Derivation of SystemC code from abstract system models
Dominique Cansell, Jean-François Culat, Dominique Méry, Cyril Proch

To cite this version:

HAL Id: inria-00107780
https://inria.hal.science/inria-00107780
Submitted on 19 Oct 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Derivation of SystemC code from abstract system models

Dominique Cansell  Jean-François Culat
Université de Metz & LORIA LORIA
Dominique Méry, Cyril Proch
Université Henri Poincaré Nancy 1 & LORIA

Abstract

Formal methods provide techniques and tools for constructing mathematical models of software systems and more generally of hardware/software systems. Mathematical models are incrementally developed and provide informations validated by a proof assistant; the development starts by a simple model and the process enriches current models by adding details from the requirements; the process leads to a sequence of models related by the refinement relation and a sequence of directed acyclic graphs stating dependency among parameters to be computed by the future monitoring tool. Those DAGs together with the events of models, can be used to generate an architecture of the final system and a code for computing each parameter. The resulting architecture and the resulting code are produced from formally validated components; moreover, DAGs help in the organization of computations among parameters. Our work shows how formal models and DAGs can be used to build a system on chip using System C.

1 Introduction

1.1 Design of System On Chip

Systems on Chip, or shortly SoCs, and SoC architectures combine problems of specification, modeling, safety, quality and structuring mechanisms; we present results of a research activity that we have carried out in collaboration with industrial partners and lead us to a design methodology for constructing models of the system and for providing formally justified hints on the future architectural choices. Our works provide also a resulting mathematical model of a tool, which will be in fine implemented on a chip. Our methodology based on the B event-based method [CM02, CTB+03, CM03] integrates the incremental development of formal models using a theorem prover to validate each step of development called refinement. A (mathematical) model is simply defined as a reactive system with invariant and safety properties and it expresses requirements of the target SoC, together with hints on the architecture. Our case study is a monitoring tool for measurement in Digital Video Broadcasting Television (DVB-T) and problems are related to the number of computations and real-time constraints. The implementation of this tool is driven by the hierarchy derived from invariants of models.

1.2 Overview of B

Classical B is a state-based method developed by Abrial for specifying, designing and coding software systems. It is based on Zermelo-Fraenkel set theory with the axiom of choice. Sets are used for data modeling, “Generalized Substitutions” are used to describe state modifications, the refinement calculus is used to relate models at varying levels of abstraction, and there are structuring mechanisms (machine, refinement, implementation) which are used in the organization of a development. The first version of the B method is extensively described in the B-Book [Abr96]. It is supported by the Atelier B tool [Cle04].
Central to the classical B approach is the idea of a software operation which will perform according to a given specification if called within a given pre-condition. Subsequent to the formulation of the classical approach, Abrial and others have developed a more general approach in which the notion of “event” is fundamental. An event has a firing condition (a guard) as opposed to a pre-condition. It may fire when its guard is true. Event-based models have proved useful in requirement analysis, modeling distributed systems and in the discovery/design of both distributed and sequential programming algorithms.

After extensive experience with B, current work by Abrial is proposing the formulation of a second version of the method[AM98, Abr03]. This distills experience gained with the event based approach and provides a general framework for the development of “discrete systems”. Although the method is generalized, mathematical foundations of both versions of the method are the same.

1.3 Proof-based Development

Proof-based development methods [Bac79, Abr96, Mor90] integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the system under development. Details are gradually added to this first model by building a sequence of more concrete ones. The relationship between two successive models in this sequence is that of refinement [Bac79, Abr96, CM88, BvW98]. The essence of the refinement relationship is that it preserves already proved system properties including safety properties and termination.

A development gives rise to a number of, so-called, proof obligations, which guarantee its correctness. Such proof obligations are discharged by the proof tool using automatic and interactive proof procedures supported by a proof engine [Cle04].

At the most abstract level it is obligatory to describe the static properties of a model’s data by means of an “invariant” predicate. This gives rise to proof obligations relating to the consistency of the model. They are required to ensure that data properties which are claimed to be invariant are preserved by the events or operations of the model. Each refinement step is associated with a further invariant which relates the data of the more concrete model to that of the abstract model and states any additional invariant properties of the (possibly richer) concrete data model. These invariants, so-called gluing invariants are used in the formulation of the refinement proof obligations.

The goal of a B development is to obtain a proved model. Since the development process leads to a large number of proof obligations, the mastering of proof complexity is a crucial issue. Even if a proof tool is available, its effective power is limited by classical results over logical theories and we must distribute the complexity of proofs over the components of the current development, e.g. by refinement. Refinement has the potential to decrease the complexity of the proof process whilst allowing for tracability of requirements.

B Models rarely need to make assumptions about the size of a system being modeled, e.g. the number of nodes in a network. This is in contrast to model checking approaches [CGP00]. The price to pay is to face possibly complex mathematical theories and difficult proofs. The re-use of developed models and the structuring mechanisms available in B help in decreasing the complexity. Where B has been exercised on known difficult problems, the result has often been a simpler proof development than has been achieved by users of other more monolithic techniques.

1.4 Related techniques

The B method is a state-based method integrating set theory, predicate calculus and generalized substitution language; it provides a way to develop incrementally models of systems by refinement and by proofs of conditions of verification called proof obligations. There are methodologies for developing and validating embedded systems as for instance the POLIS approach based on the POLIS environment [BCG00]. POLIS integrates techniques of simulation and model checking in the development of embedded systems. The POLIS system is a co-design environment for embedded systems based on a formal model of computation namely the Co-design Finite State Machines model of the CFSM model for short. The main idea is to integrate the translation of a
high level language (Esterel, for instance) into the CFSM language, the formal verification and the
synthesis of systems stated in the CFSM language by translation into the computation model of
existing verification tools, the mean to co-simulate systems and the partitioning and architecture
selection.

Work on action systems are very close to our framework; action systems are a general formalism
equipped with the refinement relation. The main difference relies upon the way we are using
the refinement. In [HS98], the specification is used for modeling the case study but no real
methodology is provided for helping in writing the specification. We obtain a formal model of the
system but we have been able to trace the construction with respect to the requirements. The
final step of our approach is to provide a way to produce code. The formal model allows us to
structure the final system into hardware and software parts.

With our partners (Thales B&M, TDF C2R and LIEN, electronic laboratory of Nancy), in
the EQUAST project, we use our formal models to produce SystemC code [Sys99] and to finally
produce VHDL code by automatic translation from SystemC code. This paper presents some
rules for translating event B models with writing conditions to a “hierarchical” SystemC code
which implements the scheduling and treatments defined in formal models.

1.5 Summary of the paper

The next section presents the construction of a hierarchy of models and the refinement over this
hierarchy. Section 3 introduces the derivation of an efficient algorithm preserving properties of
models. Section 4 presents the generation of a complete architecture in SystemC. We show how
to derive modules for different tasks and communications between these modules with the use of
SystemC facilities. Section 5 concludes and gives future works.

2 Formal derivation of a hierarchy

From the initial specifications, we have constructed B event models with a special attention to
the relation between calculi. In fact, we used invariant properties for classified the results of
computations.

2.1 A formal hierarchy

We can semantically classify errors (and related parameters) using invariant properties that relate
the associated indicators. We take a set of parameters (4 exactly for this example) which can take
three different values:

VALUES = \{OK, KO, UND\}

\text{Task}_1, \text{Task}_2, \text{Task}_3, \text{Task}_4 \in \text{VALUES}

The meaning of the three values is:

- \text{Task}_j = OK means that \text{Task}_j has been computed and it value is correct.
- \text{Task}_j = KO means that \text{Task}_j has been computed and it value is bad.
- \text{Task}_j = UND means that \text{Task}_j has not been computed yet or has no sense.

Finally, we can introduce two next properties:

\textbf{Property 1} \text{Task}_2 \neq \text{UND} \Rightarrow \text{Task}_1 = \text{OK} \text{ that we note } \text{Task}_2 \prec \text{Task}_1

\textbf{Property 2} \text{Task}_3 \neq \text{UND} \Rightarrow \text{Task}_4 = \text{UND}

The first property is dependency property and the next is mutual exclusion properties. Property
1 means that when \text{Task}_2 is active or has computed its value, the \text{Task}_1 has always computed
its result and the result is correct. Property 1 says that we can not compute \text{Task}_2 while \text{Task}_1
has not computed a good value. This property precises the scheduling of treatments, we know
exactly the progression of the different calculi in the hierarchy.
Last, property 2 expresses the mutual exclusion of two tasks and means that we cannot activate Task₃ and Task₄ at the same time. For example, if the two tasks Task₃ and Task₄ are independent, we can minimize something. We can load, on the same part of an FPGA for example, the implementation of the concerned tasks. This mechanism is called *dynamic reconfiguration* and the property 2 of our models determines the different possible configurations for a set of tasks.

Finally, we can see the progression of computation in the hierarchy: the root of the hierarchy is the most important parameter, all parameters are dependent on it. When a parameter is computed and is *OK*, the computation of these parameters that directly depend on it can be scheduled. With type of current data, any branch is not computed since not defined or in error state. Cases are shown in figure 1.

### 2.2 Transitivity of the relation \(\prec\)

We prove, with some B formal models, than the dependency relation \(\prec\) defined between results of computation is transitive. We have proved easily this interesting property with Click’n’Prove [AC03, Cle04]. The proof is completely automatic and we do not detail the proof construction (This proof is a simple “do case”) in this present article. Finally we state:

\[
\text{Task}_C \prec \text{Task}_B \land \text{Task}_B \prec \text{Task}_A \Rightarrow \text{Task}_C \prec \text{Task}_A
\]

The relation defined and used in our model is an order relation which permits the scheduling of computations. The transitivity of this relation is an important property for an easy refinement as shown in the next section.

If the relation is transitive, the detection of a cycle in the hierarchy constructed is the proof of an important inconsistency in the specification. If the hierarchy contains a cycle, each task included in this cycle must wait the correct end (*OK*) of all its predecessors. There is a *deadlock* because all tasks must be evaluated at, exactly, the same time and with the property of the relation, resulting of all tasks are never incorrect. Because results of all tasks can not be incorrect, we affirm than the cycle existence in a hierarchy is a proof of inconsistency. Finally, we can assert that the hierarchy is an acyclic graph (DAG).

In our case study, we have not found cycles during all the modeling. We affirm that it is a validation of the consistency of initial specifications (here normative requirements) because we can possibly execute all defined tasks and all defined results are reachable.

### 2.3 Refining the directed acyclic graph

The B method is based on the refinement notion and the use of refinement permits us to incrementally construct hierarchy. A correct way for introducing parameters with refinement is adding of “new” computations as leaves of the incremental hierarchy (see figure 2). But with the proof of
transitivity of the dependency relation, we can introduce also, in the hierarchy, new parameters without loss of consistency, as shown in figure 2.

In this second case, the refinement introduces an intermediate result needed to compute the leaf. It is not the best way to refine and this shows that there is a little error on the importance of computations in the normative requirements. As shown previously, the proof of transitivity of dependency relation ensures than the consistency of the incremental hierarchy is preserved.

Finally, the adding of new computations is more easy when tasks are adding as leaves of the hierarchy; it is the normal way of refinement. The adding of new calculi in the hierarchy itself is possible, without lost of the global consistency, but is more difficult and is less “natural”. If the defined relation was not transitive, new computations could only be introduced at the top of hierarchy or as leaves, and this is a strong constraint. The initial requirement doesn’t explain the scheduling of computation and the construction of formal incremental models with this constraint is really hard. In our case study, we find an example of important computations described as optional calculi.

3 Deriving an algorithm

3.1 Task as a set of events

We present now the dynamic aspect of our models: events. B events are used in our models to simulate task computation: a single computation C is represented by two events that respectively model success (OK) and failure (KO) of C:

\[
\text{event}_{C, \text{OK}} = \begin{align*}
\text{SELECT} \\
\quad \text{Task}_B &= \text{OK} \land \\
\quad \text{Task}_C &= \text{UND} \land \\
\quad \text{Cond}_C \\
\text{THEN} \\
\quad \text{Task}_C &: = \text{OK} \\
\quad x : S'(x) \\
\text{END}
\end{align*}
\]

The two events represent the computation Task\(_C\) and the assignments of the variable Task\(_C\) indicate the end of this computation. The guards of the two events preserve the invariant and, in particular, the property of order relation: Task\(_C\) \(\neq\) UND \(\Rightarrow\) Task\(_B\) = OK.

In general, for each task we construct a couple of events (for the two possible results OK and KO) with respect of invariant properties (order relation). In a couple of events, the guards contain properties to preserve the invariant (based on order relation) and “calculi needing” properties (Cond\(_C\) in our example). These properties model the computation itself and must describe the different actions made by task for its evaluation.

3.2 Pseudo-code generation

It is easy to imagine the pseudo-C code that evaluates the two results Task\(_B\) and Task\(_C\), when we suppose than Task\(_A\) has been correctly evaluated. In the next generated code, the variables

\[
\text{event}_{C, \text{KO}} = \begin{align*}
\text{SELECT} \\
\quad \text{Task}_B &= \text{OK} \land \\
\quad \text{Task}_C &= \text{UND} \land \\
\quad \neg \text{Cond}_C \\
\text{THEN} \\
\quad \text{Task}_C &: = \text{KO} \\
\text{END}
\end{align*}
\]
Task\textsubscript{X} serve only as control variables and have not a strong meaning, and they can be eliminated altogether. With the use of sensitivity mechanism of SystemC [Sys99], we can easily implement these controls (see part 4). The different assignments of these control variables have different meanings too:

- Task\textsubscript{X} = OK means that computations can progress.
- Task\textsubscript{X} = KO means that an error must be indicated.

\begin{verbatim}
Task\_A = OK;
if Cond\_B then
    /* event\_B\_OK code */
    Task\_B = OK;
    /* update variables x : S(x) */
    if Cond\_C then
        /* event\_C\_OK code */
        Task\_C = OK;
        /* update variables x : S'(x) */
    else
        /* event\_C\_KO code */
        Task\_C = KO;
        /* Error detected */
        /* End of analysis */
    endif
else
    /* event\_B\_KO code */
    Task\_B = KO;
    /* Error detected */
    /* End of analysis */
endif
\end{verbatim}

The structure of the generated code is interesting because we can really see the succession of computations. The then part of conditionals contains the stack of all dependent computations, with respect to the hierarchy, whereas the else branches of conditionals contain the error for the current computation. The dependent computations are not executed as defined in our B models and derived hierarchy.

In conclusion, we confirm that control variables Task\textsubscript{X} have not a real utility and we can suppress, in the previous code, these variables. Because error is hierarchic, we can use some common variables error\_n with an error code for implementing the detection of an incorrect result in any task (Task\textsubscript{X} = KO). In this way, we limit the number of variables used in our implementation. On another hand, the assignment Task\textsubscript{X} = OK is not interesting because, from the structure of the code itself and without know the value of Task\textsubscript{X}, we see precisely execution of the algorithm.

The generated code implements the computation of all tasks with a total respect to incremental hierarchy defined by the structure of the code itself. But, our goal is the construction of hardware architecture and not really the resulting sequential algorithm (interesting in it structure).

4 **Translation of B event into SystemC**

The SystemC language offers facilities for modular construction and sensitivity of modules on input. In fact, a module declares input and output ports and methods. The principle of sensitivity is the activation of method, when inputs are changed.

4.1 General principles

We start from a hierarchy proved with the B formal method and produce SystemC code that satisfies important properties of formal models and of hierarchy in its architecture (see figure 3). The hierarchy is derived from the formal models with invariant property and guards of events. It requires a scheduling for computations. Evaluation of a connect stops when the first error is encountered. In fact, we propose some rules for the translation of hierarchy in SystemC modules and connectors with different modules. The main idea of our translation is the generation of a
module for each different tasks. We can, with facilities of SystemC, implement control variables and structure of the previous algorithm by input ports with sensitivity. With this approach, modules are automatically executed when necessary without tests. First, we select the two events modeling the evaluation of a calculus. From the guards of these events and invariant properties, (more generally hierarchy) we can derive input ports and the sensitivity of the module:

- the presence of a $Task_X = OK$ property in the two guards and a dependency property in the invariant implies the generation of a sensitive to input port.
- the result of the treatment is systematically writing on an output port (eventually error).
- the use of other identifiers implies the generation of input ports for each different identifier.

With the action part of the event, we can derive the code of the module and the output ports.

4.2 General architecture

We require a standard structure for the generated architecture. The final architecture implements the hierarchy represented in the B models, but must also satisfy constraints related to electronics. In particular, we must anticipate a reset module for the system and an interface module needed for the communication between environment and system. We briefly describe the two modules interface and reset:

- The module interface has for input the initial data. It implements the extraction of information contained in data that is required by the computing tasks and distributes this information to concerned modules.
- The module reset has for input the output of modules implementing tasks at the leaves of the hierarchy but not least. When these are finished, this module resets all channels of communication and all values for the next computation.

Therefore, the generated system looks like the architecture on figure 4.
4.3 A simple B model

We illustrate our approach by a simple but complete example. We start from a B model which models a simple hierarchy composed by the three computations used Task_A, Task_B and Task_C with a dependency relation. First we present the header and the invariant of our model:

```
4.3 A simple B model

We illustrate our approach by a simple but complete example. We start from a B model which models a simple hierarchy composed by the three computations used Task_A, Task_B and Task_C with a dependency relation. First we present the header and the invariant of our model:

```

```

SETS
VALUE = {OK, KO, UND};
DATA
CONSTANTS
getA, getB, getC, getD

VARIABLES
current, memory,
Task_A, Task_B, Task_C, ...

INVARIANT
current ∈ DATA∧
memory ∈ N → N

... Dependencies of tasks */
(Task_B ≠ UND ⇒ Task_A = OK)
(Task_C ≠ UND ⇒ Task_A = OK)
```

In our model, the variable current represents the current data (packet for example) and the different functions getA, getB, getC and getD are some “reading functions”. This kind of B variables model the research of informations needed for tasks in the data. For example, we can imagine the reading of a flag or a field in the header of a packet. The memory variable models a memory storing important informations for treatments of Task_B.

For translating B events to SystemC, we must state writing conventions: the set VALUE contains the three possible values for results of computations. The INVARIANT clause consists of the order relation which represent the previous hierarchy.

The two tasks Task_B and Task_C can be executed at the same time when Task_A is completed and we will maintain this parallelism in the final obtained architecture.

4.4 Events of computation

We present now the events associated with the different computations:

```
4.3 A simple B model

We illustrate our approach by a simple but complete example. We start from a B model which models a simple hierarchy composed by the three computations used Task_A, Task_B and Task_C with a dependency relation. First we present the header and the invariant of our model:

```

```

```

```

```

```

```

In our model, the variable current represents the current data (packet for example) and the different functions getA, getB, getC and getD are some “reading functions”. This kind of B variables model the research of informations needed for tasks in the data. For example, we can imagine the reading of a flag or a field in the header of a packet. The memory variable models a memory storing important informations for treatments of Task_B.

For translating B events to SystemC, we must state writing conventions: the set VALUE contains the three possible values for results of computations. The INVARIANT clause consists of the order relation which represent the previous hierarchy.

The two tasks Task_B and Task_C can be executed at the same time when Task_A is completed and we will maintain this parallelism in the final obtained architecture.

4.4 Events of computation

We present now the events associated with the different computations:

```
4.4 Events of computation

We present now the events associated with the different computations:

```

```

```

```

```

```

The two events shown above represent the computation of the variable Task_A. The condition getA(current) = OK in the guard of the first event (and the opposite in the second) represents the computation itself and is different from the condition Task_A = UND which controls the execution of the events.

```

```

```

```

```

The two events shown above represent the computation of the variable Task_A. The condition getA(current) = OK in the guard of the first event (and the opposite in the second) represents the computation itself and is different from the condition Task_A = UND which controls the execution of the events.
These two events represent the computation of the task $Task_B$. We can easily differentiate the scheduling properties of the guard and the properties modeling the computation itself.

\[
\begin{align*}
\text{eventC_OK} &= \text{SELECT} \\
&\quad \text{Task}_A = OK \land \text{Task}_C = \text{UND} \land \text{getC(current)} = OK \\
&\quad \text{THEN} \\
&\quad \text{Task}_C := OK \land \\
&\quad \text{END}
\end{align*}
\]

\[
\begin{align*}
\text{eventC_KO} &= \text{SELECT} \\
&\quad \text{Task}_A = OK \land \text{Task}_C = \text{UND} \land \text{getC(current)} = KO \\
&\quad \text{THEN} \\
&\quad \text{Task}_C := KO \\
&\quad \text{END}
\end{align*}
\]

The two last events represent the computing of $Task_C$. As above, the property $\text{getC(current)} = KO$ represents the computation of the task.

### 4.5 Resulting architecture

The code of each couple of events ($eventX_{\bullet}$) is implemented in different modules. The major difficulty of the translation is to separate the different informations contained in the guards of events. As described, some conditions contained in guards are used, with invariant properties, for hierarchy and scheduling of treatments. On the other hand, some properties of event guards are used for treatments itself. In this case, the invariant of model does not contain specific informations and the concerned properties in the guard are only a conditional.

All different modules have an error output port and write an error code on this output port if necessary. A module named $ERROR$ is generated for the treatments of this different error. This module contains an array for stock the errors detected by the different modules. When treatment of an error is achieved, the output port $\text{flag_end_error}$ is positioned to up (true). This module $ERROR$ is need for statistical evaluation in our case study but we can imagine another application where this module is not created. Because the system detects only one error at the same time, on a sequential connect of modules, there is no conflict between two modules which want writing on the same communication canal at the same moment. This statement is trivially deduced from the hierarchy built in the B models because of the dependency relation used. If a computation has a bad result, all dependent computations are not evaluated and, of course, cannot produce or detect an error. In the case of parallelism, two or more errors can appears at the same time but these error are not in the same connect of hierarchy. We used a different port error for each separate connect because of parallelism. The figure 5 shows the resulting architecture from our simple example and the interaction between modules.
We present now the headers of the generated modules. For the moment, the generation of this code is manual but we plan that an automatic translation is possible. We present now the header file named `eval_X.h` obtained for the computation of Task\(X\):

```c
MODULE(EVAL_X) {
    // Input ports of module
    sc_in<???> getX;
    sc_in<bool> flag_end_Y, ...; // flag end of all fathers of Task_X
    // Output ports of module
    sc_out<int> error_X; // code of error
    sc_out<bool> flag_end_X; // end of treatments
    // Treatments of module
    void treatments();
    SCCTOR(EVAL_X) {
        SCMETHOD(treatments);
        sensitive_pos << flag_end_Y << ...; // sensitivity
        don't_initialize();
    }
};
```

The generated header file of module lists all the “end flags” of all its fathers. The method treatments is sensitive to the union of all father flags and we write it in SystemC: `sensitive_pos << flag_end_Y << ...`. This notation means that the method treatments is executed when one of its input port going up (ie the value of an input port going up from false to true). The `getX` input port contains informations needed for the treatments of Task\(X\) and the type of this port is the type of the informations needed.

In our simple example, the module EVAL_A implements the root of the hierarchy and is called when a new item of data is ready. There is no dependency on another computation. The two outputs of the module are the error and an indicator for executing dependent modules. The notation `sensitive << data_A` means that the method treatments is always executed when the input port `data_A` is modified.

The module EVAL_B (instantiation of module EVAL_X) implements the computation of Task\(B\). Because there is a dependency Task\(B < Task_A\) in our B model, this module is triggered when its input `flag_end_A` is going up (we generate the instruction `sensitive_pos << flag_end_A`). This sensitivity implements correctly the dependency between Task\(A\) and Task\(B\). For the same reasons, the module EVAL_C (instantiation of module EVAL_X but contained in header file `eval_C.h`) is a correct implementation of the computation of Task\(C\) with respect to the formal hierarchy. We can finally conclude that the proposed architecture implements correctly the formal hierarchy defined in the B model.

The next listing shows the treatments generated and implemented in the module EVAL_B. This example is little complicated but it represents a correct translation from events to the SystemC code. From the two guards of the concerned events (see page 8), we can deduce a conditional and the two different assignments of the B variable Task\(B\) are translated in writing on different output ports `flag_end_B` and `error_B`. The new assignment of the B variable memory is implemented by the new generated module `memory`. Module EVAL_B sends the needed informations (ie index and value) to `memory` via canals `index` and `memVal`. If an error occurs, a specific error code (here 2) for Task\(B\) is written on the specific error\_B port.

```c
#define  "evalB.h"

#include  "evalB.h"

void EVAL_B::treatments() {
    if (memVal.read() == getD.read() - 1) {
        // generated from evalB_OK event
        flag_end_B.write(true);
        memVal.write(getD.read());
        index.write(getB.read());
    }
    else {
        // generated from evalB_KO event
        error_B.write(2);
    }
}
```
The next code shows treatments of the module memory with two different methods Read and Write with different sensitivities and an array mem for memory. Method Read is sensitive on a changing value of getD (sensitive \(<\) getD) and method Write is sensitive on a changing of index (sensitive \(<\) index).

```cpp
#include "memory.h"

void MEMORY::Read() {
    memVal.write(mem[getD.read()]);
}

void MEMORY::Write() {
    mem[index.read()] = memVal.read();
}
```

Finally, the generated module reset for the defined hierarchy is probably the most complex and difficult to realise because it must re-initialize all connections for treatment of new data after the end of treatments of current data. Determining the end of treatments in the hierarchy (DAG) is tough. We present the different cases for our example:

- TaskA produces an error; treatments of current data are ending. More generally, if an error occurs at the top of DAG, others computations are not necessary.
- TaskB and TaskC are completed and correct; all treatments are finished and correct. More generally, all leaves of hierarchy are completed and correct.
- TaskB is finished and correct but TaskC is completed and incorrect; the current data were completely treated and the system must be reinitialized. More generally, all branches of hierarchy must give an output (error or completely ending) before execute the reset module.
- the last case is symmetric with respect to previous.

Finally the generated module reset must consider all different cases of ending. We propose to implement these treatments in different methods of the module reset with different sensibilities. For example, a method internAnd can be used to implement the end of all treatments. This method can be generated from the B model because we know exactly the set of the leaves of formal hierarchy.

```cpp
void reset::internAnd() {
    IntSig.write(flag_end_A.read() && flag_end_B.read());
} //internAnd
```

In this code, IntSig is an intern signal of the module reset and the next method GeneralReset is sensitive on an up front of this signal:

```cpp
void reset::GeneralReset() {
    flag_end_A.write(false);
    flag_end_B.write(false);
    flag_end_C.write(false);
    ...  
} //GeneralReset
```

5 Conclusion and future works

With some restrictions on the writing of B models, we can easily translate our B developments to SystemC and generate the different modules and their connections. The architecture obtained implements the properties of B models and, in particular, it conserves the scheduling and the dependency relation from the hierarchy. A difficulty in our work is the distinction of the different cases of ending and their implementations in a particular module. The code SystemC, produced with our rules from B models, is really implementable in a circuit; some rewriting are needed for an optimal translation from our SystemC code to synthetisable VHDL in particular in RESET module. The VHDL code obtained after these little transformations is synthetisable. Tests of
SystemC code have validated the built architecture on electronic aspects. Generated SystemC code is well structurated, lisible and it is easy to test separately each module for a greater test covering.

In another hand, a used of the mutual exclusion presented by property 2 with SystemC modules gives a solid base for the construction of the dynamic reconfiguration possibilities. All modules generated are, by construction and formally, correctly structured and we work on the deduction of an algorithm for dynamic reconfiguration from B models.

Some future works are the writing of a complete translator for generate automatically the SystemC code from B models and the writing of real treatments case for study the applicability and scalability of the idea.

References


