Modeling and Verification of sC++ Applications

Thierry CATTEL[1]

[1] Computer Networking Laboratory, Swiss Federal Institute
CH-1015 Lausanne, Switzerland
cattel@epfl.ch

Abstract - *This paper presents a means to model and verify concurrent applications written in sC++. sC++ is an extension of C++ that adds concurrency to the language by unifying the object and task concepts into a single one, the active object. The management of active objects is the same as the management of usual C++ passive objects. The difference is that active objects have their own behaviour and that they may either accept method calls or call other objects'methods. They are also capable to await simultaneously several non deterministic events including timers expiration. We show how to systematically model sC++ programs into Promela, a formal language supported by SPIN, a powerful and widely available model-checker. We present a classical example of a concurrent problem and give details about a tool that automatically produces the model from a program, verifies it and allows its debugging.*

1 Introduction

For representing concurrency in software engineering, all the existing analysis methods, including the object oriented paradigm [2] use dataflow diagrams that model data exchange between potentially autonomous items. Applications such as protocols, process control systems, distributed applications, are concurrent by nature and are better modelled and structured if they are modelled with explicit concurrency. The first troubles appear at the implementation of such systems because the available environments are seldom as clear, simple and powerful as the concepts used during analysis. In other words, if concurrency is easily designed with the available analysis tools, it is less easily implemented with the available programming support. This is without mentioning the overhead due to some solutions, for example Unix processes.
Concurrent applications development is also more difficult to master and flaw detection before real exploitation is highly desirable. This is usually achieved using formal methods and abstracting the real system into a mathematical model which is submitted to proofs or model-checking. The development process for modeling and verification may proceed either in a bottom-up or a top-down fashion. In the bottom-up way, the verification phase is done a posteriori. The system is already built and one extracts a model that is submitted to verification[3][7]. If a bug is found, it needs to be fixed and the implementation to be modified. In the top-down approach, the verification is done a priori. The model is elaborated and verified before the implementation. This reduces the impact of possible bugs in the development cycle. These two approaches may combine with refinement techniques, but they usually rely on different formalism: one for the model and one for the implementation, the gap between both being often large. Between these two extremes, one can imagine having the same language for modeling and implementation, the modeling language being strictly included into the implementation language. This would simplify the developers work and require

reduced skills. This supposes that one either creates a special instance of the verification tool for the language used or that bridges with existing tools are built, thus hiding the modeling language. This is the approach that will be presented in this paper. We use sC++, a minimal concurrent extension of C++ [14] as modeling and implementation language, and SPIN as verification tool. We show how to systematically translate a sC++ program into Promela (SPIN's language) and how to directly perform the verification and debugging onto the initial sC++ program using an adapted version of SPIN. We illustrate the approach with the readers and writers problem.

2 sC++

sC++ is an extension of C++ that provides the possibility to develop executable concurrent programs and distributed applications. It is supported by a set of development tools, including a compiler, which is an extension of *gcc*, code generators, and debuggers. It also exists an implementation in sC++ of an Object Request Broker that allows for developing distributed applications compliant with Corba [14].
Regarding the language itself, it is fairly simple for a programmer already knowing C++, since it possesses only few extra keywords with special semantics. sC++ adds concurrency to the language by unifying the object and task concepts into a single one, the active object. Fig.1 shows an example of active class declaration:

```
1. active class C{
2.    private:   void m1(){}
3.               @C(){...}
4.    public:    C(){...}
5.               ~C(){...}
6.               T1 rdv1(T2 t2, T3 &t3){...}
7.               void rdv2(){}
8. };
```

FIGURE 1. Active class declaration

The only syntactic differences with an usual C++ passive object are the extra keyword *active* and the optional method prefixed with @.
The instances of an active class are active objects, whose public methods, when called by other objects, have a special semantics. A call from an object to another active object's method defines a synchronization and a possibly bidirectional data exchange between both objects. It is a rendez-vous in the sense that the caller is blocked if the callee is not ready to accept the call and the callee is blocked if it is ready to accept the call but no call has been initiated. Besides, the rendez-vous is characterized by the atomic execution of the method called, during which bidirectional data exchange may occur thanks to the usual return value and parameters passed by value or reference.
If a method has no parameter and no code, it only has a synchronizing role. As a consequence of atomicity, the synchronizing methods of an active object are always executed in mutual exclusion. The private methods or the public ones called by the object itself have exactly the same semantics as for passive objects.
The optional private method (called *the body*) prefixed with @ has the same name as the class and cannot be called. It defines the behaviour of the active object and allows

an object to accept calls for its methods or to call other objects'methods. For instance the body of Fig.2 contains an infinite loop during which the object successively accepts calls for its methods *rdv1*, calls method *m* of object *o* and accepts calls for *rdv2*. Note for instance that when the object tries to call object *o*'s method *m*, it will block until the rendez-vous is possible; notice also that possible calls from outside to *rdv1* and *rdv2* will be differed until the body accepts them:

```
1. @C(){   for(;;){ accept rdv1; o->m(); accept rdv2; }   };
```

FIGURE 2. An active object's behaviour

Constructors are non-synchronizing public methods, executed before the body is created, but destructors are considered as any other public synchronizing methods. In particular destructors may be explicitly accepted by the object, which is useful for controlling the object's death.

When no body is defined for an active object, an implicit one is defined that accepts calls at any time for all its public methods including the destructor. In other words, an active object without a body behaves as a Hoare monitor [11].

The management of active objects is the same as the management of usual C++ passive objects. They may either be declared directly or through the use of pointers and the *new* statement. They may be destructed explicitly with the *delete* statement or implicitly when the block in which they are contained is exited (Fig.3). They naturally are ready to die (they accept their destructor) if their body finishes.

```
2. {  C c1; C* c2=new C;
3.    c1.rdv2(); c2->rdv2();
4.    delete c2; }/* c1 is implicitly destructed */
```

FIGURE 3. A block with active objects creation and destruction

Here we need to say something about scheduling. Conceptually, scheduling should be very unconstrained and be as if every active object had its own processor. The code of the synchronizing method should be atomically executed only during a rendez-vous, and the control given back to both the caller and the callee. Also, an active object should be thought as running as soon as it is created. In the current mono-processor implementation of the language, after a rendez-vous, the caller gets the processor and only after synchronizations does the control change of active object. This means that an object looping on some pure computations monopolizes the processor, but it is easy to have instead a pre-emptive scheduling based on time sharing. In practice this has a consequence for application startup. If one wants that the application objects begin to execute only when they are all created, one needs either to add special synchronizations for startup or manage that the main creates all the objects before attempting any possible synchronization with some of them, otherwise the control could possibly be transferred to the already created objects; this is done anyhow but only when the main finishes. One also has to be careful of possible situations of deadlocks.

Objects may explicitly suspend their execution until a given date with the statement *waituntil*. When the execution of an object reaches such a statement, its execution is suspended at least until the date mentioned as parameter. The function *now()* provides the current time (Fig.4).

```
1. const int DELAY=...;...;
2. waituntil(now()+DELAY);
```

FIGURE 4. Timeout statement

Active objects are also capable to await simultaneously several non deterministic events including timer expirations. Instead of serializing the calls to its methods, as it appeared on Fig.2, the active object can accept them nondeterministically and at the same time it can attempt to call some other object or execute a timeout with the *waituntil* statement. This is achieved thanks to the *select* statement (Fig.5).

```
1. select{ accept  rdv1;...;
2. ||      o->m();...;
3. ||      accept rdv2;...;
4. ||      waituntil(now()+DELAY);...;
5. }
```

FIGURE 5. Simultaneous events awaiting

Conceptually, the occurrence order of the branches inside the *select* is not meaningful, but in the current implementation, the first enabled branch of the *select* is the one to be executed. Once more, it is easy to change this and execute nondeterninistically a branch among the enabled ones. Each branch of a *select* must begin with an event, namely a synchronizing method call or acceptance or a *waituntil* statement. These events may be conditioned by a boolean expression, called a guard, using the *when* statement as in the following example (Fig.6):

```
1. select{ when(C1) accept  rdv1;...;
2. ||      when(C2) o->m();...;
3. ||      when(C3) waituntil(now()+DELAY);...;
4. }
```

FIGURE 6. Guarded synchronizations

Inheritance may be defined between active classes but we do not detail this point here, see [14]. We conclude with a very simple but complete example that illustrates some aspects of the language. It is a system of three processes accessing a critical section protected by a two entries semaphore.

```
1. #include <scxx.h>
2. active class Semaphore{
3.    int n;
4.    @Semaphore(){
5.       for(;;){   select{ when(n>0) accept P;
6.                  ||      accept V;
7.                  }
8.       }
9.    }
10. public:
11.   Semaphore(int N){n=N;}
12.   void P(){n--;}
13.   void V(){n++;}
14.};
```

```
15.active class User{
16.   Semaphore *s;
17.   int pid;
18.   @User(){ for(;;){s->P(); /* critical section */ s->V(); } }
19. public:
20.   User(int i,Semaphore *sem){pid=i;s=sem; }
21.};
22.main(){
23.   Semaphore s(2);User u1(1,&s);User u2(2,&s);User u3(3,&s);}
```

FIGURE 7. Mutual exclusion with a semaphore in sC++

3 Promela/SPIN

SPIN [12] is a powerful verification system that supports the design, verification and debugging of asynchronous process systems. SPIN focuses on proving correctness of process interactions that can be modelled in Promela, the language of SPIN, with rendez-vous primitives, asynchronous message passing through buffered channels and shared variables. Systems specification may be expressed thanks to boolean assertions or linear temporal logic (LTL) formulae. The model-checker may address real-sized problems of several millions states, thanks to several optimizations of depth-first search; first, memory management techniques based on state compression and bit-state-hashing, second, partial order reductions. When errors are found, traces are produced that are used to guide a graphical debugger, quite useful to identify the flaws.

Fig.9 shows problem of Fig.7, modelled in Promela. Active objects are declared with the *proctype* clause and may have parameters (Fig.9 lines 4,12). They are instantiated with the *run* statement (line 17). We have modelled the synchronizing methods *P* and *V* with synchronous channels (lines 1-2). For the moment forget that *P* is an array of three channels. Since no data is exchanged during the synchronization, these channels should convey no value, but one value at least is required, so we exchange a bit that represent the synchronization event. The nondeterministic choice and the loop around is done with the *do :: od* construct. Each branch corresponds to a *select*'s branch of the sC++ code, which is limited to the rendez-vous on *P* and *V*. The rendez-vous are *atomic* (lines 5 and 8) and are composed of the related synchronisation (lines 6 and 9) and method code execution (lines 7 and 10). By convention, we code a call by an emission on the related channel and an acceptance by a reception. We now have to explain why *P* is an array. In the sC++ program of Fig.7, the execution of method *P* of object *Semaphore* is guarded by the condition *(n>0)*. Unfortunately, there exists no dedicated construct in Promela for modeling guards – even the *atomic* statement cannot solve the problem without introducing extra deadlocks – but this can be done using a trick. The boolean type and associated constants exist but as in C, they are related to the integer type. In other words, *true* is 1, and *false* is 0.

```
1. chan R[3] = [0] of {...};
2. sender: R[cond1]!msg;...
3. receiver: R[2-(cond2)]?msg;...
```

FIGURE 8. Guarded Rendez-vous

For modeling a guarded rendez-vous *R (Fig.8)*, we declare two dummy channels *R[0]* and *R[2]* that are used to deflect handshake attempts that should fail. The handshake can only successfully complete on *R[1]* if both *cond1* at the sender side and *cond2* at the receiver side hold.
On our example, only the reception, i.e. the *accept* in the sC++ program, is guarded (Fig.9, line 6), the emission being guarded by *true* (line 13).
The global variable *in_CS* (lines 3, 14) is used for the verification of the assertion (line 14) that at most *N* (here 2) users may have the critical section at the same time, if the semaphore was initialized with *N*.

```
1. chan P[3] = [0] of {bit};
2. chan V    = [0] of {bit};
3. int in_CS=0;
4. proctype Semaphore(int n){
5.    do :: atomic{
6.          P[2-(n>0)]?true;
7.          n-- }
8.       :: atomic{
9.          V?true;
10.         n++}
11.   od }
12.proctype User(int pid){
13.   do :: P[true]!true;
14.         in_CS++; assert(in_CS <= 2); in_CS--;
15.         V!true
16.   od }
17.init{ run Semaphore(2); run User(1);run User(2);run User(3) }
```

FIGURE 9. Mutual exclusion with a semaphore in SPIN

4 Modeling of sC++ Applications

We now show how to model sC++ programs in Promela in a systematic way. Obviously we cannot treat sC++ in its totality, this would suppose treating also all C++, which would be very heavy and certainly lead to inefficient models thus making verification of any kind impracticable because of state explosion. We are first interested in checking somewhat abstract designs and thus we do not need all the power of C++. In particular we only consider active classes. Passive classes, inheritance and generics are only structuring facilities that are not essential for our purpose. We also only consider basic types, such as integers and arrays but no structure, no type definition and no pointer, excepts pointers of active objects that are necessary to pass references between active objects for them to communicate. Some instructions are necessary (*for*, *if*), if they are not (*while*, *switch*, *goto*) we drop them. We also forget compilation directives and suppose that all the methods are defined inside the class. For similar reasons, we only take into account the useful subset of Promela. Besides syntactic limitations evoked above, we still restrict the scope of our translation to the salient points. In particular we only consider direct object creation and thus forget the *new* and *delete* statements. We consider only explicit bodies but implicit constructors and destructors for active classes.

For synchronizations, we only treat methods with no returned type and no reference parameters. Nevertheless we'll treat an explicit constructor with parameters, an implicit body and a complete rendez-vous with bidirectional data exchange on an example.

We formalize the systematic modeling of the considered subset of sC++ in Promela, following the usual practice of defining a denotational semantics[15]. This semantics is defined thanks to a set of semantic functions from abstract syntactic constructions of the source language sC++ to semantics values that are in fact abstract syntactic constructions of the target language Promela. As parameter and result of the semantic functions we also sometimes need particular structures, called environments, that memorize the context of evaluation of a particular construction.

Fig. 10 shows the abstract syntactic domains of sC++ with abstract syntactic variables in font `courier` that are to be used below for the semantics definition. Fig. 11 shows sC++ related production rules, with an usual BNF syntax. The keywords of the language are in bold, ::= is the derivation symbol, | is the alternative, + and * when used as exponents means one, resp. zero, or more repetitions, the square brackets indicate optional constructions, the parenthesizes factorisation and ε the empty string.

P ∈ $Program_s$	X ∈ $Main_s$	H ∈ $Branch_s$	I ∈ $Identifier_s$
D ∈ $Declaration_s$	K ∈ $Block_s$	S ∈ $Synchro_s$	N ∈ $Natural_s$
A ∈ $ActiveClass_s$	B ∈ $Body_s$	E ∈ $Expression_s$	
M ∈ $Method_s$	C ∈ $Command_s$	T ∈ $Type_s$	

FIGURE 10. sC++ abstract syntactic domains

$Program_s$ ::= $Declaration_s^*$ $ActiveClass_s^*$ $Main_s$
$Declaration_s$ ::= $Type_s$ [*]$Identifier_s$[[$Natural_s$]]
$ActiveClass_s$ ::= **active class** $Identifier_s$ {$Declaration_s^*$ $Method_s^*$ $Body_s$ }
$Method_s$ ::= [$Type_s$] [~] $Identifier_s$ (($Type_s$ $Identifier_s$)*) $Block_s$
$Main_s$ ::= [**void**] **main()** $Block_s$
$Block_s$ ::= {$Declaration_s^*$ $Command_s^+$ }
$Body_s$ ::= @ $Identifier_s$ () $Block_s$
$Command_s$ ::= $Identifier_s$ = $Expression_s$ | $Identifier_s$++ | $Identifier_s$-- |
if $Expression_s$ $Command_s^+$ | **if** $Expression_s$ $Command_s^+$ **else** $Command_s^+$ |
for ($Command_s$; $Expression_s$; $Command_s$) $Block_s$ |
return $Expression_s$ | **assert** $Expression_s$ | **printf** $String_s$ |
select{ $Branch_s^+$ **}** | $Synchro_s$
$Branch_s$ ::= [**when**($Expression_s$)] $Synchro_s$ $Command_s^*$
$Synchro_s$ ::= **accept** $Identifier_s$ | $Identifier_s$ -> $Identifier_s$($Expression_s^*$) |
waituntil($Expression_s$)
$Type_s$ :: = **int** | **void** | $Identifier_s$
$Expression_s$ ::= ...

FIGURE 11. sC++ abstract production rules

$Program_p$	$Init_p$	$Type_p$	$Natural_p$
$Declaration_p$	$Block_p$	$Expression_p$	$String_p$
$Proctype_p$	$Command_p$	$Identifier_p$	

FIGURE 12. Promela abstract syntactic domains

$Program_p$ ::= $Typedef^*$ $Declaration_p^*$ $Proctype_p^*$ $Init_p$
Typedef ::= **#define** $Identifier_p$ $Expression_p$ | **typedef** $Identifier_p$ {$Declaration_p^+$}
$Declaration_p$ ::= $Type_p$ $Identifier_p$[[$Natural_p$]] [= $Expression_p$] |
chan $Identifier_p$[[$Natural_p$]] = [$Natural_p$] **of** {$Type_p^+$}
$Proctype_p$ ::= **proctype** $Identifier_p$ (($Type_p$ $Identifier_p$)*) $Block_p$
$Init_p$ ::= **init** $Block_p$
$Block_p$ ::= { $Declaration_p^*$ $Command_p^+$ }
$Command_p$ ::= $Identifier_p$: $Command_p$ |
$Identifier_p$ = $Expression_p$ | $Identifier_p$++ | $Identifier_p$-- |
goto $Identifier_p$ | **if** (::$Command_p$)$^+$ **fi** | **do** (::$Command_p$)$^+$ **od** |
$Expression_p$ **!** $Expression_p^+$ | $Expression_p$ **?** $Identifier_p^+$ |
assert $Expression_p$ | **printf** $String_p$ | **skip** | **break** |
atomic{ $Command_p^+$} | **run** $Expression_p$($Expression_p$*)
$Type_p$ :: = **byte** | **int** | **bit** | $Identifier_p$
$Expression_p$ ::= ...

FIGURE 13. Promela abstract production rules

The result of semantic functions are on the one hand values of Promela abstract syntactic domains (Fig.12-13) and values of semantic domains, called environments (Fig.14) defined as abstract data types with the usual manipulations operations (application, projections,...) that are not detailed here.
The environment is composed of four structures. *Active*, given a type name indicates if it corresponds to an active class or not. *Class*, given an object identifier and the identifier of the class in which the object is declared gives the object's class name. *Param* gives for a given method of a given class, the list (possibly empty) of the parameter identifiers and eventually, *Code* gives the executable code of such a method (the instructions but not the possible declarations).

Environment = Active × Class × Params × Code
active, class, params, code: *projections* (): *application*
Active = $Type_s \rightarrow$ Boolean
Class = $Identifier_s \times Identifier_s \rightarrow Identifier_s$
Params = $Identifier_s \times Identifier_s \rightarrow Identifier_s$*
Code = $Identifier_s \times Identifier_s \rightarrow Command_s$*

FIGURE 14. Semantic domains

The semantic function *model* (Fig.15) defines the modeling of a sC++ program into Promela. It is defined thanks to auxiliary functions. The function *build* builds the environment of a sC++ program. The function *creat* creates preliminary definitions for objects; for passive objects it is their declaration, for active objects it is in particular the structures used for synchronizations. The function *trans* is the core of the translation, it does the actual creation of active objects and translate the instructions. The function *destr* generates the instructions that calls the active object destruction, which is implicit each time a block is finished for the active objects created directly (without *new*) in the block. These functions have a sC++ program fragment as parameter and possibly the program environment and the identifier of the class inside which the code is currently analysed. We use the traditional special brackets || || in denotational definitions

(introduced by the special equal ≡) to separate the syntactic and the semantic worlds. For instance, *trans*||P||$_{E,C}$ ≡ ... represents trans(P,E,C) ≡ ...

model : $Program_s \rightarrow Program_p$
build : $Program_s \rightarrow Environment$
creat : $Program_s \times Environment \rightarrow Program_p$
trans,destr : $Program_s \times Environment \times Identifier_s \rightarrow Program_p$

FIGURE 15. Semantic functions

4.1 Semantics

For simplicity, we also consider that methods have different names in all the classes of the program and that method parameters and local variables have different names inside a given class. Also, local method parameters cannot be active objects. Our semantics is easily extended to take into account these restrictions and some optimizations such as rendez-vous with no data exchange, but this implies make the presentation heavier, in particular by the introduction of extra structures. We also omit the definition of *build* and the semantics of expressions or things that are rather «stable» under the translation.

Semantic equations

First a program is modelled in its environment. In order to have bounded dynamic object creation to keep the space finite and tractable, for each class we will have a maximum instance number. First are declared the possible global objects and the structures necessary for the active classes. Then the active classes are translated. Finally the main program is translated but preceded by the actual active object creations and followed by their destruction in reverse order.

model||P|| ≡ *trans*||P||$_{E,prog}$ where E=*build*||P|| and *build*||P|| ≡ ...
trans||D_1...D_d A_1...A_a X||$_{E,C}$ ≡
#define NBMAXINST 8
creat||D_1||$_E$ ... *creat*||D_d||$_E$ *creat*||A_1||$_E$ ... *creat*||A_a||$_E$
trans||A_1||$_{E,C}$ ... *trans*||A_a||$_{E,C}$
init{ int i
trans||D_1||$_{E,C}$ ... *trans*||D_d||$_{E,C}$
trans||X||$_{E,C}$
destr||D_d||$_{E,C}$ ... *dest*||D_1||$_{E,C}$ **}**

The object declaration is the usual one for passive objects and a reference (byte) for active objects which is an index in a array of all the instances of the class. A pointer to an object is exactly the same thing (we show here only array declarations, scalar definitions are easily obtained removing the indices). The declaration for an active class concern the channels, grouped in a *typedef*, used for implementing the possibly guarded rendez-vous on synchronizing methods, including the destructor (*dest*). We need two channels (*_c*, *_r*) to implement a synchronizing method call which needs to be atomic. If the method has no parameter we need to exchange a minimal value, for instance *true* of type *bit*. The class is represented by array of instances (*inst*) and an instance number (*instnb*) which is to be incremented at each object creation.

creat||T I [N]||$_E$ ≡ if active(E)(T) **byte** I[N] else *creat*||T||$_E$ I[N]
creat||T *I [N]||$_E$ ≡ if active(E)(T) **byte** I else error

creat||**active class** I {D_1...D_d M_1...M_m B}||$_E$ ≡
  typedef I{ *creat*||M_1||$_E$ ... *creat*||M_d||$_E$
    chan dest_c[3] = [0] of {bit}
    chan dest_r = [0] of {bit} }
  I I_**inst[NBMAXINST]**
  byte I_**instnb=0**
creat||**void** I (T_1 I_1...T_n I_n) K||$_E$ ≡
  chan I_**c[3] = [0] of {bit** creat||T_1||$_E$ ... creat||T_n||$_E$**}**
  chan I_**r = [0] of {bit}**

The actual active object creation, given here only for arrays, consists in incrementing the instance number of the class, store it in the instance reference array and finally *run* the process that implements the class body (*proc*) with its instance reference as parameter. Note that nothing needs to be done for pointers.

trans||T I [N]||$_{E,C}$ ≡
  if ¬ active(E)(T) ε else
    i=0
    do :: (i<=N-1)
      atomic{ class(E)(I,C)**[i]**=class(E)(I,C)**_instnb**
        class(E)(I,C)**_instnb++**
        run class(E)(I,C)**_proc**(class(E)(I,C)**[i])}**
      i++
    :: else break
    od
trans||T *I [N]||$_{E,C}$ ≡ ε

An active class is implemented as a Promela *proctype*. It corresponds to a collection of instances that are distinguished owing to a byte parameter (*this*). The class variables are declared and run (if active), declarations for each method are done (parameters and local variables), finally the body of the active class is translated. The code of the class is achieved with an acceptance of the class destructor, the execution of which ends the class instance execution. This *accept* statement will be executed if the object naturally reaches its body end. The object can also explicitly accept its destruction, in which case a jump will be done to an ultimate *skip* statement labelled with *end*.

trans||**active class** I {D_1...D_d M_1...M_m B}||$_{E,C}$ ≡
  proctype I_**proc(byte this){**
    creat||D_1||$_E$ ... *creat*||D_d||$_E$
    trans||D_1||$_{E,I}$ ... *trans*||D_d||$_{E,I}$
    trans||M_1||$_{E,I}$ ... *trans*||M_m||$_{E,I}$
    trans||B||$_{E,I}$
    trans||**accept** ~I||$_{E,I}$
  end_I: skip}
trans||**void** I(D_1...D_n){D_{n+1}...D_m C_1...C_c}||$_{E,C}$ ≡
  creat||D_1||$_E$ ... *creat*||D_n||$_E$ *creat*||D_{n+1}||$_E$ ... *creat*||D_m||$_E$

The translation of a body is its block's which is the declaration of local variables, the instructions translation and the possible destruction of active objects in reverse order of

the creation.The translation of the *main* is the translation of its block.

trans||@ I() K||$_{E,C}$ ≡ *trans*||K||$_{E,C}$

trans||{D_1...D_d C_1...C_c}||$_{E,C}$ ≡
  creat||D_1||$_E$ ... *creat*||D_d||$_E$
  trans||D_1||$_{E,C}$ ... *trans*||D_d||$_{E,C}$
  trans||C_1||$_{E,C}$ ... *trans*||C_c||$_{E,C}$
  destr||D_d||$_{E,C}$ ... *destr*||D_1||$_{E,C}$

trans||**void main(){**K**}**||$_{E,C}$ ≡ *trans*||K||$_{E,main}$

We now come to the instructions. The *select* statement naturally becomes the Promela *if :: fi*. The guarded synchronizations are the most interesting (other ones are obtained setting the guard to *true*). By convention, the calls are implemented as an emission on the first rendez-vous channel associated to the method (*_c*) and the acceptances are receptions. For the guard translation, we use the trick mentioned above. The end of the rendez-vous is implemented by a synchronisation on the second channel (*_r*). For the acceptance, the method code is inserted between both rendez-vous. Note the usage of the environment to retrieve an object's class and a method's parameters and code. Note also that an acceptance for the class destructor is closed by a jump to the *end* label.

trans||**select{**B_1...B_b**}**||$_{E,C}$ ≡ **if ::** *trans*||B_1||$_{E,C}$ ... **::** *trans*||B_b||$_{E,C}$ **fi**

trans||S C_1...C_c||$_{E,C}$ ≡ *trans*||S||$_{E,C}$ *trans*||C_1||$_{E,C}$ ... *trans*||C_c||$_{E,C}$

trans||**when(**E_0**)** I_1**->**I_2(E_1...E_n)||$_{E,C}$ ≡
  class(E)(I_1,C) **_inst[**I_1**].**I_2**_c[***trans*||E_0||$_{E,C}$**]!true** *trans*||E_1||$_{E,C}$... *trans*||E_n||$_{E,C}$
  class(E)(I_1,C) **_inst[**I_1**].**I_2**_r?true**

trans||**when(**E**) accept** I||$_{E,C}$ ≡
  C**_inst[this].**I**_c[2-***trans*||E||$_{E,C}$**]?true** params(E)(I,C)
  trans||code(E)(I,C) ||$_{E,C}$
  C**_inst[this].**I**_r!true**

trans||**when(**E**) accept ~**I||$_{E,C}$ ≡
  C**_inst[this].dest_c[2-***trans*||E||$_{E,C}$**]?true**
  trans||code(E)(~I,C) ||$_{E,C}$
  C**_inst[this].dest_r!true**
  goto end_C

A guarded timeout is translated as the guard itself. A *waituntil* without guard is just a *skip* statement, since our models are only qualitative.

trans||**when(**E_1**) waituntil(**E_2**)** ||$_{E,C}$ ≡ *trans*||E_1||$_{E,C}$

trans||**waituntil(**E**)**||$_{E,C}$ ≡ **skip**

Other instructions translate naturally. For instance *if* statements and loops. We also give some optimizations for never ending loops with a *select* as outmost statement.

trans||**assert(**E**)**||$_{E,C}$ ≡ **assert(***trans*||E||$_{E,C}$**)**

trans||**if** E C_1 **else** C_2||$_{E,C}$ ≡ **if ::** *trans*||E||$_{E,C}$ *trans*||C_1||$_{E,C}$ **::else** *trans*||C_2||$_{E,C}$ **fi**

trans||**for(**C_1;E;C_2**)**B||$_{E,C}$ ≡
  trans||C_1||$_{E,C}$
  do :: *trans*||E||$_{E,C}$ *trans*||B||$_{E,C}$ *trans*||C_2||$_{E,C}$ **:: else break od**

trans||**for(;;)select{**B_1...B_b**}**||$_{E,C}$ ≡
  do :: *trans*||B_1||$_{E,C}$ ... **::** *trans*||B_b||$_{E,C}$ **od**

Finally, destruction of active objects are the counterpart of their creation with rendez-vous on the channels associated to the class destructor.

destr||T I [N]||$_{E,C}$ ≡
  if ¬ active(E)(T) ε else
    i=N-1
    do :: (i>=0)
      class(E)(I,C)**_inst[I(i)].dest_c[true]!true**
      class(E)(I,C)**_inst[I(i)].dest_r?true**
      i--
    :: else break
    od
destr||T *I||$_{E,C}$ ≡ ε

We finish this section showing the modeling of a simple class with a constructor with parameters, a method with bidirectional data exchange and an implicit body (Fig.16).

```
1. active class C{
2.    int n;
3.    C(int N){n=N;}
4.    int M(int v1, int &v2){ v2=v2+n*v1; return v1; }
5. };
6. void main(){
7.    int j;
8.    j=0;
9.    C c(10);
10.   c.M(3,j);}
```

FIGURE 16. Constructor with parameters, complete RDV and implicit body

For the method *M*, notice that the initial value of the reference parameter is sent at the begining of the call (Fig.17, lines 27 and 12) and received at its end (lines 28 and 14).

```
1. #define NBMAXINST 3
2. byte c;
3. typedef C{ chan M_c[3]     = [0] of {bit,int,int}
4.            chan M_r        = [0] of {bit,int,int}
5.            chan dest_c[3]  = [0] of {bit}
6.            chan dest_r     = [0] of {bit} }
7. C C_inst[NBMAXINST]
8. byte C_instnb=0
9. proctype C_proc(byte this;int N){
10.  byte n; int v1;int v2;
11.  n=N;
12.  do :: C_inst[this].M_c[2-(true)]?true,v1,v2;
13.        v2=v2+n*v1;
14.        C_inst[this].M_r!true,v1,v2;
15.     :: C_inst[this].dest_c[2-(true)]?true;
16.        C_inst[this].dest_r!true;
17.        goto end;
18.  od;
19.end_C:skip}
```

```
20.init{int i;
21.     int M_ret;
22.     int j;
23.     atomic {
24.        c = C_instnb;
25.        C_instnb++ ;
26.        run C_proc(c,10)};
27.     j=0;
28.     C_inst[c].M_c[true]!true,3,j;
29.     C_inst[c].M_r?true,M_ret,j;
30.     C_inst[c].dest_c[true]!true;
31.     C_inst[c].dest_r?true;}
```

FIGURE 17. Modelling in Promela

4.2 The readers writers example

We now illustrate the application of our translation semantics on the classical mutual exclusion example of the readers writers example. Readers and writers processes attempt to access a critical resource with, for instance, the following rules. As many readers may read the data at the same time, provided no writer is writing and only one writer may access the data if no reader is reading. For the implementation of this example, we declare three classes, *SharedData*, *Reader* and *Writer*, one instance for the first class and two instances for each of the following grouped in an array. We also need two counters *nbwriters* and *nbreaders*, for the readers and writers for verifying by assertions, that the rules will hold.

```
1. int nbwriters;int nbreaders;
2. SharedData shData; Reader reader[2]; Writer writer[2];
```

The class *SharedData* manages the critical resource thanks to four methods, two for requesting the resource in read or write mode and two for releasing it (we only show what is related to the reading, what is related to writing is similar).

```
3. active class SharedData {
4.    int readersCounter;int writersCounter;
5.    void BeginRead() { nbreaders++; }
6.    void EndRead() { nbreaders--; } ...
7.    @SharedData() {
8.       readersCounter=0;writersCounter=0;
9.       for(;;){
10.         select{ accept EndRead; readersCounter--;
11.         ||      when(writersCounter==0)
12.                    accept BeginRead;readersCounter++;
13.         || ...
14.         ||      when((readersCounter==0)&&
15.                         (writersCounter==0))
16.                    accept ~SharedData;
17.         }
18.      }
19.   }};
```

The *Reader* and *Writer* classes possess a reference on a *SharedData* object, initialized

with the method *minit*, and request the critical data through it, conforming to a simple and natural protocol (we only show the *Reader* class code).

```
20.active class Reader {
21.   SharedData *sData;
22.   void minit (SharedData *SD) { sData = SD; }
23.   @Reader() {
24.      accept minit;
25.      for (;;) {
26.         select{ sData->BeginRead();
27.                 assert(nbwriters==0);
28.                 sData->EndRead();
29.         ||      accept ~Reader;
30.         }
31.      }
32.   }
33.};
```

Finally the main function initializes the counters and the active objects.

```
34.void main() {
35.   int p;
36.   nbwriters=0;nbreaders=0;
37.   for (p=0; p<=1; p++){
38.      reader[p]->minit(shData);writer[p]->minit(shData);
39.   }
40.}
```

We give the translation of some of the interesting part of this program. First, the declarations of passive and active objects:

```
1. #define NBMAXINST 3
2. int nbwriters;int nbreaders;
3. byte shData; byte reader[2];byte writer[2];
```

Then, the channels, instance reference array and counter for class *SharedData*:

```
4. typedef SharedData{  chan BeginRead_c[3]  = [0] of {bit}
5.                      chan BeginRead_r     = [0] of {bit}
6.                      chan EndRead_c[3]    = [0] of {bit}
7.                      chan EndRead_r       = [0] of {bit}
8.                      ...
9.                      chan dest_c[3]       = [0] of {bit}
10.                     chan dest_r          = [0] of {bit} }
11.SharedData SharedData_inst[NBMAXINST]
12.byte SharedData_instnb=0
```

The declarations are analogous for class *Reader* (similar for *Writer*), note data passed through method *minit*:

```
13.typedef Reader{ chan minit_c[3]    = [0] of {bit,int}
14.                chan minit_r       = [0] of {bit}
15.                chan dest_c[3]     = [0] of {bit}
16.                chan dest_r        = [0] of {bit} }
17.Reader Reader_inst[NBMAXINST]
18.byte Reader_instnb=0
```

For the process implementing *SharedData* behaviour, we only show the translation of acceptance of *EndRead*, a method with some code and the acceptance of the destructor:

```
19.proctype SharedData_proc(byte this){
20.   int readersCounter;int writersCounter;
21.   readersCounter=0;writersCounter=0;
22.   do :: SharedData_inst[this].EndRead_c[2-(true)]?true;
23.         nbreaders--;
24.         SharedData_inst[this].EndRead_r!true;
25.         readersCounter--
26.      ...
27.      :: SharedData_inst[this].dest_c[2-((readersCounter==0)
28.                                  &&(writersCounter==0))]?true;
29.         SharedData_inst[this].dest_r!true;
30.         goto end_SharedData;
31.   od;
32.end_SharedData:skip}
```

We now show the creation of readers, their initialization with the call to *minit* method and their destruction:

```
33.init{int i; ...
34.      i=0;
35.      do :: (i<=2-1);
36.            atomic {
37.               reader[i] = Reader_instnb;
38.               Reader_instnb++ ;
39.               run Reader_proc(reader[i]) };
40.            i++
41.         :: else;break
42.      od;
43.      ...
44.      int p;
45.      nbwriters=0;nbreaders=0;
46.      p=0;
47.      do :: (p<=1);
48.            Reader_inst[reader[p]].minit_c[true]!true,shData;
49.            Reader_inst[reader[p]].minit_r?true;
50.            Writer_inst[writer[p]].minit_c[true]!true,shData;
51.            Writer_inst[writer[p]].minit_r?true;
52.            p++
53.         :: else;break
54.      od;
55.      ...
56.      i=2-1;
57.      do :: (i>=0);
58.            Reader_inst[reader[i]].dest_c[true]!true;
59.            Reader_inst[reader[i]].dest_r?true;
60.            i--
61.         :: else;break
62.      od;
63.      ...}
```

4.3 Implementation

The semantics presented above has been implemented in Prolog[1]. Apart from translating some sC++ code into Promela, the translator generates a correspondence table that is used for debugging. This table implements a function that relates the line number of the produced code to the line number of the source code. This table is used by a modified version of XSPIN[6] (the graphical interface of SPIN). This allows to do simulations and debugging directly on the sC++ code. When loaded a sC++ program is translated in two files. A first one contains the Promela code and this is the actual file that is used to perform the simulation and verification. The second one contains the table which is used each time a display operation is done in the simulator.

5 Conclusion

This paper presented the key points of the formalization of modeling sC++ programs into Promela and illustrated it through a simple but complete example.
The approach this formalization supports has several advantages. The language itself is to some extent more supple that other similar languages, for it unifies the task and object concept and it possesses symetrical select statements (with acceptances and calls). Beside using a single language for both implementation and verification, once it is clear that one focuses on process interactions and works on some abstraction of the problem under study, the full power of the language is not needed and thus the approach may be fully automatized. We presented only part of the translation without optimizations, but the tool could implement a more complete translations with lots of optimizations. Nevertheless, the produced models are easily readable and tunable if necessary.

If we use the approach in a top-down way, from the beginning we build sC++ programs that are verifiable and already executable. Of course these programs cannot contain all the details, they are a kind of abstraction of the final system, otherwise their verification leads to intractable state explosion. For getting a real system, we only need to add details to it. Thus, the task of the developer is reduced. Obviously, at a given point the program gets too complicated and cannot be verified any more with our method because of state explosion. At this point we can use other tools[13] developed around sC++ that rely on random walk techniques[16].

If we use the approach in a bottom-up way, we always need to make an abstraction effort, but this is partially done by simply deleting lines in the original code, all the tedious work of representing synchronizations through channels and managing the various variables is taken in charge by the tool. Also the intricate semantics of objects death is done automatically, but could be present as an option if objects death is not the key point of verification.

Naturally, for being really useful in practice on real applications, the tool should take into account applications scattered in several files, and treat methods defined outside their classes. This could be included in future extensions of the tool. However, taking into account more sophisticated data types (usual passive C++ classes) and inheritance is a bit in contradiction with the scope of this work, mostly focused on interactions between processes.

This approach has been inspired of bottom-up[3][5][7]as well as top-down[4][8][9][10] experiences. It may easily be adapted for other input languages such as Ada, or Java.

6 References

1. N. Begat, Réalisation d'un interface entre sC++ et SPIN, IIE, Evry, France, engi neer report, in french, 1996.
2. G. Booch, J. Rumbaugh, I. Jacobson The UML User Guide, Add. Wesley, 1998.
3. T. Cattel, Modeling and Verification of a Multiprocessor Realtime OS Kernel, FORTE'94, Berne, 1994.
4. T. Cattel, Using Concurrency and Formal Methods for the Design of Safe Process Control. PDSE/ICSE-18 Workshop, Berlin, March 1996
5. J. Daems, Aide à l'engagement de centrales d'alarmes, EPFL engineer report, in french, 1995
6. G. Duval, Réalisation d'un debugger pour SPIN, report of Maîtrise d'informatique, Université de Besançon, France, in french, 1994.
7. G. Duval, J. Julliand, Modeling and Verification of the RUBIS Micro-Kernel with SPIN, Proc. First SPIN Workshop, INRS Quebec, Canada, Oct. 1995.
8. G. Duval, T. Cattel, Specifying and Verifying the Steam Boiler Controler with SPIN. Springer-Verlag, LNCS vol. 1165, 1996.
9. G. Duval, T. Cattel, From Architecture down to Implementation of Safe Process Control Applications. HICSS-30 Wailea, Maui, Hawaii, U.S.A. 1997.
10. G. Duval, Specification and Verification of an Object Request Broker, submitted to ICSE, Kyoto, Japan, 1998.
11. C.A.R. Hoare, Monitors: An Operating System Structuring Concept, Communications of the ACM, 12(10), October 1974.
12. G.J. Holzmann, The Model Checker SPIN, IEEE Transactions on software engineering, vol. 23, no.5, May 1997.
13. J. Madsen, Validation and Testing of sC++ applications, IEEE conference Engineering of Computer Based Systems, Monterey, California, U.S.A. 1997.
14. C. Petitpierre, sC++, A Language Adapted to Interactive Applications, accepted for IEEE Computer Journal, March 1998. http://diwww.epfl.ch/w3lti/
15. K. Slonneger, B.L. Kurtz, Formal Syntax and Semantics of Programming Languages, Addison-Wesley, 1995.
16. C. West, Protocol validation by random state exploration. PSTV, VI, 1987.