN components into the more complex EJB (Enterprise Java Beans)[12] components.

4.1 Example of integration: Lift applet

The Lift Applet is composed of three layers: Interface, Interconnection and Command. The Interface layer visualizes an interactive and animated user interface (figure 24). The Command layer is represented by a `ListSystem` JavaBean generated from corresponding context. The Interconnection layer links together user interface and command layer.

The Interface is composed of a hierarchical tree of graphical objects (images). The user can put in motion some parts of the interface by assigning them a speed. Detectors are used to find object positions (an event is emitted for selected positions). Selecting an interface object also emits an event.

The code generator produces a standalone collection of JavaBeans representing specification modules and a few

support classes (like `CoopnTransaction` or `Coopn-MethodNotFirableException`).

The final step is the construction of the Interconnection layer. It consists of connecting Interface events to methods of the Command layer and vice versa. Because the control object, corresponding to CO-OPN context, was generated with respect to the standard JavaBean conventions, it can be easily integrated into a software system using standard market tools, like *JBuilder*.

5. Future work and conclusions

In this paper, we presented a code generation technique for CO-OPN specifications (i.e. for coordinated algebraic Petri nets), which is actually an extended synthesis (for the translation of the gates into the events) of the partial techniques we used until now [6]. These techniques are based on an implementation of a transaction mechanism and the implementation of non-determinism. As a direct application, we provide JavaBeans standard components that can be easily integrated into any kind of applications.

We applied our techniques on a medium-sized example. The lift system example was developed from a UML model and we found, by animation of the prototype, several significant errors. Our web site (http://lglwww.epfl.ch) contains the complete source code of the presented examples as well as the supporting tools.

In the future, we would like to improve our work into two directions: the first is to be able to guarantee the correctness of the translation from CO-OPN to programming languages. The idea is to use the natural decomposition of our implementation into transactional support, non-determinism and object oriented structure in order to factorize the correctness proof. The second direction is to improve the translation in order to cover more cases; incremental prototyping (i.e. the replacement by hand-written code, now studied only for ADT); distributed applications and mobility. In addition, we will also develop an extended version of the translation in which not only local competitive concurrency (with respect to resource acquisition) is supported but also concurrency between distributed entities.

Acknowledgments

We wish to thank Shane Sendall for his careful reading and remarks on this paper. We also thank the anonymous referees for their pertinent comments about the readability of the paper.

References

[1] R. Ben-Natan, "CORBA: a Guide to Common Object Request Broker Architecture", MacGraw-Hill, 1995.

[2] Didier Buchs and Nicolas Guelfi, ‘‘A Formal Specification Framework for Object-Oriented Distributed Systems,” IEEE TSE, vol. 26, no. 7, July 2000, pp. 635-652.

[3] G. Weikum, "Principles and Realization Strategies of Multilevel Transaction Management", ACM Transactions of Database Systems, Vol 16, No 1, pp 132-180, 1991.

[4] J.E.B. Moss, “Nested Actions: an Approach to Reliable Distributed Computing”, PhD thesis, Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[5] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. In G. Agha and F. De Cindioand G.Rozenberg, editors, *Advances in Petri Nets on Object-Orientation*, LNCS. Springer-Verlag, 2001.

[6] Didier Buchs and Mathieu Buffo. Rapid prototyping of formally modelled distributed systems. In Frances M. Titsworth, editor, *Proc. of the Tenth International Workshop on Rapid System Prototyping RSP’99*. IEEE, june 1999.

[7] Mathieu Buffo. Experiences in coordination programming. In *Proc. of the workshops of DEXA’98 (Int. Conf. on Database and Expert Systems Applications)*. IEEE , aug 1998.

[8] 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.

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

[10] Ph. Schnoebelen. Refined compilation of pattern-matching for functionnal languages. *Science of Computer Programming*, pages 11:133-159, 1988.

[11] Olivier Biberstein and Didier Buchs. Structured algebraic nets with object-orientation. In *Proc.of the first int. workshop on “Object-Oriented Programming and Models of Concurrency” within the 16th Int. Conf. on Application and Theory of Petri Nets*, Torino, Italy, June 26-30 1995.

[12] Sun Microsystems: Enterprise Java Beans Spec, V2.0 (October 2000).

[13] Sun Microsystems: JavaBeans spec. v1.01 (July, 1997).

[14] Mathieu Buffo and Didier Buchs. A coordination model for distributed object systems. In *Proc. of the Second Int. Conf. on Coordination Models and Languages COORDINATION’97*, vol. 1282 of *LNCS*, pp. 410–413. Springer , 1997.

[15] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs*. MIT Press, Cambridge and London, 1990.

[16] Jeff Kramer, Jeff Magee, Morris Sloman, and Naranker Dulay. Configuring object-based distributed programs in rex. *IEEE Software Engineering Journal*, 7(2):139–149, 1992.

[17] D.H.D.Warren, L.M.Pereira, F.C.N.Pereira. Prolog—The Language and its Implementation Compared with LISP. In *ACM SIGPLAN Notices*, Vol. 12, No. 8, pp 109-115, 1977.