Polymorphism and Module-Reuse Mechanisms for Algebraic Petri Nets in CoopnTools*

Mathieu Buffo[1], Didier Buchs[1]

[1] Software Engineering Laboratory, Swiss Federal Institute for Technology,
1015 Lausanne, Switzerland
{Mathieu.Buffo,Didier.Buchs}@epfl.ch

Abstract. This paper introduces CoopnTools, a tool set allowing the support of object-oriented specifications written by means of the language CO-OPN$_{/2}$, based on synchronised algebraic Petri nets. In particular, this paper shows how concrete mechanisms dealing with polymorphism and module-reuse are implemented in CoopnTools.

1 Introduction

According to software engineering principles, the development of a correct distributed software requires to elaborate its specification, i.e. a description of what the proposed software must do. The specification should be formalised by means of a suitable language, allowing developers to expose their wishes, as well as analysis tools to verify assumed properties.

The CO-OPN language (Concurrent Object-Oriented Petri Nets) was proposed as an apposite specification language, based on synchronised algebraic Petri nets [3]. Moreover, a tool set, called SANDS (Structured Algebraic Nets Development System), was offered as a support for the language [4] [5] .

Strictly speaking, CO-OPN is an object-based language, and a new version, called CO-OPN$_{/2}$, was proposed to deal with the full paradigm of objects [1] [2]. In particular, CO-OPN$_{/2}$ includes the notion of classes and types. In addition, a coordination language and model, COIL (Contexts and Objects Interface Language) is joint to CO-OPN$_{/2}$ [7], allowing to refine specifications into component-based software architectures [6]. The differences between both versions of the language, and the introduction of COIL, was so important that we decided to implement a new tool set for the support of CO-OPN$_{/2}$, called CoopnTools.

The goal of this paper is to introduce the CoopnTools tool set, and to explains how concrete mechanisms dealing with polymorphism and module-reuse are implemented in CoopnTools. The next section briefly introduces CO-OPN$_{/2}$, and Section 3 states the main characteristics of CoopnTools. Then, Sections 4 and 5 presents the polymorphism and the module-reuse mechanisms, respectively. Finally, Section 6 concludes the paper.

* This work has been partially sponsored by the Esprit Long Term Research Project 20072 "Design for Validation" (DeVa) with the financial support of the OFES, and by the Swiss National Science Foundation project "Formal Methods for Concurrent Systems".

2 The Object-Oriented Specification Language CO-OPN$_{/2}$

CO-OPN$_{/2}$ concrete specifications are collections of ADT and class *modules* [1] [2]. Syntactically, each module has the same overall structure; it includes an *interface section* defining all elements accessible form the outside, and a *body section* including the local aspects private to the module. Each section is composed by *fields*, themselves declaring *symbols*. A symbol is the association of a *name* and its *definition*; names or declarations can be optional, according to the particular kinds of symbols.

CO-OPN$_{/2}$ keywords are reserved and case-insensitive. On the contrary, names are case-sensitive. Semi-colons are used as terminators. Single-line comments are introduced by the symbol ";;", while multi-line comments are place inside "(:" and ":)".

2.1 ADT modules

CO-OPN$_{/2}$ ADT modules define data types by means of algebraic specifications. Below is the code of a simple abstract data type defining two messages "M" and "M′ ". The *interface* section is introduced by the keyword "Interface" and corresponds to signatures in algebraic specifications. The *sort* field defines a unique sort called "message". The *generator* and *operation* fields define the algebraic operations. For convenience reasons, operations generating new values into the carrier sets are called generators. In other words, this signature defines two values and one operation. The *body* section is introduced by the keyword "Body". The *axiom* and *where* fields define respectively the axioms and the variables of the ADT.

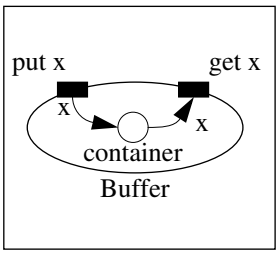
```
ADT Message;    (: Data type for two messages :)
Interface              ;; start of the interface section
    Sort message;
    Generators M, M' : -> message;
    Operation other _ : message -> message;
Body                   ;; start of the body section
    Axioms  m=M  => other m  = M';
            m=M' => other m  = M;
    Where m : message;
End Message;
```

2.2 Class Modules

CO-OPN$_{/2}$ classes are described by means of modular algebraic Petri nets with two kind of transitions. First there are internal transitions (usual *transitions* of Petri nets), which correspond to spontaneous internal reactions. Then, there are parameterised external transitions (*methods*), which are events visible from the outside. The transitions are defined by so-called *behavioural formulas*. Cooperation between objects is achieved by means of a synchronization mechanism between a transition and a method (*synchronous method call*). In addition, synchronizations between method calls are provided by means of the tree operators "//" for concurrency, ".." for sequence and "⊕" for alternative of method calls.

Below is the code and the associated Petri net of a class modelling a simple buffer of messages. The *use* field of the interface list all required modules inside the class; an use field in the body would have indicted a list of modules required inside the body only. The *type* field defined the sort of the ADT induced by the class. The *object* field declares a static object B of the class. Finally, the *method* field allows the declaration of two methods for accessing the buffer. The *place* field declares a place used in the net. Like ADT, classes bodies have *axiom* and *where* fields, but axioms are based upon behavioural formulas. In our case, two axioms define the behaviours of the methods.

```
Class Buffer;
Interface
    Use Message;
    Type buffer;
    Object B : buffer;
    Methods put _ ,  get _ : message;
Body
    Place container _ : message;
    Axioms
        put x :: -> container x;
        get x :: container x -> ;
    Where x : message;
End Buffer;
```



2.3 COIL

COIL is a coordination language for CO-OPN$_{/2}$ [6] [7]. It adds a third kind of modules to CO-OPN$_{/2}$, the *context* modules, which share the same overall structure with ADT and class modules. Due to lack of space and to the fact that COIL is more related to the coordination theory than to Petri nets, context modules are not illustrated here.

3 CoopnTools

CoopnTools is a tool set for the support of CO-OPN$_{/2}$ and COIL. It's aim is to support the so-called *CO-OPN development method*, from the first informal specification up to the last incremental implementation, as explained in [4]. Obviously, CO-OPN$_{/2}$ and COIL have a crucial role in this development method; CO-OPN$_{/2}$ is the basic formalism, used during the *formal specification phase* as well as during the *design phase*, while COIL is introduced during the *design phase* to exhibit the proposed *software architecture*. CoopnTools is currently under development, but the core of the tools, dealing with the semantics of the language, i.e. dealing with algebraic Petri nets, is about to be finished yet. For portability reasons CoopnTools is written is Java, and it requires no licenses - at least in its basic version. CoopnTools is freely available on the Web; interested readers can contact the authors or consult the page "`http://lglwww.epfl.ch`".

CoopnTools is composed of four parts, namely the CFC, the documentation, the examples, and the tools themselves:

- the CFC (CO-OPN Foundation Classes) is a library of standard ADT, classes and contexts; it includes the definitions of the basic data types such as BlackTokens, Booleans, Naturals, Integers and Strings for instance;
- the documentation covers the supported language as well as the tools; it is provided as a set of HTML pages, following a Java-like practice;
- the examples cover CO-OPN$_{/2}$ as well as COIL; they are documented too;
- the tools themselves includes yet tools for checking the semantics of the languages and tools allowing pretty-printing of source code. Tools for graphical visualisation of Petri nets, for simulation of Petri nets and for automatic Java-like documentation generation are currently under development. Finally, we plan to introduce in a near future the test set generator and the incremental prototyper, which was part of the former SANDS tool set, and to develop a new tool acting as a bridge between Fusion and CO-OPN$_{/2}$.

CoopnTools implied the definition of concrete mechanisms, such as the mechanisms dealing with polymorphism and module-reuse. These mechanism were never described up to now and one of the goal of this paper is to present them in a pragmatical way. The next two sections are devoted to a survey of these mechanisms.

4 Polymorphism in CoopnTools

A great challenge for modern computer languages is to incorporate facilities for data type management. Important flexibility in terms of data type management is achieved by used the concept of *polymorphism*. When a software element can have more than one type, or can work on more than one type, we say that this element is *polymorphic*. CoopnTools supports three kinds of polymorphism, namely the polymorphism by *genericity*, the polymorphism by *overloading* and the polymorphism by *subtyping*.

4.1 Genericity

CoopnTools allows the definition of generic modules, which are kinds of templates of modules; they must be instantiated to produce a normal module. Generic modules can have parameters, which actually are modules, the elements of which are used to build the generic modules. Below is the code of a generic buffer. The buffer has one parameter, defining abstract items for the buffer. Generic modules, like parameter modules, have no existence during the execution-time of a specification; they are merely syntactic constructions.

```
Parameter ADT Item;
Interface
    Sort item;
End Item;
```

```
Generic Class GenBuffer(Item);
Interface
    Use Item, Message;
    Type genBuffer;
    Object B : genBuffer;
    Methods put _ ,  get _ : item;
Body
    Place container _ : item;
    Axioms
        put x :: -> container x;
        get x :: container x -> ;
    Where x : item;
End Buffer;
```

The instantiation of a module is the mechanism of making a module from another one, solving (part of) its parameters. This mechanism requires the application of a so-called instantiation morphism, allowing to map formal to effective parameters, and to rename the element of the source module. Instantiation morphism are described by a fourth kind of modules defined in CoopnTools, the morphism modules. A morphism module contains only two fields, describing the parameter morphism and the rename one. Below is the code of the instantiation morphism allowing to obtain the buffer class seen above from the instantiation of generic buffers.

```
Morphism Morph;
    Morphism item -> message;
    Rename genBuffer -> buffer;
End Morph;
```

The instantiation of a module is denoted by the name of the source module, followed by the list of the names of the effective parameters and by the name of the morphism module. For instance, the name of the instantiated buffer is: "`GenBuffer(Message):Morph`". CoopnTools supports *partial instantiation* (instantiation in which at least one effective parameter is itself a *parameter module*, and the resulting module still a *generic module*), and *automatic instantiation* (instantiation depending on another instantiation).

4.2 Overloading

Overloading polymorphism is achieved by allowing a single name denoting several different notions, as for instance the name "+" denoting the usual addition as well as string concatenation. CO-OPN$/_2$ allows the definition of several symbols with the same name, discriminating them according to their kind, their profile and the module in which they are defined.

CoopnTools defines an *overloading resolution method*, allowing to know, for each encountered name, to which symbols it refers. According to this method, all symbols

of adequate kind, with same name and satisfying the typing system are elibible for the resolution of a name. During this resolution, non-local symbols are masked by local symbols with same name, kind and profile, but masked symbols are accessible by suffixing their name with the keyword "In" followed by the name of the definition module. At the end, if just one eligible symbol exists then it is the resolution, else the name has no resolution and the code is erroneous.

For instance, consider the three modules shown below, which define overloaded sorts. One of the four operations has a co-domain which has no resolution.

```
ADT A;
Interface
    Sorts s1,s2;
End A;
```

```
ADT B;
Interface
    Sorts s1,s2,s3;
End B;
```

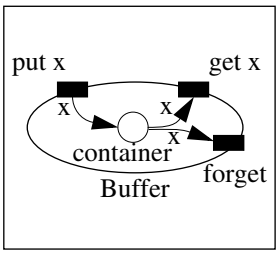
```
ADT C;
Interface
    Use A,B;
    Sort s1;
    Operations
        o1 : s1;        ;; correct: local s1
        o2 : s2;        ;; erroneous: which s2? in A? in B?
        o3 : s2 In A;   ;; correct: s2 in A
        o4 : s3;        ;; correct: s3 in B
End C;
```

4.3 Sub-Typing

CO-OPN$_{/2}$ is an object-oriented language supporting behavioural sub-typing according to the strong substitutability principle; in short, an instance of a sub-type can be used in any context in which an instance of a super-type is expected, provided that the behaviour of the system remains unchanged.

CO-OPN$_{/2}$ supports sub-sorts (for ADT) and sub-types (for classes) by means of apposite declarations in the interface of modules. For instance, below is the code of a kind of buffer which forget messages; each time the method "forget" is called, a (randomly chosen) message is forgotten. It can be shown that this kind of buffer is a subtype of the standard buffers described above, according to their formal semantics, and this should be intuitive for experts in Petri nets. Accordingly, the new buffer class is marked as a sub-type of the standard buffers, and its instances can be used each time buffers are expected.

```
Class FBuffer;      ;; buffer with forget
Interface
    Use Message, Buffer;
    Type fbuffer;
    Subtype fbuffer -> buffer;
    Object B : fbuffer;
    Methods put _ ,  get _ : message;
            forget;
Body
    Place container _ : message;
    Axioms
        put x :: -> container x;
        get x :: container x -> ;
        forget :: container x -> ;
    Where x : message;
End Buffer;
```



CoopnTools use sub-sort and sub-type declarations to solve *overridden* symbols. Overridden symbols are kinds of overloaded symbols, the profile of which are related by sub-typing. For instance, the static objects "`B`" of types "`buffer`" and "`fbuffer`" are overridden symbols. The *overriding resolution method* is similar to the overloading resolution method, with the exception that symbols with smallest sorts and types mask symbols with greater sorts and types, according to *covariant* rules. Hence, among the two objects "`B`" shown above, the object with type "`fbuffer`" masks the normal buffer object.

Finally, CoopnTools defines a special type, called "`(Object)`", as the type of an empty class (an empty Petri net). This type is a super-type of all class-types, and it can represent any object in a system; it is the root of the type hierarchy.

5 Module-Reuse as Defined in CoopnTools

It is often fruitful to reuse existing modules in the definition of new ones. This mechanism is of great importance in object-orientation, where it is known as inheritance or sub-classing. We must mention that *usually*, the term inheritance refers to both processes of sub-classing and sub-typing; however, modern object-oriented languages tend to separate both concerns, just like CO-OPN$_{/2}$ does.

CoopnTools allows the definition of asbtract modules, i.e. modules the implementation of which is not (completely) defined. Abstract modules are marked by the keyword "`abstract`".

CoopnTools support module reuse by allowing a CO-OPN$_{/2}$ module to *include* another CO-OPN$_{/2}$ modules; the included modules are *merged* to the local module. CoopnTools provide two slightly different mechanism for module reuse, the *copy mechanism* and the *inheritance*.

5.1 Copy Mechanism

The copy mechanism works at the semantic level; the semantics of the source module is merged to the semantics of the target module. Abstract modules cannot be merged with non-abstract ones. The copy mechanism is often used in conjunction with the genericity to define specific instantiated modules. For instance, the code below provides a dedicated module (including a specific name) to the instantiation of the generic buffer shown above. The copy mechanism is introduced by the keyword "`As`" in the header of the module; at most one "`As`" is present in one module.

```
Class MessageBuffer As GenBuffer(Message):Morph;
End MessageBuffer;
```

5.2 Inheritance

The inheritance mechanism work at the syntactic level; the symbols of the source module are merged to the symbols of the target module, and every symbols are then solved (using the overloading and overriding resolution methods) in the context of the target module. During the merge, symbols can be renamed or deleted. Multiple inheritance (i.e. inheritance of many modules) is allowed; in this case, possible pragmatical problems are avoided by inheriting each single symbol once. Moreover, CoopnTools removes the abstract tag of inherited modules; normal modules can inherit from asbtract ones.

CoopnTools' inheritance is useful for source code reuse. For instance, recall the buffers with the forget capability. The definition given above can be shortened by using inheritance, as shown below (for illustration purposes, the variable "x" is deleted, and then redefined). Each inherited module is handled by an inheritance section, stating the operations (renaming and deletion) to perform during the merge of symbols. Deletions are of two types: *undefine*, which simply deletes symbols with the related name, and *redefine* (for method symbols only), which deletes the axioms dealing with the symbol.

```
Class FBuffer;      ;; buffer with forget
Inherit Buffer;                  ;; symbols of buffer are added
    Rename buffer -> fbuffer;    ;; but buffer is renamed fbuffer
    Undefine x;                  ;; x is undefined here
Interface
    Use Buffer;
    Subtype fbuffer -> buffer;
    Method forget;
Body
    Axioms
        forget :: container x -> ;
    Where x : message;           ;; x is redefined here...
End FBuffer;
```

6 Conclusions

CO-OPN$_{/2}$ and the related CO-OPN development methods are now well diffused among the Petri nets community [1] [2] [3] [4] [5]. This paper introduces the new CoopnTools tool set for CO-OPN$_{/2}$ and COIL, and explains how concrete mechanisms dealing with polymorphism and module-reuse are defined, through some examples. Despite the fact that these mechanisms are not of primary interest when speaking about CO-OPN$_{/2}$, they form a sound and useful base for building high-level specifications. For instance, more than 90 percent of the modules member of the standard CO-OPN$_{/2}$ library (CFC) are built using polymorphism and module reuse.

We are currently working on extending the tools of CoopnTools, as mentioned above. In addition, we plan to add a "Java" library to the CFC, for specifying the most useful elements of the Java Foundation Classes.

References

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

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

[3] Didier Buchs and Nicolas Guelfi. CO-OPN: A concurrent object-oriented Petri nets approach for system specification. In M. Silva, editor, *12th International Conference on Application and Theory of Petri Nets*, pages 432–454, Aahrus, Denmark, June 1991.

[4] Didier Buchs, Jarle Hulaas, and Pascal Racloz. Exploiting various levels of semantics in CO-OPN for the SANDS environment tools. In *Tool Presentations, International Conference on Application and Theory of Petri Nets ICATPN'97*, pages 34–43, Toulouse, France, June 1997.

[5] Didier Buchs, Jarle Hulaas, Pascal Racloz, Mathieu Buffo, Jacques Flumet, and Erik Urland. SANDS structured algebraic net development system for co-opn. *16th International Conference on Application and Theory of Petri Nets*, pages 45–53, June 1995. Turin-Italie.

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

[7] Mathieu Buffo and Didier Buchs. A coordination model for distributed object systems. In *Proceedings of the Second International Conference on Coordination Models and Languages COORDINATION'97*, volume 1282 of *Lecture Notes in Computer Science*, pages 410–413. Springer Verlag, 1997.