Compositional certification: the CERCLES^2 project

Philippe Baufreton[1], Emmanuel Chailloux[2], Jean-Louis Dufour[1], Grégoire Henry[3]*, Pascal Manoury[3], Etienne Prun[4], Florian Thibord[3], Philippe Wang[2]†

[1] Sagem Défense Sécurité - [2] LIP6/UPMC - [3] PPS/Paris Diderot - [4] ClearSY

1 Industrial motivation

The CERCLES^2 project (in French: *CERtification Compositionnelle des Logiciels Embarqués critiqueS et Sûrs*[1], that is to say: compositional certification of critical and safe embedded software) is a collaboration between two industrial partners (*Sagem* and *Clearsy*) and two academic partners (*Pierre et Marie Curie* and *Paris Diderot* universities) aiming at technically and economically improving the development process of safety-critical software, a typical target being avionics DO-178 level A software ([9]). These technical and economical objectives are driven by two corresponding convictions:

- technically, we believe that *formal methods* are the *key* to transforming the software certification objectives from the current obligation of means to an obligation of result, but how can we introduce them in the current test-oriented practice?
- economically, we believe that *reuse* is a key to cost reduction, but how can we introduce it in a domain where certification credits are hard to transfer from a project to another?

Of course, the link between the two objectives and the two techniques is not one-to-one: reuse must also have the ambition to have a positive impact on product's quality, and formal proof must also have the ambition to reduce the cost. This last point is interesting, because for some of us, replacing a means obligation by a result obligation borders on "masochism". The point is that a formal approach is the only way to avoid the "precautionary principle" in the design of the process answering to the DO-178 objectives: a formal approach permits to retain only added value activities and especially to reject nice-to-have activities.

The changing development process of critical software The last decade has confirmed a deep change in the way we design critical software: for the application part (the part that contains the "transfer function"), textual programming languages are "dead". More precisely, C is still "alive", but only as an intermediate format between graphical data-flow models and the compilers. Graphical data-flow languages like Scade ([3]) or Simulink are invading the embedded market because of two interesting characteristics:

- they are readable by system engineers, permitting an early detection of bugs;
- they are simpler than conventional languages, reducing the introduction of bugs.

Of course this vertuous combination of "lesser introduction / better detection" of bugs is not guaranteed simply by the graphical and data-flow paradigms: it also depends on the kind of application, and in some cases a textual (and/or non data-flow) description is still the clearest way to program (and in a few cases it is the only way). But the tone is set for at least the ten years to come, with a visible impact on the organizations: when we write "readable by system engineers" you must read "readable by 'old' system engineers, and written by 'young' ones", thus moving the system/software interface to

*This work started while the author, now at ocamlpro, was at PPS/Paris Diderot.

†This work started while the author, now at University of Cambridge, was at LIP6/UPMC.

[1] ANR-10-SEGI-017

the reduced perimeter of "connecting the application to the real world". The entropy decrease at this interface is economically appealing but changes significantly the manpower distribution between software and system people.

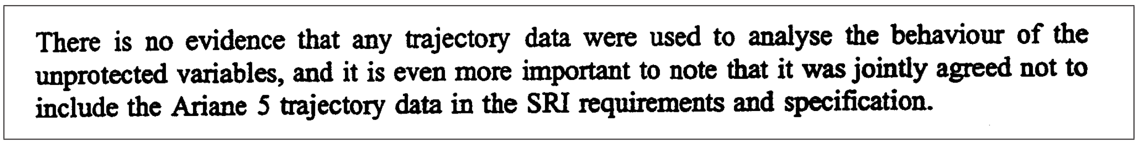

There is no evidence that any trajectory data were used to analyse the behaviour of the unprotected variables, and it is even more important to note that it was jointly agreed not to include the Ariane 5 trajectory data in the SRI requirements and specification.

Figure 1: The reason of the flight 501 failure ([7] middle of page 5)

Complexity, reuse and the Ariane flight 501 trauma We could naively believe that this evolution will improve the quality of products, and at a reduced cost! This would be true if complexity was kept constant, but of course complexity follows the capability of the hardware platforms. To date, the most efficient way to deal with complexity is the *divide-and-conquer* strategy. For critical software, this strategy is practiced in a top-down way, bottom-up occurring only at the level of basic libraries (mathematical functions [square root, ...], RT sequencer, ...).

It is to be contrasted with non-critical software (or other engineering disciplines), where the reuse level is not elementary, but can concern very complex components: it is not uncommon to have applications smaller than the librairies they use. In fact, in a critical context, reuse of complex components must be carefully planned, and certification authorities are rightly suspicious on the subject and formalized their requirements in the very prescriptive advisory circular AC20-148 (Reusable Software Requirements). The best lesson is given by the Ariane flight 501, where the Inertial Reference System (*SRI* in French) was "reused" from Ariane IV, missing the fact that the operational context was different (Fig 1).

This is an "equipment reuse bug" (coupled with a second bug in the process: the omission of verification at the system and equipment integration levels), and absolutely not a software bug as it is sometimes believed (the SRI does the job perfectly for Ariane IV).

The data-flow paradigm has a common feature with the object-oriented paradigm (and probably only one): they easily permit reuse at any scale. The reason is similar, and is called "encapsulation". Global variables don't exist, and instead blocks (objects) possess states, which are inaccessible from outside. In fact, a block can be seen as an object with two methods: an initialisation and a "step".

From the point of view of software correctness, this good reuse capability is just a new source of bugs if it is not accompanied by a verification of the compatibility of the reused subsystem with its new environment. In CERCLES[2] we propose to do this verification with the B method, which is to date the only formal method with powerful composition mechanisms. These mechanisms permit to have a simple translation between the data-flow world and the B world.

How to integrate a formal specification activity in a model-based process? To do this verification we have to formally specify the models: in DO-178 jargon data-flow models are formal LLRs (Low-Level Requirements; "formal" because they have [or should have] a formal semantics), and we need formal HLRs (High Level Requirements) or "contracts" to build B models.

For example, for the operation $y = sqrt(x)$, the LLR (and the B implementation) describes 6 iterations of the Newton-Raphson algorithm, whereas the HLR (and the B specification) will consist in the pre-condition $x >= 0$ and the post-condition $fabs(y * y - x) < 1e - 12$. The B method ([5]) framework provides those two levels of specification. Namely LLR are *implementations* and HLR are higher logical level specifications called *abstract machines*. Moreover, HRL and LLR are linked by a formal relation: *the refinement*.

The next section sets the guide line of *how to integrate a formal specification activity in a model-based process* and section 3 gives an instance of this. The last section discusses on the use of formal methods and standard certification compliance.

2 Integrated process

We can roughly describe the conventional process of software development as:

1. Provide the High Level Requirements allocated to the software.
2. Detail them in terms of data structures and computation procedures.
3. Realize them using a programming language.
4. Check that the running program meets its specification.

At step 2. the ability of reusing existing components can shorten the complexity of what is needed to detail: large parts of computation procedures may already have been implemented and the *divide-and-conquer* process can end with already conquered areas. The top-down strategy turns to be a bottom-up one by gluing components to achieve an overall computation procedure.

But as mentioned above, we must be sure that the area is actually conquered. With the rough development process we described, this verification comes at the very end of the all process. Using *formal verifications* allow early verification of the accuracy and the safety software component. But in order to achieve this, we need to have *formal components*. A conventional *Model Based Development* (MBD) process is closed to our needs but must be completed.

In a conventional approach, MBD is used for development and verification is done by testing, review and analysis. In CERCLES^2 approach, MBD is a means for component based design through legacy components reuse, thus augmented with some formal contract-based development so that we can proceed to formal verifications of system properties at the software-software integration level. That is to say: we can formally check the correctness of the composition.

Assume that components have been developed from SIMULINK®/ SCADE® models, assume they have been already validated in a previous context. Assume that, using SCADE® *assertions*, the models are equipped with their contracts. To inject them into a formal contract-based process, we need to have their formal expression in the chosen formal framework: which is, in our case, the one of the B method. This formal expression of SIMULINK®/ SCADE® can be obtained by means of a *SCADE® to B gateway* that translates SCADE® models into B *machines* as illustrated in figure 2

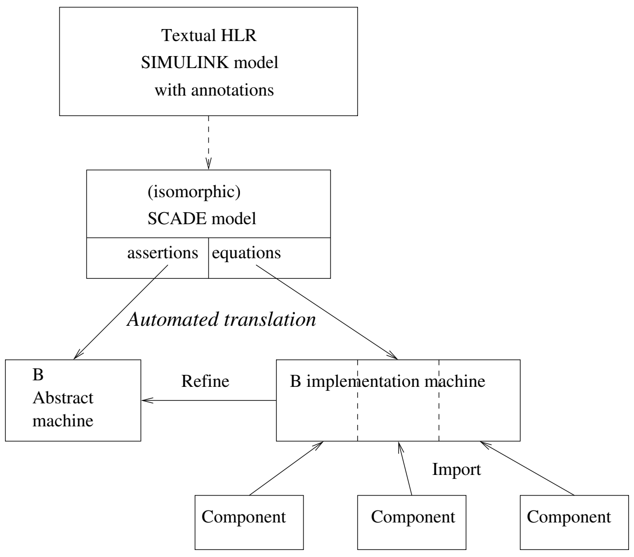


Figure 2: General translation schema

There are some points to detail:

1. the translation is feasible as the data-flow language SCADE® is semantically founded (it must be reachable to prove that the translation preserves the semantics).

2. the translation is recursive: SCADE® models components must also be translated and this recursive process ends with a one-to-one mapping of the basic operators.

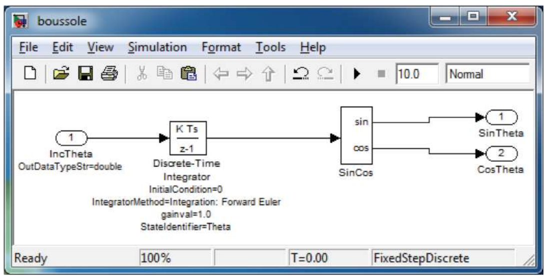
We can get now benefits of the formalization of both: the whole software model and the components models. This is due to the design of the formal B method:

- each B machine (abstract or concrete) is formally checked for internal coherence. This means, in particular, that concrete machines that make use of components are submitted to *proof obligations* in order to check that pre-conditions of components are satisfied in the context of their (re)use. This is the formal activity of *integration verification*;

- abstract machine (contracts) and concrete machines (implementations) are set to be related by the *refinement relation*. This means that the concrete machine is submitted to the *proof obligation* that, assuming the pre-condition of the abstract machine are satisfied, the computation settings of the concrete machine satisfies the requirement of the abstract post-condition. This is the formal activity of *unit verification*.

This double effect of formalizing with method B may be compared with the [11] experiment where Hoare logic was used to address only the second aspect of unit verifications on C code sources. More precisely, [11] provides a means for *unit proofs* of C modules. As we proceed to formal verification on the rephrasing (in B method) of the SCADE® model, we do less than [11]. But if we were able to justify the equivalence of SCADE® models and B models, as the code generation from SCADE® to C is qualified, we could pretend to do also *unit proofs*. This point is a matter of discussion.

Integration formal verification example Let's consider the following gyroscopic compass system (see figure 3) which computes a state value `Theta` and pass it to a `SinCos` function.



OutDataTypeStr=double
InitialCondition=0
IntegratorMethod=Integration: Forward Euler
gainval=1.0
StateIdentifier=Theta

Figure 3: Gyroscopic compass system

When formally checking it what appends is a failure to fulfill proof obligations:

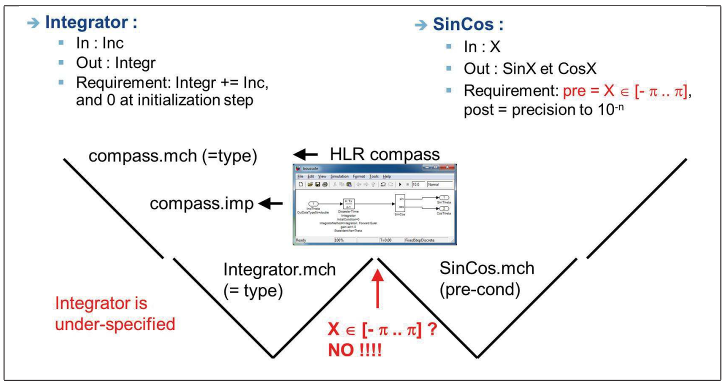


Figure 4: Integration failure

At this stage, we can

1. either change the integrator for a more accurate one computing a '2π modulo' on its output (impact on the code).
2. or add a '2π modulo' at the level of the glue code (impact on the design).

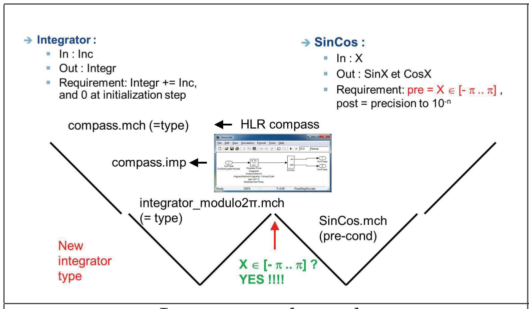
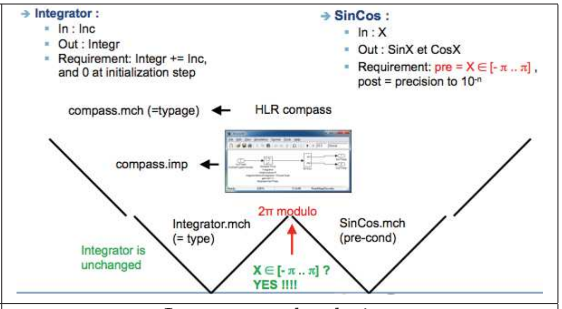
 Integrator : In : Inc Out : Integr Requirement: Integr += Inc, and 0 at initialization step SinCos : In : X Out : SinX et CosX Requirement: pre = X ∈ [- π .. π] , post = precision to 10⁻ⁿ compass.mch (=type) HLR compass compass.imp integrator_modulo2π.mch (= type) SinCos.mch (pre-cond) New integrator type X ∈ [- π .. π] ? YES !!!!	 Integrator : In : Inc Out : Integr Requirement: Integr += Inc, and 0 at initialization step SinCos : In : X Out : SinX et CosX Requirement: pre = X ∈ [- π .. π] , post = precision to 10⁻ⁿ compass.mch (=typage) HLR compass compass.imp Integrator.mch (= type) 2π modulo SinCos.mch (pre-cond) Integrator is unchanged X ∈ [- π .. π] ? YES !!!!
Impact on the code	Impact on the design

Figure 5: Proved Integration

Let's now detail a full example of this integration of MBD with formal verification.

3 From SCADE® to B

Let's consider the simple example of the specification of the system *nav1d*. This embedded system aims at estimating the position (abscissa) of a train on a track by integrating the values given by an odometer and the one given by markers put along the curve.

The *nav1d* system is built upon two preexistent components: *correction* and *integration*. The first one computes the abscissa by averaging the last estimated abscissa and the one given by a marker (when it exists). The second computes the current abscissa according to the value given by the odometer and the possible correction given by the *correction* component.

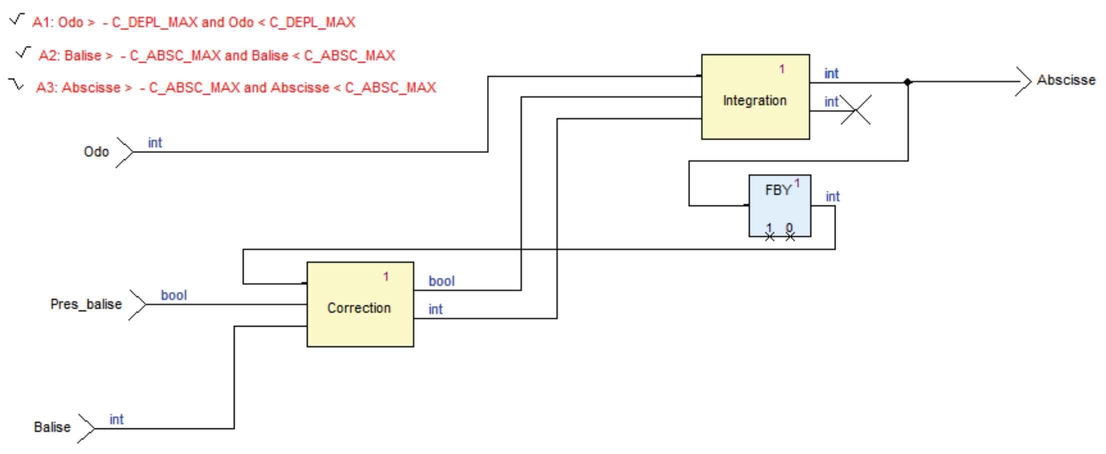

A1: Odo > - C_DEPL_MAX and Odo < C_DEPL_MAX
A2: Balise > - C_ABSC_MAX and Balise < C_ABSC_MAX
A3: Abscisse > - C_ABSC_MAX and Abscisse < C_ABSC_MAX
Odo
int
Integration
1
int
int
Abscisse
FBY
1
int
1 0
Pres_balise
bool
Correction
1
bool
int
Balise
int

Figure 6: *nav1d* component

To be able to prove the absence of overflow, it is necessary to put some constraints on the inputs and outputs of the node. They express some pre and post conditions richer than typing. So, let the following be: the input `Odo` belongs to the interval `-C_DEPL_MAX..C_DEPL_MAX`; the input `Balise` and the output `Abscisse` belong to the interval `-C_ABSC_MAX..C_ABSC_MAX`, where `C_DEPL_MAX` and `C_ABSC_MAX` are constants defined elsewhere. Those constraints are expressed in SCADE® by means of assertions.

From this graphical model, we get the textual SCADE® *node* definition

```
node Systeme(Odo : int; Pres_balise : bool; Balise : int)
  returns (Abscisse : int)
var
  _L1 : int;
  _L2 : bool;
  _L3 : int;
  _L5 : int;
  _L4 : bool;
  _L7 : int;
  _L6 : int;
  _L8 : int;
let
  _L1= Odo;
  _L2= Pres_balise;
  _L3= Balise;
  Abscisse= _L6;
  _L4, _L5= #1 Correction(_L8, _L2, _L3);
  _L6, _L7= #1 Integration(_L1, _L4, _L5);
  _= _L7;
  _L8= fby(_L6; 1; 0);
  assume A_1 : Odo <= C_DEPL_MAX and Odo >= -C_DEPL_MAX
  assume A_2 : Balise <= C_ABSC_MAX and Balise >= -C_ABSC_MAX
  guarantee G_1 : Abscisse <= C_ABSC_MAX and Abscisse >= -C_ABSC_MAX
tel
```

From this definition we isolate two parts:

- its *contract*, which is composed of the signature of the *node* and the assertions;
- its *implementation*, which is given by the sets of dataflow equations.

These two parts are the basis of the transcription of the SCADE®'s definition to its formal expression in the *method B* framework.

3.1 About B

One key concept of the formal method B is the *generalized substitution*. This is a language setting that allows to express abstract (and formal) specification of contracts as well as the program (in a minimalist imperative set of instructions) that realizes this contract. The point is that both belong to the same theoretical framework: a first order set theory. So the adequacy of a program to its expected contract can be set as a simple first order formula which validity can be verified by means of an usual mathematical proof. Those requirements to prove are called *proof obligations*.

A way to give an intuition of how generalized substitutions allow to formally verify that a program fulfills its contract is to consider the simplest substitution. Let's use the *Hoare's triple* notation ([4]) to express that, given some precondition, the execution of an instruction will establish some postcondition:

$$\{\ \mathrm{x{=}0}\ \}\ [\ \mathrm{x := x{+}1}\ ]\ \{\ \mathrm{x{=}1}\ \}$$

This means that starting from a configuration where the variable x has value 0, after having incremented it by 1, it has value 1. This can be *mathematicaly* expressed as

$$(x = 0) \Rightarrow (x = 1)[x := x + 1]$$

where the assignment $x := x + 1$ is interpreted as the usual operation of substitution of variables in a formula. That is to say that we want to ensure that

$$(x = 0) \Rightarrow (x + 1 = 1)$$

which is a mathematical truth.

Let's now exemplify this line on our *"Systeme"* example.

Contracts Using some set theoretic features, the expected contract can be formulated as

> *Given any* `Odo`$\in$`int`, `Pres_balise`$\in$`bool` *and* `Balise`$\in$`int` *such that* `Odo <=` *C_DEPL_MAX and* `Odo >= -`*C_DEPL_MAX and* `Balise <=` *C_ABSC_MAX and* `Balise >= -`*C_ABSC_MAX then* `Abscisse` *takes a value belonging to* $\{x \in int \mid x \leq$ *C_ABSC_MAX and* $x \geq$ -*C_ABSC_MAX*$\}$.

This is now expressible as the following generalized substitution (concrete ascii syntax):

```
PRE
   Balise : INT & Pres_balise : BOOL & Odo : INT &
   Odo <= C_DEPL_MAX & Odo >= -C_DEPL_MAX &
   Balise <= C_ABSC_MAX & Balise >= -C_ABSC_MAX
 THEN
   Abscisse :: { ii | ii : INT & ii <= C_ABSC_MAX & ii >= -C_ABSC_MAX}
 END
```

We use this to define an *operation* corresponding to the SCADE® node and we encapsulate this operation into a *machine*:

```
MACHINE M_Systeme
SEES M_Consts
OPERATIONS
Abscisse <-- Systeme(Odo, Pres_balise, Balise) =
  PRE
     Balise : INT & Pres_balise : BOOL & Odo : INT &
     Odo <= C_DEPL_MAX & Odo >= -C_DEPL_MAX &
     Balise <= C_ABSC_MAX & Balise >= -C_ABSC_MAX
   THEN
     Abscisse :: { ii | ii : INT & ii <= C_ABSC_MAX & ii >= -C_ABSC_MAX}
   END

 END
```

The *machine* is the B specification units. We assume that constants are defined in the machine `M_Consts`.

Implementations The second step of the transcription from SCADE® to B is the definition of the implementation that realizes the above contract. The key points of this goal are:

1. Use the previously defined components *Correction* and *Integration*.
   Actually, in the B framework, we only need to *import* their contracts that had to be given as B machines.

2. Express the dataflow equations as computable substitutions.
   This is, here, simply achieved by using basic substitutions of B that modelize the programming language's imperative *sequence* of instructions and *assignment* instructions. There is a dedicated syntax for the case where the right-hand part of the assignment is the result of the call of an operation of an imported component. The B machines containing computable operations are called *implementations*.

3. Take into account the use of the SCADE®'s temporal operator `fby`.
   For this, we introduce a state *concrete variable* to store the *previous value* of the dataflow for which we set an *initialisation* and an *invariant*. As we pass from a functional dataflow computation to an imperative state computation, we have to take care of the sequencing of assignments. We are guaranteed that a correct sequencing exists because we only translate SCADE®'s node definition that have been checked for *causality*.

4. Assert that the transcription of the equations defines a computation that fulfills the contract. This is achieved by using an other key concept of the method B: the *refinement*.

So, we get the following:

```
IMPLEMENTATION M_Systeme_i
REFINES M_Systeme
SEES M_Consts
IMPORTS M_Correction, M_Integration

CONCRETE_VARIABLES L8
INVARIANT
   L8 : INT
INITIALISATION
   L8 := 0

OPERATIONS
Abscisse <-- Systeme(Odo, Pres_balise, Balise) =
 VAR L1, L2, L3, L5, L4, L7, L6 IN
   L3 := Balise;
   L2 := Pres_balise;
   L1 := Odo;
   L4, L5 <-- Correction(L8, L2, L3);
   L6, L7 <-- Integration(L1, L4, L5);
   Abscisse := L6;
   L8 := L6
 END
END
```

For commodity, in the sequel, this literal transcription of the operation will be shorten into

```
Abscisse <-- Systeme(Odo, Pres_balise, Balise) =
 VAR L5, L4 IN
   L4, L5 <-- Correction(L8, Pres_balise, Balise);
   Abscisse, _ <-- Integration(Odo, L4, L5);
   L8 := Abscisse
 END
```

3.2 Formal verifications

With the method B, there are two kinds of formal verifications that are required. The first one aims to establish the internal coherence of a machine (or an implementation). The second aims at establishing the validity of the refinement relation between a machine and its claimed implementation.

Checking the internal coherence addresses the eventual invariant properties of the machine (or implementation). That is to say: verify that the initialization establishes the invariant and the operations do not violate it. Checking the validity of the refinement relation is to verify that the outputs of the operation computed by the implementation are those specified by the abstract operation of the refined machine. This is achieved by checking that, assuming the invariant of the abstract machine, the composition of the substitution defining the abstract operation (the contract) with the one defining the implementation of the operation validate the quality of outputs and the invariant of the implementation[2].

```
Balise : INT  ∧ Pres_balise : BOOL  ∧ Odo : INT  ∧
Odo ≤ C_DEPL_MAX  ∧ Odo ≥ -C_DEPL_MAX  ∧ Balise ≤ C_ABSC_MAX  ∧ Balise ≥ -C_ABSC_MAX

⇒

[ L4, L5 ← Correction(L8, Pres_balise, Balise);
    Abscisse', _ ← Integration(Odo, L4, L5);
    L8 := Abscisse'  ]
¬ [ Abscisse ::  ii | ii : INT  ∧ ii ≤ C_ABSC_MAX  ∧ ii ≥ -C_ABSC_MAX ]
¬ (Abscisse' = Abscisse)
```

Both initialization and operation are given in terms of generalized substitutions. Their semantics is given as a *predicate transformer semantics*, following the work of Hoare and Dijkstra. The application of a generalized substitution S to a formula F is noted $[S]F$. An application rule is given for each generalized substitution. For instance $[\texttt{PRE}\ G\ \texttt{THEN}\ S]F$ reduces to $G \wedge [S]F$; an operation call substitution is replaced by the defining substitution of the operation, up to suitable renaming of parameters. The base case is the one given at the beginning of this section: the usual substitution of a variable by an expression.

Due to the policy of the application of the operation calls, when an operation is defined with the preconditioned substitution, this precondition will appear in the transformation, though ensuring that it is satisfied at the call site.

4 Formal methods and certification standard compliance

Translating SCADE® models to the *formal models* that are the B machines offers the opportunity to use B-based frameworks (*e.g.*, *Atelier B*) as means of *formal analysis* of requirements.

This follows the *Formal Methods Supplement to DO-178C* ([10] FM.5.0)

> *Using formal models to define requirements or design artifacts allows some of the verification objectives to be satisfied by the use of formal analysis.*

The *Formal Methods Supplement* (FMS, in the sequel) defines three typical kinds of formal analysis. The one used in the CERCLES^2 project is (FM.1.6.2)

> *(1) Deductive methods involve mathematical arguments, such as mathematical proofs, for establishing a specified property of a formal model. A correct proof of a property provides rigorous evidence of that property for the formal model. These proofs are typically constructed using an automated or interactive theorem proving system. Even with such assistance, constructing proofs can be difficult, or impossible in some cases. However, once a proof is completed, automatic checking of the correctness of that proof is usually trivial.*

The others are model checking and abstract interpretation. Abstract interpretation has already been used and obtained certifications credits in the development of the Airbus A380 ([2]). Using model checking in relation with model-based development with SCADE® is illustrated in [8]. This usage of formal method shares with our approach a translation of the SCADE® models into available inputs for model checkers engines as well as theorem provers, including the mixed approaches that combine test and proof as the Hi-Lite project [6].

Of course, as a new accepted means (FM.6.2.1)

> *When formal analysis is used to meet a verification objective, there is an additional objective to ensure that the formal methods are correctly defined, justified, and appropriate to meet this verification objective. Activities for this are as follows:*

[2]Actually, it is checked that this composition cannot validate the non equality between (renamed) outputs.

a. All notations used for formal analysis should be verified to have precise, unambiguous, mathematically defined syntax and semantics; that is, they are formal notations.

b. The soundness of each formal analysis method should be justified. A sound method never asserts that a property is true when it may not be true.

c. All assumptions related to each formal analysis should be described and justified; for example, assumptions associated with the target computer or about the data range limits.

The long experiment of using the method B in industrial safety-critical software developments (even, if it was not in avionics – [1], for instance) make us confident with the reachability of the extra objectives given here.

References

[1] Behm, P., Benoit, P., Faivre, A., and Meynadier, J.-M. MéTéOR: A Successful Application of B in a Large Project. In *Proceedings of World Congress on Formal Methods (FM) - LNCS1709* (1999).

[2] Delmas, D., and Souyris, J. Astrée: From research to industry. In *Static Analysis Symposium (SAS)* (2007), pp. 437–451.

[3] Esterel Technologies. Scade Language Reference Manual. Tech. rep., Esterel Technologies SA, 2012.

[4] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (Oct. 1969), 576–585.

[5] J-R. Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 2005.

[6] Kanig, J., Guitton, J., and Moy, Y. Hi-Lite - Verification by Contract. *Softwaretechnik-Trends 31*, 3 (2011).

[7] Lions, J.-L., and et al. Ariane 5, Flight 501 Failure - Report by the Inquiry Board, 1996. Downloadable from http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf.

[8] Miller, S. P., Whalen, M. W., and Cofer, D. D. Software Model Checking Takes Off. *Communications of the ACM 53*, 2 (February 2012), 58–64.

[9] RTCA SC205, and EUROCAE WG71. DO-178C/ED-12C – Software Considerations in Airborne Systems and Equipment Certification. *Radio Technical Commission for Aeronautics* (Dec. 2012).

[10] RTCA SC205, and EUROCAE WG71. DO-333/ED-216 – Formal Methods Supplement to DO-178C and DO-278A. *Radio Technical Commission for Aeronautics* (Dec. 2012).

[11] Sourys, J., Wiels, V., Delmas, D., and Delseny, H. Formal verification of avionics software products. In *FM'09 Proceedings of World Congress on Formal Methods* (2009), pp. 532–546.