UNIVERSITÉ DE GENÈVE

Archive ouverte UNIGE
https://archive-ouverte.unige.ch

Article scientifique | Article | 2003

Published version | Open Access

This is the published version of the publication, made available in accordance with the publisher's policy.

Interfacing Software Libraries from Nondeterministic Prototypes

Chachkov, Stanislav; Buchs, Didier

How to cite

CHACHKOV, Stanislav, BUCHS, Didier. Interfacing Software Libraries from Nondeterministic Prototypes. In: Design Automation for Embedded Systems, 2003, vol. 8, n° 4, p. 327–343. doi: 10.1023/B:DAEM.0000013066.31883.73

This publication URL: https://archive-ouverte.unige.ch//unige:120883
Publication DOI: 10.1023/B:DAEM.0000013066.31883.73

© The author(s). This work is licensed under a Other Open Access license
https://www.unige.ch/biblio/aou/fr/guide/info/references/licences/


Design Automation for Embedded Systems, 8, 327–343, 2003.
© 2004 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Interfacing Software Libraries from Nondeterministic Prototypes

STANISLAV CHACHKOV stanislav.chachkov@epfl.ch
Software Engineering Laboratory, Swiss Federal Institute of Technology, 1015 Lausanne, Switzerland

DIDIER BUCHS didier.buchs@cui.unige.ch
Computer Science Department, University of Geneva, 1211 Geneva, Switzerland

Abstract. Modeling software systems is one of the most obvious uses of a formal specification language. A software prototype, automatically generated from the specification, enables the developer to validate the system in a real environment. However, real software systems are seldom developed from scratch, but rather built using existing libraries. In this paper, we show how a program based on existing software libraries is modeled in the Concurrent Object-Oriented Petri Net specification language, and how a prototype code is generated from the specification. In particular, we study the interface between non-determinist synchronous prototypes and determinist asynchronous software libraries. We take into account, problems related with nonreversibility of actions and the transactional semantics of the specification language that is kept in the prototypes. A convenient delayed execution mechanism is proposed for the library linking part of the generated code.

Keywords: prototyping, non-determinism, Petri nets, external laboratory

1. Introduction

Previously, we have defined a method for the generation of executable code from our formal specification language Concurrent Object-Oriented Petri Net (CO-OPN) [10], [11]. The interface of the generated code was simple, modular (each component of the specification language is a component of the target language), and sufficient for systems encapsulated into a single object. In this paper, we shall present an extension of our approach that enables a more subtle and fine-grain object-oriented form of interface, and allows the integration of existing components (for instance, from libraries).

To achieve this goal, we shall insert models of the objects of these libraries into the formal model of the system, and then generate the code for the whole new system. The use of proxies to connect generated code with objects from libraries will ensure the translation of the different execution paradigms. The execution model of the generated code is close to the resolution [6] and has a transactional behavior, but this is not the case for library components that support neither backtracking nor transactions. Instead, we use information from the model in order to build a plan (sequence of future method calls) that is guaranteed to work.

Our formal approach, called CO-OPNS, includes a coordination layer that has been developed to deal with distributed architecture.

CO-OPN is an object-oriented modeling language, based on Algebraic Data Types (ADT), Petri Nets, and Idealized Workers Idealized Manager (IWIM) coordination

models [12]. Hence, CO-OPN concrete specifications are collections of ADT, class and context (i.e., coordination) *modules* [10]. Syntactically, each module has the same overall structure; it includes an *interface section* defining all elements accessible from the outside, and a *body section* including the local aspects, private to the module. Moreover, class and context modules have convenient graphical representations, showing their underlying Petri Net model. Low-level mechanisms and other features specifically dealing with object-orientation, such as subclassing and subtyping, are out of the scope of this paper, and can be found in [2].

Our specification language allows to fully exploit features of object-oriented libraries, such as dynamic aspects: object creation, destruction, and reference handling.

In Section 2 we introduce the example of a software system—software video recorder—built on top of existing libraries. In Section 3, we briefly describe our modeling language. In Section 4, we present the model of the system. In Section 5, the concept of prototype is introduced and then the object-oriented interface of generated code is presented. Section 6 illustrates an application of our technique to the Video Cassette Recorder (VCR) example. The last two sections contain some related work and the conclusion.

2. Example: The VCR Program

We would like to model a software system that can play and record video. The main component of our system is a VCR. The other components are cameras, cassettes, and displays. In fact, they are not real equipment, but instead software abstractions that incorporates data feeds, files, and windows.

We also choose not to build our system from scratch, but instead to use an existing software library, namely Java Media Framework (JMF) [1]. This library provides a way to display or record video files and feeds from video sources by configuring video data flow controllers.

The components of JMF that we use are Processor, Data Source, Data Sink, Media Locator, and Content Descriptor. We also use components from the Java GUI libraries in order to display video on the screen.

A Media Locator object is a kind of Uniform Resources Location (URL) that identifies the video data location. This may, for example, be a file or a live feed. Data Source and Data Sink are constructed using a Media Locator. Data Source reads the data from a location indicated by Media Locator. Data Sink writes data.

The heart of JMF is the Processor object. A Processor is always connected to a Data Source. Processor transforms the data it receives from its Data Source either for displaying or for recording video (Figure 1).

To record data we specify, using a Content Descriptor, the format in which the Processor will encode its output. Later, a Processor configured with a Content Descriptor will provide its output in the form of a new Data Source object. The Data Sink is then created using the output of the Processor and a Media Descriptor.

Alternatively, we may omit specifying a Content Descriptor, and later ask the Processor to give us the Renderer (a Java GUI component in which the video will be shown).

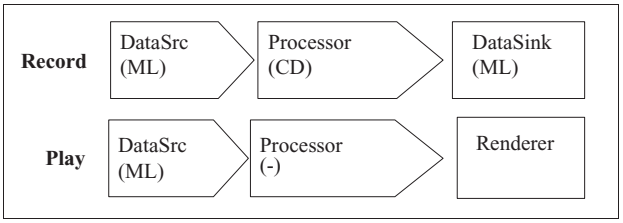

Figure 1. Two configurations of processor. ML: media locator, CD: content descriptor.

Processor has three successive states. It is created in the `Configured` state. In this state, the processor is already connected to a Data Source and has recognized the format of input data; the Processor may be provided with a Content Descriptor object. It is then moved to the `Realized` state, in which an output Data Source, or Renderer, becomes available. Finally, it can be started and begin to process the video feed. This behavior will be modeled in the CO-OPN formalism, the data flow being omitted (the actual flow of video data); this is a useful abstraction, which keeps all the interesting properties of the libraries for validation and verification of the global system.

For our VCR program, we will use Media Descriptor to implement the camera and the cassette, and a GUI Window to implement the monitor. A VCR object will accept inputs/outputs in the form of cassettes, cameras, monitors, and control events such as play, record, and stop.

3. CO-OPN Specification Language

The CO-OPN specification language features three kinds of modeling entities: abstract algebraic data types (ADT), classes, and contexts. Contexts are coordination entities, they coordinate activities among classes; the example of this paper does not use them. ADT and classes are presented below.

3.1. 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.

One simple example of ADT from our specification is `MediaLocator` (Figure 2). In order to simplify the specification, the exact identification of the media location is not modeled, but only the type of the resource. The ADT module declares one *sort* `mediaLocator` and two *generators* `renderer` and `storage`. This ADT defines no operations.

```
ADT MediaLocator;
Interface
  Sort mediaLocator;
  Generators
    renderer : -> mediaLocator;
    storage : -> mediaLocator;
End MediaLocator;
```

Figure 2. ADT MediaLocator.

```
ADT Cassette;
Interface
  Use       MediaLocator;
  Sort      cassette;
  Generator      cassette _ : mediaLocator -> cassette;
End Cassette;
```

Figure 3. ADT Cassette.

```
ADT Booleans;
Interface
  Sort      boolean;
  Generator      true,false : -> boolean;
 Operation
  _ and _ :  boolean, boolean -> boolean;
...
Axioms
  X and true = X;
  X and false = false;
...
Where
 X: boolean; End Booleans
```

Figure 4. ADT Booleans.

The Cassette ADT is another example (Figure 3). Thanks to its generator, a value of this sort encapsulates a MediaLocator.

A slightly more sophisticated example of ADT is given by the CO-OPN standard library module `Booleans` (Figure 4).

Any kind of data structure could be modeled in this language with a high level of abstraction and a clear declarative style. Even if we have a declarative approach, most of the usual data structures can be animated using rewriting techniques (essentially, rewrite systems are obtained by orienting the equations). This is the main principle used for the code generation supported by our tools.

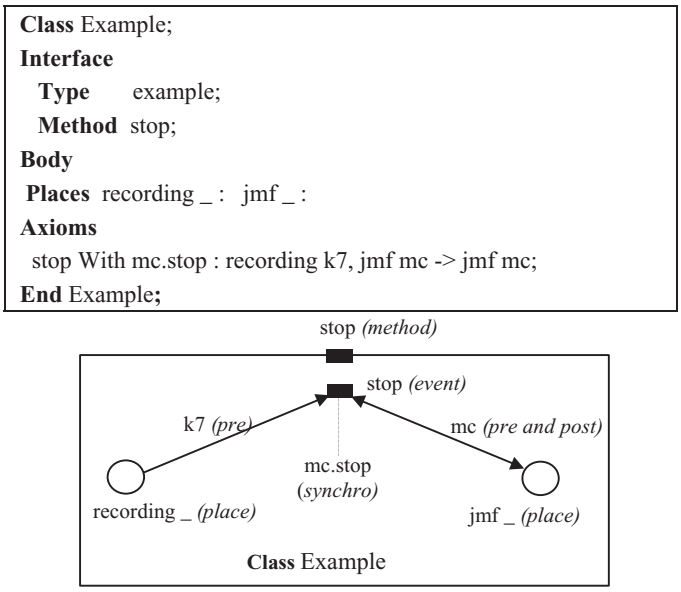

Figure 5. Textual and graphical representation of an Example Class.

3.2. Classes

CO-OPN classes are described by means of modular algebraic Petri Nets with particular parameterized external transitions that are called methods of the class. As usual, instances of classes are called objects. Objects are instantiated by using one of user-defined *creation* methods, or a predefined method *Create*.

The behavior of transitions is defined by so-called behavioral axioms, similar to the axioms in an ADT. A method call results in an external transitions synchronization, similar to the fusion of transitions technique. The axioms have the following syntax (see Figure 5, for an example of corresponding graphical representation of axiom):

$$Cond \Rightarrow \ eventname \ \textbf{With} \ synchro\text{: } pre \rightarrow post$$

In which the terms have the following meaning:

- *Cond* is a set of equational conditions—the guard.
- *eventname* is the name of a method with the algebraic term parameters.
- *synchro* is the synchronization expression defining the policy of transactional interaction of this event with other events. Synchronization expressions are the CO-OPN equivalent of method calls. The dot notation is used to access events of specific objects. Synchronization expressions can be combined using the synchronization operators: the sequence, the simultaneity, and, the nondeterminism.

- *Pre* and *Post* are the usual Petri Net flow relations determining what is consumed and what is produced in the object state *places*.

The state of a CO-OPN object is stored in *places*, which are really named multisets of algebraic terms, object references, or tuples of them.

CO-OPN provides tools for the management of graphical and textual representations. Figure 5 shows the partial class. description net corresponding to a simple class Example in textual form; the equivalent graphical description (the Petri Net, plus additional information concerning the interface) is also depicted in Figure 5.

4. CO-OPN Model of VCR Program

When the overall architecture of the system is well understood, one question remains open: which elements do we have to include in the model? There are two options: model only the controller or model both the controller and the library.

In order to be able to perform real validations of the system, we choose the second option, but with some limitation for the libraries. We model, using ADT and Classes, the desired behavior of the VCR program and the simplified behavior of a subset of JMF, presented later.

4.1. Controller Model

Parameterized inputs of the VCR class are listed in Figure 6 and the CO-OPN model of the behavior of the VCR is shown in Figure 7.

`Cassette` and `Camera` are two ADTs which encapsulate a MediaLocator. `Monitor` is a Class with an interface composed of two methods: show _: `component` and `hide`. In fact `Monitor` models a window with a GUI component inside.

VCR is created (*creation method* `init`) in the "empty" state (represented by an anonymous token @ in *place* `empty_`). `Init` also creates a new `MediaCenter` object by synchronizing with its default creation method `Create`. `MediaCenter` is stored in the *place* `jmf_`. VCR provides methods to connect/disconnect `Cassette`, `camera`, and `monitor` and also places to store them. Methods `play` and `record` move `VCR` from an inactive to an active state. The `play` method, for example, takes `monitor`, `cassette`, and a `MediaCenter` as preconditions from the corresponding places. It next initializes and starts the `MediaCenter` in a sequence of three synchronizations.

insert _ : cassette;	connect _ : monitor;
play;	connect _ : camera;
record;	disconnect monitor;
stop;	disconnect camera;

Figure 6. Simplified interface of VCR (in CO-OPN notation underscores denote places for parameters).

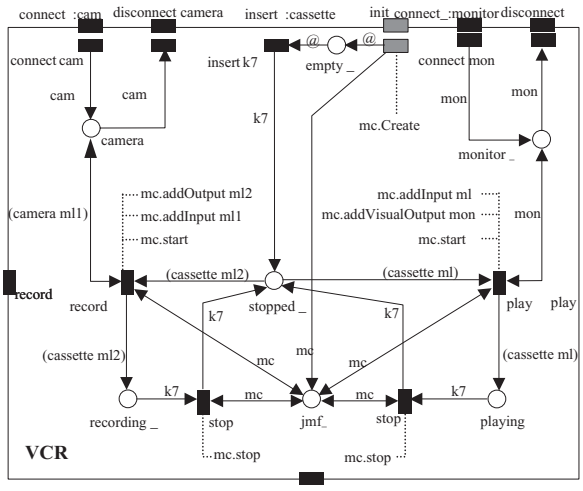

Figure 7. Behavior of VCR expressed as a CO-OPN network.

Creation:	build _ : dataSource;		
Methods:	start;	output _ : dataSource;	
	stop;	close;	configure _ : content;

Figure 8. Interface of a processor.

The `record` method proceeds analogously. Note that those two methods extract the `MediaLocator` from the higher-level abstractions, `cassette` and `camera`.

4.2. JMF Model

There is also a model of JMF. This model is composed of classes: `Processor`, `DataSink`, `DataSource`, `Component`, and ADTs: `Content` and `MediaLocator`. These entities model a simplified abstraction of JMF, including dynamic aspects. For example, dynamic creation of a new `DataSource` by a `Processor` in response to a `getDataOutput` request is modeled. The interface of the Processor is presented in Figure 8 and the CO-OPN class in Figure 9.

The possible states of `Processor` are described by the ADT `ProcessorStates`. The state is explicitly stored in place `state`. The methods of the `Processor` class move the processor from one state to another accordingly to the documented behavior of real JMF object. The method `realize` creates a `DataSource` or a GUI `Component` depending on the contents of the place `format_`. Methods `dataOutput_` and `visualComponent_` give access to one of the possible outputs.

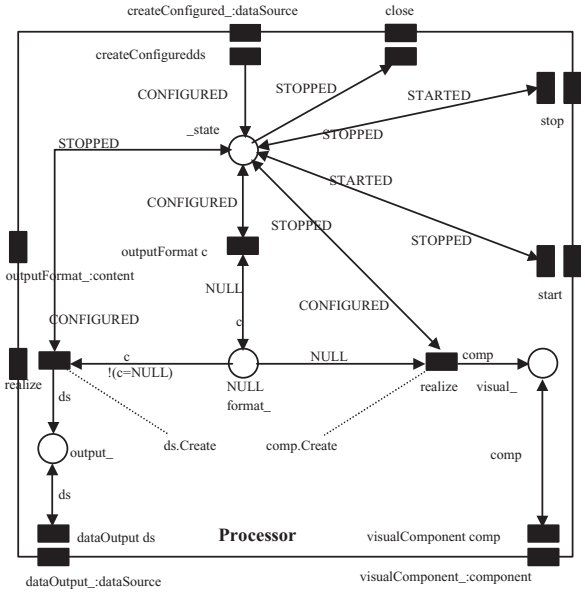

Figure 9. Behavior of processor expressed as CO-OPN network.

addInput _ : mediaLocator;	start;
addOutput _ : mediaLocator;	stop;
addVisualOutput _ : monitor;	

Figure 10. Interface of MediaCenter class.

4.3. JMF and VCR Interconnection by Means of MediaCenter

Finally, to connect these two parts, we define a class, named `MediaCenter` (Figure 10), which is a kind of manager of JMF. `VCR` uses `MediaCenter` to implement its functionality.

The working scheme of `MediaCenter` is quite simple and independent of the desired operation (play a cassette, record the video from external source or other). All possible operations are abstracted by the method `start`. This method creates different configurations of JMF in function of inputs and outputs submitted to `MediaCenter`. If there is no configuration that fit submitted inputs/outputs this method fails.

Method `start` has a nondeterministic definition: it tries to find a configuration that matches submitted inputs and outputs (Figure 11). For example, if the input is a `MediaLocator` that points to a file and the output is a `Monitor`, then the `MediaCenter` will create, configure, and put to work a chain of JMF components that play a video from a file. More concretely, a `DataSource` will be created and then used to create a `Processor`. The `Processor` will first be asked to produce a video output and then moved to `Realized` state. The video display (`Renderer` in JMF

```
start With
 dsrc.build(ml1)..
 p.createConfigured
   (dsrc)  ..
 p.realize ..
 p.visualComponent
   (comp)  ..
 mon.show(comp)..
 p.start::

input ml1, monitor mon
->
monitor mon,
processor p;
```

```
start With
  dsrc1.build (ml1) ..
  p.createConfigured
(dsrc1) ..
  p.outputFormat   (AVI)
..
  p.realize ..
  p.dataOutput(dsrc2)..
  dsk.build(dsrc2,  ml2)
..
  dsk.open  ..
  dsk.start ..
  p . start ::
input ml1, output ml2
->
processor p, dsink dsk;
```

Figure 11. Definition of the *start* method.

terms) will be acquired from the realized `Processor` and passed as argument to the `show` method of `Monitor`. Finally, the chain will be started by a call to the `start` method of the `Processor`. The definition of `start` is based on many choices but its behavior is mostly from an external point of view deterministic. The crucial point is that it is not easy to eliminate the exploration of the possibilities expressed in the behavioral definition when methods are evaluated.

5. Prototypes

We define a prototype as a program generated automatically from a formal specification. Such a specification is expressed in a high-level specification language with formally defined semantics, in our case CO-OPN [10]. The prototype must fit the semantics defined by the specification, and this can be done in multiple ways. For example, one specification can be translated to various programming languages, adopt diverse execution models, or use different algorithms. For one specification there are a large number of prototypes to investigate. In this paper, variations in the interface between generated code and the rest of the system are of particular interest.

We have defined a systematic approach [3], [4] to produce a Java prototype from a CO-OPN specification. We now extend this approach to fit real object-oriented systems in a better way. In our improved approach, the main characteristics of the generated prototype remain unchanged; we only add new features to the interface between generated code and external objects that have to be (re) used.

5.1. Characteristics of Generated Prototype

Independently of its interface, we generate the prototype code with the characteristics that follow.

The generated prototype preserves the object structure of a CO-OPN specification. A class or a context in a CO-OPN specification is always translated to a class in the target OO language.

While the CO-OPN classes are defined using Petri Nets, generated classes are always reactive. Indeed, the stabilization semantics of CO-OPN [2] corresponds better to reactive then to active objects.

The methods of a CO-OPN class (its inputs) are mapped to the methods of the generated class. The gates (outputs) of a CO-OPN class correspond to callbacks provided by the generated class. When generating Java code, we use the standard notion of JavaBean event [5] to represent those callbacks.

CO-OPN specifications allow nondeterminism and transactions. For this reason, the execution model of generated code is close to resolution [6]. This implies the use of a backtracking mechanism, like in Prolog, in order to compute the results of an invocation. This aspect differentiates automatically generated code principles to manually generated code in which choice points can be coded more cleverly by introducing design choices.

Let us explain this a little more with an example: suppose we have a method called `start` that nondeterministically chooses between two actions: `play` and `record`. When we execute `start` we do not know which action to choose, so we choose one arbitrarily. Suppose that we choose the wrong one (that will later fail), say `play`, and then continue the execution. At some moment the execution reaches a failure. As we have made arbitrary choices, we have to reconsider them. So we go back in the execution path, canceling their effects (*undo*), up to the last choice point. So we get back to `start`, undo the effect of `play`, and look for alternatives. As the `record` action has not been tried yet we *redo* the method `start` by executing `record`, then continue the execution and finally reach success.

To implement this undo–redo mechanism, the prototype objects have to keep some information about executed actions. Moreover, there is another meta-operation called "commit" which notifies objects that there is no more need to keep this information. Commit is invoked when some global execution reaches a success. Commit is propagated to all objects that participated in the execution. The effect of commit is that the system assumes its new state. No undo is possible anymore.

As we shall see, this undo–redo mechanism is an obstacle to the connection between the generated code and existing software libraries or other external systems because this computational behavior is not that of imperative languages.

More concretely, if we already start reading the video stream of a pay TV channel because it is one possible behavior of the system, and if we discover lately that this behavior cannot be executed in the system environment the provider will charge the user. In this case, from a commercial point of view, reading streams is a nonreversible action and cannot be undone.

5.2. Simple Interface

Our first approach was to generate the prototype as a single black box object that accepts some inputs and produces some output signals. From the programmer's point of view, such prototype is a single Java object.

This kind of prototype fits the case where the system is encapsulated into a single object, for example, in a controller software for some physical equipment [3].

In CO-OPN this kind of system is often represented by a single top-level context encapsulating other objects and sub-contexts.

In the case of systems composed of multiple objects, some of which already exist in the real world and others of which appear only in the model, a prototype with a unique interface is useless.

For example, let us examine the model of the VCR program. This program manipulates objects from an existing library: it creates them and calls their methods. For instance, to see the video on the screen, the `DataSource`, `DataSync`, and `Processor` objects from JMF library should be instantiated and used by the generated code.

Furthermore, in order to model the JMF behavior, modeled library objects should interact between them. For example, Processor creates a Data Source or a Renderer object. Clearly, a simple interface does not suffice for the prototype of the VCR system.

5.3. Object-Oriented Interface

Before explaining our technique we formulate two important adequacy assumptions on library objects and their models.

- First, the library objects that we use are deterministic and, supposedly, behave exactly as defined in the model. This can be checked using other techniques, such as the test selection techniques we already developed [13].

- Second, the model is complete enough (not depending on real objects during execution choices), so that we do not need to interact with the real objects when executing the prototype.

The prototype of a library object acts as a proxy. It first participates in the execution without interaction with the library object and collects the trace of required actions, and then, if success is reached, invokes the corresponding actions of a real object. If the execution of the prototype fails then there is no interaction with the real object.

To implement this behavior, prototypes of library objects are divided into three parts (Figure 12). The first part, that we will call "pure prototype," is essentially the kind of code that is generated for any CO-OPN class. This code implements the Petri Net defined in the model and the backtracking mechanism. The difference is in the implementation of the `commit` meta-operation. The second part, called `proxy`, plays the role of an interpreter between the generated code and the library object. The proxy has the same interface as the pure prototype. The last part is the real object.

The interaction between the prototype and the real object is delayed. All backtracking actions are executed only on the "pure prototype" part of the generated code. Consequently, we do not need any undo–redo mechanism on real objects.

Let us take a closer look at the execution of the `commit` operation. It is important to note that our `commit` is composed of multiple actions. Each action corresponds to one method execution. These actions are executed in the same order as the methods

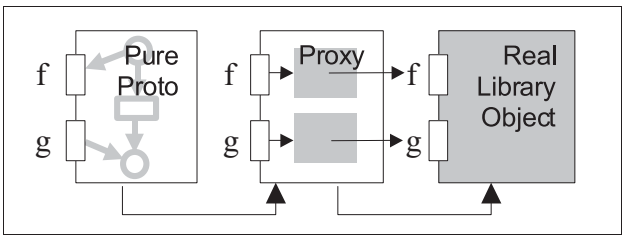

Figure 12. Prototype of a library object.

were. The method names and arguments are collected by the `commit` (it is in fact a sequential program that is collected). The `commit` operation calls the same methods with the same arguments in the same order on the proxy. Then, the proxy transforms the call and interacts with real objects.

One important point is that not all invocations are repeated on the proxy. Interactions between library objects are not repeated. We suppose that library objects themselves perform these interactions. Only the invocations coming from pure model (nonlibrary) parts of the system are repeated.

The proxy class is generated as skeleton. The user has to program by hand the interaction with real objects. In fact, there are a number of interactions that are easy to program by hand, but complex or near to impossible to program automatically. Examples are interactions with asynchronous code, and determination of the dataflow (see Section 6.1 for more examples).

The disadvantage of this approach is that the user should have knowledge of the target programming language and should understand the details of the execution model of the prototype. Fully automating the generation of these proxies is future work and needs additional information that should be provided (for instance, the directions of dataflow) in order to realize this generation. Alternatively, this drawback can be avoided by building libraries of models and corresponding proxy objects.

6. The Prototype of VCR

6.1. Structure

For each CO-OPN class that corresponds to a JMF object, the code is generated as described in Section 5.3. For example, for the CO-OPN class `Processor` presented in Section 3, two Java classes are generated: `Processor` and `ProcessorProxy`. `Processor` has a reference to `ProcessorProxy`. In turn, the latter has a reference on the real processor: `javax.media.Processor`.

As said previously, `Processor` and `ProcessorProxy` present the same interface, i.e., java counterparts of CO-OPN methods `createConfigured`, `realize`, `start`, etc. The `Processor` class implements the Petri Net from the CO-OPN specification.

`ProcessorProxy` is generated with empty methods. The user completes the definition of `ProcessorProxy` by filling methods with code.

The simplest methods are `stop`, `start`, and `close`. They simply forward the request to the library object (`proc` is the reference on `javax.media.Processor` object):

```
public void start(){
   proc.start();
}
```

The `realize` method is more complex. For real objects it is asynchronous: it immediately returns, before the `Processor` moves to `Realized` state. Later, an event is sent to notify that the work is done. Calling some methods before the notification may lead to error. On the other hand, the prototype has synchronous semantics, i.e., when the call terminates all work must be done. To connect those two paradigms the method `realize` waits for the actual end of the processing:

```
public void realize(){
  proc.realize();
  waitForState(Processor.Realized);
}
```

6.2. Transforming Prototype Parameters into Real Parameters

Methods with parameters need another adaptation. First, parameter data types of a prototype method are also prototypes. We have to extract real objects from those prototypes and pass them as arguments of a real method.

Second, CO-OPN does not define the dataflow of parameters, but Java does. For example, the real Processor has two methods:

```
void setContentDescriptor(ContentDescriptor)
ContentDescriptor getContentDescriptor().
```

In the model one method suffices: `contentDescriptor _`. The direction of the data-flow is determined at runtime. When interacting with the real object we have to choose the appropriate method: `set` or `get`.

```
public content contentDescriptor_( content arg1){
     if(... parameter arg1 is ''in'' ...)
             proc.setContentDescriptor(
             arg1.getContentDescriptor());
     else // parameter arg1 is ''out''
          arg1.setContentDescriptor(
          proc.getContentDescriptor());
  return arg1;}
```

This procedure mimics the unification mechanism, where prototype objects are logical variables and real objects are possible values. The unification mechanism is one of the natural techniques used for the operational interpretation of CO-OPN models in our tools.

6.3. *Initialization Method*

Finally, the most complicated code belongs to the initialization method: `createConfigured_`. In fact, the real `Processor` having no such operation, the model abstracts two consecutive steps: the creation of a processor with a given Data Source, and then it moves it to the CONFIGURED state. The creation is not achieved by a constructor of the `Processor` class, but by a static factory method of another class (`javax.media.Manager`). Moreover, the `configure()` method is asynchronous.

6.4. *Example of Execution Semantics*

Let us examine the execution of the following sequence:

```
VCR.insert k7(ml1) ..
VCR.connect mon    ..
VCR.play
```

The execution of these three operations should open a window and play a video inside. `VCR`, `mon`, and `ml1` have already been created and initialized (Figure 13). For instance, `mon` references (via a proxy) a `javax.swing.JFrame` object and `ml1` points to a valid `javax.media.MediaLocator`.

First, the sequence is evaluated by the pure prototype part of generated code. The execution finds a sequence of interactions that leads to success. At this stage, the execution does not yet interact with real library objects.

After that, `commit` is executed. As seen previously, the `commit` operation will follow the path of execution in order to validate the results of each call. If we just keep the interactions that have to be repeated, we obtain the following simplified

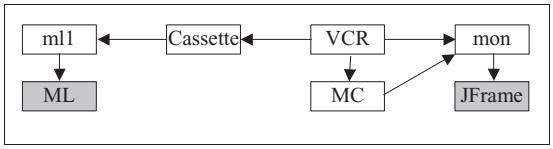

Figure 13. Before the execution of `VCR.play`. White rectangles: generated classes, gray rectangles: objects from Java libraries.

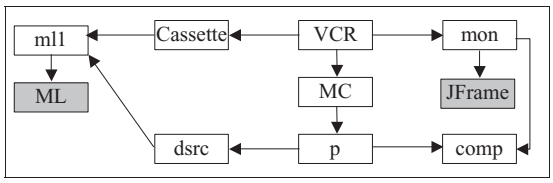

Figure 14. After the execution of VCR play.

sequence of method invocations:

```
dsrc.build (ml1)          ..
p.createConfigured (dsrc)  ..
p.realize                 ..
p.visualComponent (comp) ..
mon.show (comp)           ..
p.start
```

In CO-OPN, methods are synchronous; this means that each invocation has to be completed before the next invocation starts. It is important to note that the above expression does not contain interactions between library objects or pure specification objects.

In this expression `dsrc`, `ml1`, `p`, `mon`, and `comp` are variables. Only the `mon` (monitor) and `ml1` (media locator of the input) variables have a value before the execution, the others being empty. The variables receive values during the execution of the sequence. In fact, in CO-OPN, the expression `dscr.build`, where `dsrc` is a variable of type `DataSource` and `build` is a creation method, stands for the creation of a new object and assignment of a reference to `dsrc`.

At commit time, all these operations have already been executed on the prototype objects and the variables have received their values (Figure 14). Nevertheless, real objects have not yet been created (except for the `Monitor` and the `MediaLocator`). The execution of `commit` repeats those operations on real objects, including creation.

For example, the result of `commit` of `p.createConfigured(dsrc)` is:

1. Extract the real part from `dsrc`, i.e., the object of type `javax.media.DataSource`.

2. Create, as described in Section 6.1, a new `javax.media.Processor` using the `DataSource`.

3. Store a reference onto the `javax.media.Processor` in the appropriate variable of `ProcessorProxy` object.

By continuing in the same way, we obtain the creation, connection, configuring, and starting of a chain of the JMF components (gray part of Figure 15). Then a window appears on the screen and the video is played inside.

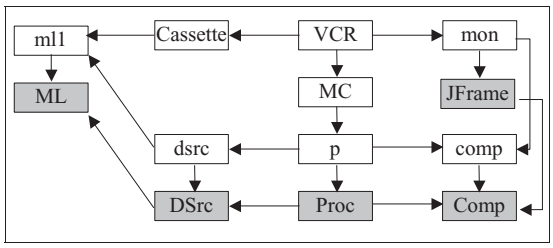

Figure 15. After execution of commit.

7. Related Work

The usefulness of executable specifications for rapid prototyping and validation of systems is widely accepted [14], [15]. Nevertheless, rapid prototyping is often seen as a kind of animation. In our work, we push the executable specification to interact with real systems. By the way, it is also possible to use generated code for animation or other validation-related tasks.

The execution of specifications in the domain of Petri Nets is widely used because of the latter's simple operational semantics (e.g., see [16]). Even if CO-OPN is partially based on Petri Nets there is an additional complexity in operational evaluation due to the algebraic aspects and the transactional semantics of synchronizations.

Interfacing Prolog with procedural languages also needs similar concepts in order to deal with backtracking by means of iterators or using always-succeeding clauses.

Delayed execution was studied and used in domains such as fault tolerance or database systems. In [7] a taxonomy of fault-tolerance techniques applied to non-program objects is given (for instance, external physical objects that cannot be undone). Delayed execution plays an important role in these techniques. In particular, it makes it possible to combine undo–redo techniques and nonreversible operations of external objects. Interestingly, authors use ADT to model the behavior of the external objects (operation being deterministic). Our approach is similar, although we use delayed execution for different purposes. Moreover, our modeling language is more expressive than classical ADT description languages.

Another similar technique is planning [8]. In fact, we use a model of external library objects to generate a correct plan and then we execute it on real objects. Modern planning techniques, like Model Based Planning [9], may help extend our approach to nondeterministic external objects.

8. Conclusions

In this paper we have shown how to combine formal specifications with existing software libraries. Due to the expressivity of our specification language, our automatically generated prototype code has nonstandard execution model (nondeterministic, transactional, synchronous). This aspect prevents the simple linking between

existing sequential code and prototype. Instead, we use object proxies combined with delayed execution.

Our plans are to extend our techniques to handle nondeterminist external objects. We would also like to investigate the possibility of semiautomatic generation of proxies.

References

1. Java Media Framework. http://java.sun.com/products/java-media/jmf/index.html.
2. Biberstein, O., D. Buchs, and N. Guelfi. Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism. In Agha, G., F. De Cindio, and G. Rozenberg (Eds) *Advances in Petri Nets on Object-Orientation*, Lecture Notes in Computer Science, no. 2001, Springer-Verlag, May 2001, pp. 70–127.
3. Chachkov, S., and D. Buchs. From an Abstract Object-Oriented Model to a Ready-to-Use Embedded System Controller. *Rapid System Prototyping*. IEEE Computer Society Press, Monterey, CA, June 2001a, pp. 142–148.
4. Chachkov, S., and D. Buchs, From Formal Specifications to Ready-to-Use Software Components: The Concurrent Object-Oriented Petri Net Approach. *International Conference on Application of Concurrency to System Design*. IEEE Computer Society Press, Newcastle, June 2001b, pp. 99–110.
5. Sun Microsystems. *Java Beans Specification v.1.01*. July 1997, pp. 24–32.
6. Padawitz, P. *Computing in Horn Clause Theories*. Springer, 1988.
7. Romanovsky, A. B., and I. V. Shturtz. Unplanned Recovery for Non-program Objects. *Computer Systems Science and Engineering*, 1993, vol. 2, pp. 72–79.
8. Warren, D. Generating Conditional Plans and Programs. Proceedings of AISB-76, 1976.
9. Cimatti, A., M. Pistore, M. Roveri, and P. Traverso. Weak, Strong and Strong Cycling Planning via Symbolic Model Checking. IRST Technical Report, April 2001, submitted to Journal of AI.
10. Buchs, D., and N. Guelfi. A Formal Specification Framework for Object-Oriented Distributed Systems. *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 635–652, 2000.
11. Buchs, D., and M. Buffo. Rapid prototyping of Formally Modelled Distributed Systems. In Frances M. Titsworth (Ed.), *Proceedings of the Tenth International Workshop on Rapid System Prototyping RSP'99. IEEE*, June 1999.
12. Buffo, M. Experiences in Coordination Programming. In Proceedings of the Workshops of DEXA'98 (Int. Conf. on Database and Expert Systems Applications). *IEEE*, Aug 1998.
13. Barbey, S., D. Buchs, and C. Péraire. A Theory of Specification-Based Testing for Object-Oriented Software. *Proceedings of EDCC2*, Taormina, Italy, October 1996, LNCS (Lecture Notes in Computer Science) 1150, 1996, pp. 303–320.
14. Nota, G., and G. Pacini. Querying of Executable Software Specifications. *IEEE Transactions on Software Engineering*, vol. 18, no. 8, 1992.
15. Ghezzi, C., D. Mandrioli, and Morzenti, A. TRIO, a Logic Language for Executable Specifications of Real-Time Systems. In Proceedings of 10th French-Tunisian Seminar on Computer Science, pp. 322–349, 1989.
16. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Monographs in Theoretical Computer Science, Springer-Verlag, 1994.