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

Didier Buchs
Software Engineering Laboratory
Swiss Federal Institute of Technology
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 that arise when implementing synchronization and non-determinism of CO-OPN classes in procedural languages. Our method proposes a solution to these problems. Another interesting point of our method is the easy integration of a generated prototype into any existing system. This paper focuses on the generation of Java code fulfilling the Java Beans component architecture, but the issues discussed here are applicable more generally to other object-oriented implementation languages with a component architecture.

1. Introduction

Developing complex distributed reactive software systems needs 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 which 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 distributed architecture.

In order to produce an implementation from the abstract CO-OPN model we will describe the automated techniques that we can use to produce programming language code. Apart from the problem of producing programs that respect the abstract 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. For our purpose it will be 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). This will be achieved by introducing transparently the support for transactions and non-determinism, and by fulfilling the rules of a 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 concrete 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. The lift problem entities, composing the whole system (figure 1) can 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. In addition it is assumed that there is an external system imposing the doors to close when they are open for a certain duration.

![Figure 1: The actors (real life view)](image)

For such a system we are interested in a first approach to determine the components and the various interactions between them, 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.

![Figure 2: The actors and their interaction (controller view)](image)

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 kind of top-down strategy for modelling, we will 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 appeared that coordination patterns are likely to be applied since 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.

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](image)

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 oriented 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](image)

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 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.

![Figure 5: The Direction ADT](image)

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 al-
gebraic Petri nets with particular parameterised 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:

\[ \text{Cond} \implies \text{eventname}\] With \text{synchro} : \text{pre} \rightarrow \text{post} \]

In which the terms have the following meaning:
- \text{Cond} is a set of equational conditions, similar to a guard;
- \text{eventname} is the name of the methods with the algebraic term parameters;
- \text{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.
- \text{pre} and \text{post} are the usual Petri Net flow relation determining what is consumed and what it 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.

![Figure 7: The Floors class graphical description](image)

We will explain the behaviour of the cabin when the event \text{closed} (door closed) is received. The direction of the movement of the cabin in the lift system must be defined accordingly to the floors that have been requested. In figure 8 two cases are formalised, the first one when previous events do not need to plan a movement (place \text{move} with value \text{STOP}) and the second one when a movement is planned and transmitted to the motor of the lift (modify d).

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 it is also easy to understand/use by a human programmer. Because produced Java classes satisfy the requirements of a Java component architecture, namely JavaBeans, they are easy to use by a development
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 for easily changing the implementation in a modular way as justified in [9] and initially proposed in [8]. Because of a leak of space, this aspect will not be discussed further.

### 3.2 Code generation for ADT

The lift specification uses few ADTs. One of them is List (Figure 9). Values of this data type are used by the Building class to store a list of floors.

In CO-OPN types and functions are dissociated. An ADT module can declare few types and few 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.

![Figure 9: List ADT (partial)](image)

### 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 discriminate the generator used to build a term, while inverse methods allow to retrieve the parameters of a term generator (i.e. sub-terms). For optimisation 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 identifiers. 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.

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

**Figure 11 : Interface list**

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.

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

**Figure 12: Syntactic Representation of lists**

Then, we define one class for each generator. Values are represented as syntactic trees, where nodes are instances of those “generator” classes.

In the case of the Lists, we produce two classes listSyntacticisEmpty and listSyntacticIsAdd. Figure 13 details the class recording non-empty lists. The reader can observe how the test and inverse methods are implemented.

```java
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 class of the ADT module List.

```java
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**

A default implementation of this class is produced. This implementation is based on the syntactic representation of values and on the axioms of the ADT. For each operation in ADT, we compute a set of rewrite rules, obtained from the axioms by orienting them [10]. The core of the associated method is mainly a multiple choice between a set of branches, each of them corresponding to an applicable axiom. Undefined cases throw exceptions.

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 none of axioms apply, a COOPNNoSemanticsException is thrown, signaling that the operation is undefined on given parameters (never occurs for the size method). The reader can observe how the test methods are used.

### 3.5 Specific class principles

In this section we will describe CO-OPN classes. Our primary goal is to show specific class principles. The implementation of those principles will be presented in following sub-sections. For the sake of clarity we will compare classes with ADTs. Obviously, some of concepts appearing in classes do not exist in the ADT world.
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 and an ADT is a definition of a set of functions (deterministic). In both modules, the meaning of the axioms are rather different, in the ADTs each axiom is a property that must be satisfied by the operations while in Classes, each axiom is a behaviour that can be followed by the method.

3.5.1 States

First of all, CO-OPN objects (class instances) 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: named multiset of values. 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.

3.5.2 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. Successful method execution results in change in object’s state expressed by method post-condi-

3.5.3 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.

3.5.4 Non-determinism

Finally, a very important aspect of CO-OPN methods is the non-determinism. Remember that ADT operations are deterministic partial functions always mapping the same parameters to the same result. 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.

Another important feature of CO-OPN class are gates. When methods represent services provided by a class, gates abstract the services required. The gates will be further described in section 4.
3.5.5 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

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

3.6 Translating a class interface

We will start by describing the interface part of java classes generated from CO-OPN classes, then we will continue by describing their implementation. We will call java classes or java-CO-OPN classes, the classes that are generated from CO-OPN specification. Some concepts of the CO-OPN class to Java translation are common to that of the translation from a CO-OPN Context to Java, they will be mentioned in this section.

The rules used to create a java-CO-OPN class interface are similar to those for translating from ADTs.

The name of a java-CO-OPN class is a 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-CO-OPN class are the same as those of its CO-OPN peer.

In order to interact, a CO-OPN system 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.

On the other hand the usual way to communicate between objects in Java is the exchange of messages (aka. method invocation). Although CO-OPN synchronization and Java method invocations have different semantics, it is very attractive and natural to represent the first by the second.

For a Java client interacting with a CO-OPN system (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 Java exception mechanism. Successful synchronization corresponds to normal method return and failed ones to method execution throwing an exception. A special exception class is defined to represent such a case, namely COOPNMethodNotFirableException.
- Second, CO-OPN synchronizations are transactional. If a part of a synchronization fails then the whole synchronization fails and 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:

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

In this code, we first create a new Cabin object and a new transaction descriptor T. A new Cabin instance is initialized with an initial state as described by its CO-OPN specification.

Then we invoke the closed method with T as parameter. If the synchronization succeeds T.commit() validates the change of Cabin state. Else if the synchronization fails the exception is thrown and Cabin state remains unchanged. More details on CoopnTransaction will be given in next sub-sections.

The example shows the obvious 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 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.

3.7 Body

Now we will look inside the generated Java class. In order to facilitate the understanding of the translation process we start with a simplified view of this process, and after introducing it we will progressively complicate it until we will reach the actual implementation.

A very simple view of a generated method is that a method is similar to ADT operation (deterministic) with some additional parameters, namely values in places and synchronization with other objects. So, 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 (figure 18 shows a Warren abstract machine like) (condition, pre-condition and synchronization). Unlike in an ADT, more than one class axiom is allowed to match.

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, one has to search for values in places that match the preconditions and then remove matching values. Evaluating the Post-conditions requires one to creates new values and put them in corresponding places. Pre/post-conditions 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/postcondition can take/put several tokens from/to the same place.

3.8 Object states

The next step is to define how to implement object state

- places.

In our implementation the state of a CO-OPN object (all places) is maintained in one instance variable called state.

State provides iterators on values in a sub-multiset of places (figure 20). Those iterators can be used to obtain all combinations of values that the precondition have to match. Values currently pointed to by an iterator are locked: they become unavailable for other iterators. If the values do not satisfy the pre-condition axiom, the evaluation asks for the next combination..

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 concurrent iterators (at this level...
resources are allocated to the concurrent events). At the end of a successful synchronization, locked values are removed from places. Otherwise, if the synchronization is cancelled, the corresponding locks are removed.

3.9 Synchronizations

The other important part of class axioms are synchronizations. This is 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 type of resource sharing with synchronizations

\[
\begin{align*}
\text{pre a} & \rightarrow \text{post a} \\
\text{pre b} & \rightarrow \text{post b}
\end{align*}
\]

\[
\begin{align*}
\text{pre a} & \rightarrow \text{post a} \\
\text{pre b} & \rightarrow \text{post b}
\end{align*}
\]

\[
\begin{align*}
\text{a \lor b} & \rightarrow \text{a \land b}
\end{align*}
\]

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

Two synchronizations occurring simultaneously have to share the same initial 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. The state after a successful "sim" can be written as usual in Petri nets:

\[
s_1 = s_0 - \text{pre(left)} + \text{post(left)} + \text{post(right)}
\]

where \(s_0\) denotes the shared initial state, \(s_1\) is the resulting state, \(\text{left}\) and \(\text{right}\) denotes two synchronizations that compose a sim, \(\text{pre(s)}\) are resources consumed and \(\text{post(s)}\) resources are produced by a synchronization.

For two sequential synchronizations the picture is different. If the first synchronization composing a sim (left) starts on an initial state \(s_0\) and succeeds then the initial state for the second synchronization (right) is given by \(s_0 - \text{pre(left)} + \text{post(left)}\). The state after a successful sequential synchronization is given by the following formula:

\[
s_1 = (s_0 - \text{pre(left)}) + (\text{pre(right)} + \text{post(right)})
\]

We can remark that (1) and (2) denote the same state. This property is used to simplify the implementation of the composed synchronizations.

Alternative synchronizations are seen in the current implementation as multiple axioms.

3.9.3 Representation and management of synchronizations and states

As explained before 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 have 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 produced resources can be used for new initial state and which part must be shared with previous invocations. This is achieved using CoopnTransaction - the transaction identifier always present in method parameters.

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

At the end of each successful synchronization the associated CoopnTransaction object is saved together with information about the produced and the consumed resources. When a CO-OPN object receives a new synchronization request it can easily determine the new initial state by comparing previously saved CoopnTransactions with a new one.

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 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 syn-
chronization. The simple State is thus replaced by a multiset of states called MultiState. Given a synchronization context represented by a CoopnTransaction, the MultiState class computes the initial State for the call. This State is then used by pre/post condition computations. The figure 22 illustrate the structure and relation between the various representations of an object state.

![Figure 22: Relation between the class managing object states](image)

### 3.9.4 Nested transaction for synchronizations

CO-OPN synchronizations are atomic. As we have seen before the method succeeds if all of requested synchronizations succeeds. 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 a possible 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 execution of first synchronization.

Also, in order to minimize space requirements, it is useful to notify participants of a successful composed 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 other synchronizations they are executed as nested transactions. Aborting a transaction will also abort all its nested subtransactions, and committing a transaction commits all nested subtransactions. To implement those commit and abort operations each node of the synchronization tree has an associated transaction manager - the corresponding MultiState object. To cancel (or undo) 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. The new unique state is computed using the formula (1) of subsection 3.9.

### 3.10 Non-determinism

The non-determinism is a very powerful feature of CO-OPN. There is another use of nested transactions in a CO-OPN class implementation: the management of non-determinism such as that of figure 23.

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. The choice from matching values is non-deterministic.

Control non-determinism occurs when more that one method’s axiom can apply in the given system state.

![Figure 23: Non-determinism due to axiom multiplicity.](image)

In both cases it is necessary to choose one of 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: as choice points are in fact represented by precise locations in code execution path, returning to a choice point means restoring the state of program execution. Such operations imply some stack manipulations and, possibly, can be implemented in some programming languages, but not in Java, where the execution stack is completely hidden from the user. We propose a pure language-based solution.

There is two possible cases. First, the last choice point can be in a parent synchronization then it is sufficient to go back in the execution path - for example by throwing an exception. Second, the last choice point can be in one of previously evaluated "sibling" synchronizations.

### 3.10.1 Non-deterministic Java

To solve this problem we choose to apply to java the prolog execution model (enter-exit-fail-redo) of Warren. Suppose that you can extend java language to have some methods with two "entry points": enter and redo. The first is
the standard java method call. Enter a method means just execute it from the beginning. The second - redo - can be applied only to a method that was already entered and successfully exited in this execution context. "Redo" that method means execute it from the last exit point or in java terms - from the last return instruction. It is possible to organize method code that can restart after the last return statement. This will bring the execution path to the next choice point or to fail, if no more choice points exists. Finally, like in prolog, there must be two possibilities to end the method evaluation: exit or fail.

The other possibility to understand this technique is to take the method's point of view. Let us say that there is no "return" instruction, but instead an "exit" and a "fail" instructions. The method has some work to do. Sometimes it has different possibilities to achieve this work. The method chooses one of the possibilities, completes the related work and then proposes the results to the caller by calling the "exit", then waits. The caller can either accept or reject those results. If the results are rejected "exit" returns and the method continues.

The last instruction in this method is implicitly a fail, which notifies the caller that there is no more possibilities to accomplish the request.

We use some simple code-rewrite rules to add "exit" and "fail" primitives to the java programming language. There is also some additions to State class to save the values of the method's local variables and the 'id' of the last exit point for the possible redo.

To summarize: non-determinism is implemented in prolog-like fashion, nested transactions are used to undo changes of system state and code rewriting to implement the prolog "redo" primitive.

3.11 Code instrumentation: execution trace

Execution of specifications is a powerful approach for validation of models. The mapping of specification elements to software components simplifies the integration of of executable specifications into software systems and facilitates the process of validation.

The main purpose of validation is to insure that the specification behaves exactly as it is supposed to. Using standard debugging facilities on generated code is not of great help. In fact, standard debuggers will make no difference between useful information (as synchronization requests, axiom evaluation and variable bindings) and non-functional aspects (as management of non-determinism or nested transactions).

In order to provide a convenient debugging facility we instrument the generated code with a trace feature. The execution of synchronization requests generates a trace. Such a trace consists of information about synchronizations that succeeded and failed, applied axioms etc. Figure 24 shows an example of a trace, following the CO-OPN inference rules [5]:

```
ENTER cabin.pressed(3)
ENTER building.setRequest(3)
ENTER building.iterateFloorRequest((flr1'(flr2'(flr3'[])))3)
  ENTER flr1.stopWillBeRequested(3)
  FAIL flr1.stopWillBeRequested(3)
  ENTER flr1.stopNotWillBeRequested(3)
  EXIT (axiom #10)flr1.(stopNotWillBeRequested 3)
  ENTER building.iterateFloorRequest((flr2'(flr3'[]))3)
  ENTER flr2.stopWillBeRequested(3)
  FAIL flr2.stopWillBeRequested(3)
  ENTER flr2.stopNotWillBeRequested(3)
  EXIT (axiom #10)flr2.(stopNotWillBeRequested 3)
  ENTER building.iterateFloorRequest((flr3'[]))3)
  ENTER flr3.stopWillBeRequested(3)
  EXIT (axiom #9)flr3.stopWillBeRequested(3)
  EXIT (body axiom AxR)building.setRequest(3)
  EXIT (case2)cabin.pressed(3)
```

Figure 24: Example of a trace

3.12 Contexts

In CO-OPN contexts are configuration entities that allows the creation of systems by connecting together objects (by the mean 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 the inheritance mechanism

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 this chapter we will describe how to integrate generated Java code into a software system. The CO-OPN component architecture was already presented in section 2.3. Here we will present how the CO-OPN component architecture is mapped to a Java component architecture, namely JavaBeans [13]. Finally an example will be given, presenting an integration of code generated for Lift specification into a graphical user interface.
The CO-OPN component architecture is based on two complementary mechanisms: methods and gates. We already presented CO-OPN methods and the code generated for them. Now we will present CO-OPN gates.

A gate is, in some sense, the opposite of a method: if the methods can be seen as the "entry points" of an object, gates can be seen as the "exit points". When a method represents a provided service, a gate represents a required service. A context links together methods and gates of its members. Nevertheless, methods and gates of a top-level context represents services provided and required by a 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. In the JavaBeans event model, each event has an associated EventListener interface. Connection context axioms, that route service requests to service providers, are represented by event handlers implementing the EventListener interface of corresponding gate and synchronizing with its corresponding methods. Mapping CO-OPN methods to Java methods and CO-OPN gates to Java events is very useful in practice. Indeed we represent CO-OPN components by Java software components, that can be easily integrated in a software system using existing tools.

Now we are looking how to map CO-OPN components into more complex component architecture, namely EJB (Enterprise Java Beans).[12]

4.1 Example of integration: Lift applet

The Lift Applet is composed of three layers: Interface, Interconnection and Command layers. The Interface layer visualizes an interactive and animated user interface (figure 25). 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 hierarchical tree of graphical objects. Each of these objects have a color or an associated image. The User can put in motion some parts of interface by assigning them a speed. Detectors are used to track moving object position: where a moving object encounters a detector, an event is emitted. Finally, the event is emitted when user mouse-click on an interface object.

To produce the Command layer we automatically generate Java code from CO-OPN specification of Lift System. Java code generator produce a standalone collection of JavaBeans representing specification modules and a few support classes (like CoopnTransaction or CoopnMethodNotFirableException).

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. Tables 2 and 3 presents some of those connections:

<table>
<thead>
<tr>
<th>Interface event</th>
<th>Synchronization request</th>
</tr>
</thead>
<tbody>
<tr>
<td>button1.pressed</td>
<td>pressed 1</td>
</tr>
<tr>
<td>doorDetector.activated</td>
<td>closed</td>
</tr>
<tr>
<td>Floor1Detector.activated</td>
<td>arriveAt 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Synchronization request</th>
<th>Interface method invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>door.setSpeed(-1)</td>
</tr>
<tr>
<td>open</td>
<td>door.setSpeed(1)</td>
</tr>
<tr>
<td>modify STOP</td>
<td>cabin.setSpeed(0)</td>
</tr>
</tbody>
</table>

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 present a code generation technique for CO-OPN specifications (i.e. for coordinated algebraic Petri nets), which is actually a extended synthesis of the partial techniques we used up to now [6]. These techniques are based on implementation of transactional techniques and implementation of non-determinism, and respect the layers of CO-OPN.

As a direct application of this technique, we provide Java standard components that can be integrated into any kind of application using the JavaBeans components.

We made an experience of our techniques on a medium size example and this was very useful to validate models. For instance, the lift system example was developed from a UML model and we found, by animation of the prototype, several significant errors that were not previously discovered were found. Our web site (http://fluwww.epfl.ch) con-
tains 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 language. 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. We will have:

- a correctness proof at the module level that is based on the correctness of the axiom implementation and the correctness of the non-deterministic behaviour
- correctness proof of the composition mechanism based on the correctness of the transactional support.

The second direction is to improve the translation: to have a better transformation of axioms in rewrite rules, in order to cover more cases: better compilation of rewrite rules, efficiency; "out" and "in/out" kinds of parameters in methods (actually we have only "in" parameters); internal transactions; subtypes; external world connections (asynchronous); 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, this will imply to consider distributed algorithms.

References

