D.3.2.b Formal Verification of Properties
D.3.2.c Error Interpreter

Abstract

This report presents the work done for the instantiation of Formose methodology into the Atelier B. It gives the syntax of events accepted in Formose B Components and defines the meaning of refinements in terms of substitutions. Moreover, it provides implementation aspects of the configuration of the Proof Obligation Generator of the Atelier B to generate the Proof Obligations of the Formose methodology and explores the proof activity and the feedback of the results of this activity to the Formod tool.

Revision Table

Version	Date	Contributors	Contribution
1.0	11/10/2019	H. Ruiz Barradas (Clearsy)	Initial version

Contents

1 Introduction **3**

2 Formal verification of properties **4**
2.1 Syntax 4
2.2 Semantics and Proof Obligations 8
2.2.1 The Main Goal 8
2.2.2 Data Refinement 10
2.2.3 AND Refinement 11
2.2.4 OR Refinement 13
2.2.5 MILESTONE Refinement 14

3 Implementing Formose Proof Obligations **17**
3.1 Adaptations of the Atelier B 17
3.2 Parametrization of the Proof Generator 18
3.3 Adaptations to Formod Tool 22
3.3.1 Goal and Domain Diagrams 22
3.3.2 Proof Activity and Feedback to Formod 24

4 Conclusions **29**

A Proof of Theorem 1 **31**

B Proof of Derivation 2.1 **34**

Chapter 1

Introduction

A system specification in the Formose methodology is developed by stepwise refinements. At each refinement step the system specification is described by a Goal and Domain diagrams through the Formod tool using the requirements modeling language propose by Formose. In order to formally verify the specification process, these diagrams are automatically translated into B templates and then they are manually completed to produce B components adapted for the Formose methodology. These B components are syntactically verified and proved by the Atelier B tool.

In this report, we present the theoretical and practical aspects of the instantiation of the Goal and Domain models into the B methodology. In particular, a rigorous definition of the syntax and semantics of the Formose B components is given as well as practical aspects of this instantiation in the Atelier B tool.

This report is structured in two chapters. The first chapter presents the syntax and semantics of the Formose B components. The syntax restricts to SELECT substitution to be used as the body of events and expands the type of refinements supported for the events, whereas the semantics defines the Formose refinements through weakest preconditions or substitutions. These definitions, founded in the style of [1], are proved to be conform to original development in [3] inspired from [2]. The second chapter gives the practical aspects of the work made in the Atelier B. It explains the setup of resources to be set in the Atelier B to process Formose B components and shows how the Proof Obligations Generator is parametrized to generate the Proof Obligations defined in the Formose methodology. The chapter terminates with an exploration of the proof activity in the Atelier B and how the result of this proof activity can be feedback to the Formod tool. The report terminates with a conclusion of the work done.

Chapter 2

Formal verification of properties

Formal verification of properties in Formose is made by the formal proof of refinements. These properties can be explicitly expressed by logical formulas in the Domain Diagrams and then translated into invariants in the B components. Other properties can be implicitly stated by the type of refinement, stating that a goal must be established by the establishment of its subgoals in any particular order, or by the establishment of only one of its subgoal or by the establishment of the subgoals in a particular order.

In order to formally prove these properties, a formal semantics must be given to the requirement modeling language used to describe the Goal and Domain diagrams. The semantics of that language is formalized through the mapping into the Generalized Substitution Language of the B Method having an axiomatic semantics defined by weakest preconditions or substitutions. In this paragraph the syntax and semantics of the B components adapted for the Formose methodology are explained and the Proof Obligations to validate these components are discussed.

2.1 Syntax

B components in the Formose methodology have the same clauses that components in the Event B method, except the VARIANT clause which is not allowed in Formose. The difference among components in Formose and Event B is in the content of the EVENTS clause.

Events in the EVENTS clause for Formose are in bijection with the goals of a Goal diagram at certain abstraction level. The syntax of the body of these events is given by the following definition:

Definition 1 (*Formose Event B Body*)

The body of events is constrained to SELECT substitutions:

```
SELECT
 Guard
THEN
 Substitution
END
```

where *Guard* is a predicate over the state of the component and *Substitution* can be a simple substitution:

$$\texttt{x} := \texttt{E}$$

or a "becomes such that" substitution:

$$\texttt{x} : (Post(\texttt{x}))$$

where $Post(\texttt{x})$ is a predicate stating the properties of `x` holding after execution of the substitution.

Two things are to be noted about the *Substitution* in the body of the `select` substitution in Definition 1:

1. `x` can be a list of distinct variables, and in this case, `E` a list of expressions.
2. A simple substitution $\texttt{x} := \texttt{E}$ can always be rewritten as a "becomes such that" substitution $\texttt{x} : (\texttt{x} = \texttt{E})$.

At the highest abstraction level, the EVENTS clause of the component containing the main goal of the system specification contains the event denoting the main goal of a system specification. Therefore this clause must be defined as follows:

Definition 2 (*EVENTS Clause for Main Goal*)

The EVENTS clause must contain only one event:

```
EVENTS
  event_name = event_body
```

where *event_name* corresponds to the name of the main

goal and *event_body* is a SELECT substitution according to Definition 1.

According to the Formose methodology, any goal can be refined by four kinds of refinements: AND, OR, MILESTONE or data. In order to support these kind of refinements the `ref` keyword has been extended.

As a reminder, in Event B, an event e can be split in a refinement by a set of events $e_1, e_2, \ldots e_n$ or a similar set of events can be merged by a refinement into a single event. Syntactically, the split or merge of events in a refinement is managed by the keyword `ref` as follows:

Split: Event `e` is split by the set of events `e_1`, `e_2`, ... `e_n`:

```
e_1 ref e = b_1 ;
e_2 ref e = b_2 ;
...
e_n ref e = b_n
```

Merge: The set of events `e_1`, `e_2`, ... `e_n` is merged into event `e`:

```
e ref e_1, e_2, ... e_n = b
```

where `e` and each `e_i` are names of events and `b` and each `b_i` are the body of events coded by SELECT substitutions according to Definition 1.

In the Formose methodology the merge or split of events is not defined and the `ref` keyword is used to denote only a data refinement according to the following definition:

Definition 3 (*Data Refinement Syntax*)

Data refinement of an abstract event `e_a` by a refined event `e_c` is denoted by the following notation:

```
e_a ref e_c = event_body
```

where `event_body` is a SELECT substitution according to Definition 1.

AND, OR or MILESTONE refinements are specified by an extension of the `ref` keyword as stated by the following definitions:

Definition 4 (*AND Refinement Syntax*)

AND refinement of an abstract event `e_a` by a set of concrete events `e_c_1`, `e_c_2`, ... `e_c_n` is denoted by the following notation:

```
e_c_1 ref_and e_a = b_1 ;
e_c_2 ref_and e_a = b_2 ;
...
e_c_n ref_and e_a = b_n
```

where each `b_i` is a SELECT substitution according to Definition 1 not having common updated variables and the order of the refined events does not matter.

Definition 5 (*OR Refinement Syntax*)

OR refinement of an abstract event `e_a` by a set of concrete events `e_c_1`, `e_c_2`, ... `e_c_n` is denoted by the following notation:

```
e_c_1 ref_or e_a = b_1 ;
e_c_2 ref_or e_a = b_2 ;
...
e_c_n ref_or e_a = b_n
```

where each `b_i` is a SELECT substitution according to Definition 1 and the order of the refined events does not matter.

Definition 6 (*MILESTONE Refinement Syntax*)

MILESTONE refinement of an abstract event `e_a` by a sequence of concrete events `e_c_1`, `e_c_2`, ... `e_c_n` is denoted by the following notation:

```
e_c_1 ref_milestone e_a = b_1 ;
e_c_2 ref_milestone e_a = b_2 ;
...
e_c_n ref_milestone e_a = b_n
```

where each `b_i` is a SELECT substitution according to Definition 1. In this case the order of the refined events specifies the execution order of the concrete events.

In the next section we discuss the meaning of these refinements in the context of a Formose B component.

2.2 Semantics and Proof Obligations

In Event B, the computational model attached to any component is represented by a loop of guarded commands. Only one command is executed at any time if the condition stated in its guard holds. If the guards of two or more guarded commands hold at any iteration of the loop, only one of these commands is chosen for execution in a non deterministic way. Formose B components do not have attached such a computational model.

In the following sections the meaning of events in Formose B components is described and the proof obligations needed to validate these components are stated.

2.2.1 The Main Goal

As state by Definition 2, there is only one event at the top level of a Formose specification. The computational model associated at this top level can be seen as a guarded command executed under the condition of its guard. In this way, the meaning of the event associated to the main goal is given by the following definition:

Definition 7 (*Main Goal Semantics*)

Let e be the event denoted by $event\_name$ in Definition 2. Let x be the list of variables declared in the Formose B component containing e, I be the invariant to be preserved by e and C be the properties of the constants declared in the component. Under this context, the meaning of e is the update of x by the execution of *Substitution* preserving I under the assumption $I \wedge C \wedge Condition$, that is:

$$I \wedge C \wedge Condition \Rightarrow [Substitution]I$$

As noted in Section 2.1, *Substitution* can always be written as a "becomes such that" substitution $\mathtt{x} : (Post(\mathtt{x}))$. Therefore, the Proof Obligation to be proved to validate the Formose B component containing the main goal is stated as follows:

Proof Obligation 1 (*Main Goal*)

According to Definition 7 and *Substitution* being `x` : $(Post(\mathtt{x}))$, the Proof Obligation to be satisfied by the component containing the main goal is:

$$\forall x' \cdot (I \wedge C \wedge Condition \wedge Post(x') \Rightarrow [x := x']I)$$

where $x' \setminus (I, C, Guard\ Post(x))$, $Post(x') = [x := x']Post(x)$ and $Post(x)$ is the predicate satisfying the "becomes such that" substitution defining the body *Substitution*.

The event denoting the main goal, or any other event at lower abstraction level, must be proved to be feasible. It allows to ensure the existence of a state reached by the execution of that event when its guard is enabled. The semantics of this requirement is given as follows:

Definition 8 (*Feasibility*)

Let e by an event in a Formose B Component having invariant I and properties C. The feasibility of e is stated by the following predicate:

$$I \wedge C \wedge grd(e) \Rightarrow \neg[e]false$$

where $grd(e)$ for any event e satisfying Definition 1, is the condition *Guard* in its SELECT substitution.

Given e satisfying Definition 1, the Proof Obligation to be satisfied by a feasible event is:

Proof Obligation 2 (*feasibility*)

Let e by the event

$$\text{SELECT } Guard \text{ THEN } x : (Post(x)) \text{ END}$$

in the context of Definition 8. In order to be feasible, the Proof Obligation to be satisfied by e is:

$$I \wedge C \wedge Guard \Rightarrow \exists x' \cdot (Post(x'))$$

The event denoting the main goal, or any other event at lower abstraction level can be refined in four different ways as described in the previous

section. These refinements are defined in the following paragraphs and its corresponding proof obligations are stated.

2.2.2 Data Refinement

Data refinement is the only one refinement having a semantics taken from Event B. In this kind of refinement an abstract variable is refined by a concrete one and the relation between them is stated by a "gluing" invariant. Then, the meaning of a data refinement of an abstract event by a concrete one is defined by a predicate stating that the execution of the concrete event under the context of the abstract and gluing invariant must establish a post-condition indicating that the abstract event is not able to establish the negation of the gluing invariant. This is formalized by the following definition:

Definition 9 (*Data Refinement*)

Let M be a Formose B component with variables x, invariant I and properties C; let a be an event from M. Moreover, let N be another Formose B component with variables y, properties D and gluing invariant J relating variables y to x. Under this context, the meaning of a data refinement of an event a by an event c is stated as follows:

$$C \wedge I \wedge D \wedge J \Rightarrow [c]\neg[a]\neg J$$

Taking into account the form of events according to Definition 1 and the fact that substitutions in the body of a SELECT can be rewritten as a "becomes such that" substitution, the Proof Obligation required to validate a data refinement is stated as follows:

Proof Obligation 3 (*Data Refinement*)

Under the context of Definition 3, let a and c be the following events:

$$a = \text{SELECT } Guard_a(x) \text{ THEN } x : (Post_a(x)) \text{ END}$$
$$c = \text{SELECT } Guard_c(y) \text{ THEN } y : (Post_c(y)) \text{ END}$$

The proof obligation to be satisfied by the data refinement of

a by c is the following:

$$\forall y' \cdot (C \wedge I \wedge D \wedge J \wedge Guard_c \wedge Post_c(y') \Rightarrow \\ Guard_a \wedge \exists x' \cdot (Post_a(x') \wedge [y := y'][x := x']J))$$

where (x', y') are fresh variables not free in the context of components M and N, $Post_c(y') = [y := y']Post_c(y)$ and $Post_a(x') = [x := x']Post_a(x)$.

Data refinement semantics in Definition 9 is stated in terms of substitutions in the style of [1]. This style is well suited to be coded into the Atelier B tool. However, by substitution calculus, the Proof Obligation 3 are presented in terms of guards and postconditions in a style similar to [2]. In this way we can distinguish in Proof Obligation 3 the GRD, SIM and INV proof obligations described in [2]:

GRD

$$C \wedge I \wedge D \wedge J \wedge Guard_c \wedge Post_c(y') \Rightarrow Guard_a$$

SIM and INV

$$C \wedge I \wedge D \wedge J \wedge Guard_c \wedge Post_c(y') \\ \Rightarrow \exists x' \cdot (Post_a(x') \wedge [y := y'][x := x']J)$$

Finally we note that in this case, the predicated to be proved for SIM and INV contains an existential quantifier, and therefore a witness for x' must be exhibited. In [2] the WFIS rule asks for proof this witness explicitly. It allows to split SIM and INV Proof Obligation.

2.2.3 AND Refinement

The AND Refinement in Formose methodology is defined as the satisfaction of a parent goal by the conjunction of its subgoals [3]. These subgoals can be executed in any order, therefore in order to avoid dependences among the accesses to the variables manipulated by the subgoals, it is imposed that the set of manipulated variables in a subgoal must be disjoint with respect to the set of manipulated variables of any other subgoal.

Before giving the semantics of the AND Refinement in terms of substitutions, we present a theorem indicating that the parallel execution of two substitution having no common variables is equal to the sequential execution of the substitution in a non deterministic way. This is formalized as follows:

Theorem 1 (*Parallel Execution*)

Let $G1$ and $G2$ be two substitution defined as follows:

$$G1 = \text{SELECT } Guard_1(x) \text{ THEN } x : (Post_1(x)) \text{ END}$$
$$G2 = \text{SELECT } Guard_2(y) \text{ THEN } y : (Post_2(y)) \text{ END}$$

and $y \setminus (Guard_1(x), Post_1(x))$ and $x \setminus (Guard_2(y), Post_2(y))$.

The result of the parallel execution of $G1$ and $G2$ is the same that their sequential execution in a non deterministic way:

$$G1 \parallel G2 = ((G1 \,;\, G2) \,[\!]\, (G2 \,;\, G1))$$

The proof of this theorem is given in appendix A. A generalization of this theorem can be used to argue that the result of any execution sequence of events $e_1, e_2, \ldots e_n$, where $vars(e_i) \setminus vars(e_j)$, $i \neq j$, $\{i, j\} \subseteq 1..n$ and $vars(e)$ denoting the list of variables accessed by e, is equal to the parallel execution of these events. In this way we can state the semantics of an AND Refinement as follows:

Definition 10 (*AND Refinement*)

Let e be any event and $vars(e)$ be the list of variables accessed by e. Let M be a Formose B component with variables x, invariant I and properties C; let a be an event from M. Moreover, let N be another Formose B component with variables z, properties D and gluing invariant J relating variables z to x. Let E be a set of events $\{c_1, c_2, \ldots c_n\}$ from N, where $vars(c_i) \setminus vars(c_j)$ for any distinct indexes i and j in the interval $1..n$. Under this context, the meaning of an AND Refinement of an event a by the set of events E is stated as follows:

$$C \wedge I \wedge D \wedge J \Rightarrow [c_1 \parallel c_2 \parallel \ldots c_n] \neg [a] \neg J$$

Using the form of events in Definition 1 and the generalization of A.1 in appendix A to n events in Theorem 1, we state the Proof Obligations for the AND Refinement as follows:

Proof Obligation 4 (*AND Refinement*)

In the context of Definition 10 let a and c_i, for $i \in 1..n$, be

the events

$$a = \text{SELECT } Guard_a(x) \text{ THEN } x : (Post_a(x)) \text{ END}$$
$$c_i = \text{SELECT } Guard_i(y_i) \text{ THEN } y_i : (Post_i(y_i)) \text{ END}$$

Let y be the list of variables $y_1, y_2, \ldots y_n$ and y' be the list of variables $y'_1, y'_2, \ldots y'_n$. The Proof Obligation to be satisfied by the AND Refinement of a by the set of events E is:

$$\forall y' \cdot (\forall i \cdot (i \in 1..n \Rightarrow C \wedge I \wedge D \wedge J \wedge Guard_i(y_i) \wedge Post_i(y'_i)) \\ \Rightarrow Guard_a \wedge \exists x' \cdot (Post_a(x') \wedge [y := y'][x := x']J))$$

In must be noted that the Proof Obligation 4 is weaker than the Proof Obligations for AND Refinement presented in [3].

2.2.4 OR Refinement

The OR Refinement in Formose methodology of a parent goal by a set of mutually exclusive subgoals is defined as the achievement of only one subgoal. This kind of refinement is formalized with the help of a bounded choice substitution as follows:

Definition 11 (*OR Refinement*)

Let M be a Formose B component with variables x, invariant I and properties C; let a be an event from M. Moreover, let N be another Formose B component with variables z, properties D and gluing invariant J relating variables z to x. Let E be a set of mutually exclusive events $\{c_1, c_2, \ldots c_n\}$ from N where any distinct events u and v from E satisfy $\neg(grd(u) \wedge grd(v))$, where $grd(e)$, for any event e satisfying Definition 1, is the condition *Guard* in its SELECT substitution. Under this context, the meaning of an OR Refinement of an event a by the set of events E is stated as follows:

$$C \wedge I \wedge D \wedge J \Rightarrow [c_1 \,[\!]\, c_2 \,[\!]\, \ldots c_n]\neg[a]\neg J$$

Taking the form of events in Definition 1 and applying the substitution calculus rules, we get the Proof Obligation for the OR Refinement:

Proof Obligation 5 (*OR Refinement*)

In the context of Definition 11 let a and c_i, for $i \in 1..n$, be the events

$$a = \text{SELECT } Guard_a(x) \text{ THEN } x : (Post_a(x)) \text{ END}$$
$$c_i = \text{SELECT } Guard_i(y_i) \text{ THEN } y_i : (Post_i(y_i)) \text{ END}$$

The Proof Obligations to be satisfied by the OR Refinement of a by the set of events E are:

$$\forall i.(i \in 1..n \Rightarrow \forall y' \cdot (C \wedge I \wedge D \wedge J \wedge Guard_i(y_i) \wedge Post_i(y_i') \Rightarrow Guard_a \wedge \exists x' \cdot (Post_a(x') \wedge [y_i := y_i'][x := x']J))) \text{ and}$$

$$\forall j, k.(\{j, k\} \subseteq 1..n \wedge j \neq k \Rightarrow \neg(Guard_j(y_j) \wedge Guard_k(y_k)))$$

In order to guarantee that only one event in an OR Refinement can be executed at any time, Proof Obligation 5 states that any two distinct events of the refinement set c_j and c_k do not have to be enabled at the same time : $\neg(grd(c_j) \wedge grd(c_k))$. We note that this statement is different from [3] indicating that the exclusivity condition is $\neg(Post_j(y_j) \wedge Guard_k(y_k))$ which does not provides the exclusivity guarantee because nothing ensures that c_j is not enabled when c_k is enabled.

2.2.5 MILESTONE Refinement

The MILESTONE Refinement in Formose methodology is defined as the satisfaction of a parent goal by a sequential execution of subgoals introducing intermediate states to reach a final state where the parent goal is fulfilled [3]. As it is needed to reach intermediate states to reach a final state, miraculous events are proscribed. This kind of refinement is formalized with the help of sequential substitution as follows:

Definition 12 (*MILESTONE Refinement*)

Let M be a Formose B component with variables x, invariant I and properties C; let a be an event from M. Moreover, let N be another Formose B component with variables z, properties D and gluing invariant J relating variables z to x. Let E be a sequence of non miraculous events $[c_1, c_2, \ldots c_n]$ from N where $[c_1]grd(c_2)$ and $[c_1 \ ; \ \ldots c_i]grd(c_{i+1})$, for $i \in 2..n-1$, where

$grd(e)$, for any event e satisfying Definition 1, is the condition *Guard* in its SELECT substitution. Under this context, the meaning of a MILESTONE Refinement of an event a by the sequence of events E is stated as follows:

$$C \wedge I \wedge D \wedge J \Rightarrow [c_1 \ ; c_2 \ ; \ldots c_n] \neg [a] \neg J$$

As $[S \ ; T]Q \Leftrightarrow [S][T]Q$, the generalization to the sequence of events E in Definition 12 cannot be expressed by universal quantification over the set of events as it is done in Proof Obligations 4 and 5. Therefore, a recursive definition is given to express the Proof Obligation:

Proof Obligation 6 (*MILESTONE Refinement*)

For a list of substitutions l and predicate Q, let wp_seq be the following function:

$$wp\_seq(l, Q) = \begin{cases} [\mathsf{first}(l)] wp\_seq(\mathsf{tail}(l), Q) & \text{if } l \neq [] \\ Q & \text{if } l = [] \end{cases}$$

In the context of Definition 12, the Proof Obligation to be satisfied by the MILESTONE Refinement of a by the sequence of events E are:

$$C \wedge I \wedge D \wedge J \Rightarrow wp\_seq(E, \neg [a] \neg J) \text{ and}$$

$$C \wedge I \wedge D \wedge J \Rightarrow ([c_1] grd(c_2) \wedge \forall i.(i \in 2..n-1 \Rightarrow [c_1 \ ; \ldots c_i] grd(c_{i+1})))$$

where $\neg [a] \neg J \Leftrightarrow Guard_a \wedge \exists x' \cdot (Post_a(x') \wedge [x := x']J)$.

Let us suppose an abstract event a and two concrete events c_1 and c_2 satisfying Definition 1:

$$a = \text{SELECT } Guard_a(x) \text{ THEN } x : (Post_a(x)) \text{ END}$$
$$c_1 = \text{SELECT } Guard_{c_1}(y_{c_1}) \text{ THEN } y_{c_1} : (Post_{c_1}(y_{c_1})) \text{ END}$$
$$c_2 = \text{SELECT } Guard_{c_2}(y_{c_2}) \text{ THEN } y_{c_2} : (Post_{c_2}(y_{c_2})) \text{ END}$$

The application of the first part of the Proof Obligation 6, with a sequence E of two events $[c_1, c_2]$, milestone refined by event a, gives the following

derivation (see proof in appendix B):

$$\begin{aligned}&\forall y'_{c_1}, y'_{c_2} \cdot (C \wedge I \wedge D \wedge J \wedge Guard_{c_1} \wedge Post_{c_1}(y'_{c_1}) \\ &\quad \wedge [y_{c_1} := y'_{c_1}]Guard_{c_2} \wedge [y_{c_1} := y'_{c_1}]Post_{c_2}(y'_{c_2}) \\ &\quad \Rightarrow \\ &\quad Guard_a \wedge \exists x' \cdot (Post_a(x') \wedge [y_{c_1} := y'_{c_1}][y_{c_2} := y'_{c_2}][x := x']J))\end{aligned} \tag{2.1}$$

The application of the Substitution Calculus to the second part of the Proof Obligation 6 gives as result:

$$\begin{aligned}&\forall y'_{c_1} \cdot (C \wedge I \wedge D \wedge J \wedge Guard_{c_1} \wedge Post_{c_1}(y'_{c_1}) \\ &\quad \Rightarrow [y_{c_1} := y'_{c_1}]Guard_{c_2})\end{aligned} \tag{2.2}$$

From the Proof Obligation 2.1 and 2.2 we can observe that, under the context of the refinement according to Definition 12, the three proofs obligations given in [3] must be done to validate the MILESTONE refinement of a by $c_1 \,;\, c_2$:

- Abstract Guard $Guard_a$ must be proved from the concrete guard $Guard_{c_1}$.
- Abstract Postcondition $Post_a(x')$ must be established by $[y_{c_1} := y'_{c_1}]Post_{c_2}(y'_{c_2})$ and
- The Guard of c_2, $[y_{c_1} := y'_{c_1}]Guard_{c_2}$, must be established by $Post_{c_1}(y'_{c_1})$.

However, the Proof Obligation 6 are weaker than the ones given in [3] and are sufficient to prove the intended meaning of the MILESTONE Refinement.

Chapter 3

Implementing Formose Proof Obligations

As explained in the previous chapter, the implementation of the Formose methodology is supported through two tools: Formod and Atelier B. Formod tool generates templates of Formose B components which are completed and validated by the Atelier B tool. In order to support this validation phase some adaptations were done in the Atelier B tool. This chapter traces these adaptations and explore the adaptations to be done in the Formod tool to support the feedback provided by the Atelier B.

3.1 Adaptations of the Atelier B

As the syntax and semantics of events in Formose B components are a variant of Event-B, the Atelier B projects intended to contain Formose B components must be declared as Event-B projects. However, in order to support the extended syntax of `ref` keyword as given in Definitions 4, 5 and 6 and the generation of the corresponding Proof Obligations, the following resources must be set in the configuration Files `AtelierB`:

Resources Atelier B

Atelier B projects containing Formose B Components must have the following resources in the configuration file `AtelierB`

in the `bdp` directory of the project:

```
ATB*ATB*Proof_Obligations_Generator_NG:TRUE
ATB*POG*Generate_EventB_Feasibility_PO:TRUE
ATB*TC*Event_B_Formose:TRUE
ATB*BCOMP*Event_B_Formose:TRUE
```

As can be noted from the previous resource set, the New Proof Obligation Generator of the Atelier B is activated. The following section explains how this generator was updated to generate the Formose Proof Obligations.

3.2 Parametrization of the Proof Generator

The New Proof Obligations Generator uses a parametrization file to describe how the Proof Obligations must be generated. This file describes a set of rules, or templates, to be applied to the XML representation of the B Components in order to generate the corresponding Proof Obligations. The XML format of B components is named BXML. The outcome of the application of the parametrization file to the BXML document is a set of Proof Obligations coded into the Generalized Substitution Language as XML elements. The XML format of the generated Proof Obligations is named POXML. The POXML file is further processed to discharge obvious proof obligations and then produce a file, with extension `.po` in Theory Language, to be processed by the provers of the Atelier B.

The standard parametrization file `paramGOPSystem.xsl`, used to generate Proof Obligations in Event-B, was adapted to generate the Formose Proof Obligations described in section 2.2. The adapted file `paramGOPFormose.xsl`, uses the same rules from Event-B to generate the Proof Obligations for invariant preservation, feasibility and data refinement. The rules for AND, OR and MILESTONE refinements were written from scratch.

The root element of a POXML document is described by the following excerpt in the RELAX NG Compact Syntax [4]:

Definition 13 (*POXML Root Element*)

```
start =
  element Proof_Obligations {
    element Define {...}+,
    element Proof_Obligation {
      ...
      element Definition{ attribute name{text}}+,
```

```
    element Hypothesis{ Predicate ? },
    element Goal{Predicate_or_SubCalculus}
  }+
}
```

According to the excerpt in Definition 13 a POXML document contains a list of `Define` and `Proof_Obligation` elements embedded in the root element. The contents of the `Define` elements are used to factorize global hypothesis that will be referenced by the Proof Obligations. Each Proof Obligation has its corresponding `Proof_Obligation` element. Apart from definitions and management information, this element contains the hypothesis and goal to be proved embedded into its corresponding elements. The contents of the `Hypothesis` element is a predicate containing the conjunction of all hypothesis in the Proof Obligation. The contents of the `Goal` element is given by the following definition:

Definition 14 (*Predicate or Substitution Calculus*)

```
Predicate_or_SubCalculus = Predicate
  | element Sub_Calculus {Substitution,
      Predicate_or_SubCalculus}
  | element Not { Predicate_or_SubCalculus }
```

From Definition 14, the contents of the `Goal` element can be a predicate, a `Sub_Calculus` element or a `Not` element. Here, it is important to note that the contents of a `Sub_Calculus` element is a substitution and another Predicate_or_SubCalculus expression. Therefore, we can use directly Definitions 10, 11 and 12 to describe the Proof Obligations of the AND, OR and MILESTONE refinements as they are already stated in terms of substitutions and we do not have to use the Proof Obligations 4, 5 and 6 because they are calculated by the Proof Obligations Generator.

Then, the role of the parametrization file is to instruct the XSL processor to create mainly the list of `Proof_Obligation` elements from the BXML contents of the B components by applying the corresponding definitions.

The parametrization file starts by the creation of a map associating for each abstract event a and its type of refinement t, the list of refined events $[c_1, c_2, \ldots c_n]$:

Definition 15 (*Refinement Map*)

Let a be an event according to Definition 1, t an integer in the interval $0 \ldots 3$ denoting the type of refinement applied

to a, and ref_type the function associating to t a type of refinement:

$$ref\_type(t) = \{0 \mapsto \texttt{ref}, 1 \mapsto \texttt{ref\_and}, 2 \mapsto \texttt{ref\_or}, 3 \mapsto \texttt{ref\_milestone}\}(t)$$

The map $m(a,t)$ denotes the list of refined events, in the order where they appear in the source code of the Formose B component, refining the abstract event a:

$$m(a,t) = [c \mid c\ ref\_type(t)\ a]$$

It must be noted that in the case of data refinement, $m(a,0)$ denotes a list of a single element. In the other cases, $m(a,t)$, for $t \neq 0$, denotes a list having a size greater than 1.

Using the refinement map, for each type of refinement AND, OR or MILESTONE, the parametrization file contains a rule to create a substitution defined as follows:

Definition 16 (*Semantic Substitution*)

Let t be an integer in the interval $1 \ldots 3$ denoting a type of Formose refinement and $ref\_comp(t)$ be the following function:

$$ref\_comp(t) = \{1 \mapsto \|, 2 \mapsto [], 3 \mapsto ;\}(t)$$

The Semantic Substitution of the non empty list of events l for a refinement of type t, $Semantic\_Subs(l,t)$, is defined as follows:

$$Semantic\_Subs(l,t) = f(\mathsf{first}(l), \mathsf{tail}(l), t)$$

where

$$f(S,l,t) = \begin{cases} f(S\ ref\_comp(t)\ \mathsf{first}(l), \mathsf{tail}(l), t) & \text{if } l \neq [] \\ S & \text{if } l = [] \end{cases}$$

In this way, for example, using Definition 16, we can verify that

$$Semantic\_Subs([c_1, c_2, c_3], 1) = c_1 \parallel c_2 \parallel c_3$$

The Semantic Substitution can now be used to define the `Proof_Obligation` elements according to the following definition:

Definition 17 (*XSL Proof Obligation*)

The `Proof_Obligation` element corresponding to the Proof Obligation of the refinement of type t of an abstract event a by a sequence of concrete events $m(a,t)$ is defined as follows:

```
<Proof_Obligation>
   ...
   <Goal>
      <Sub_Calculus>
         XML_Semantic_Subs(m(a,t),t)
         <Not>
            <Sub_Calculus>
               XML_a
               <Not>
                  XML_J
               </Not>
            </Sub_Calculus>
         </Not>
      </Sub_Calculus>
   </Goal>
</Proof_Obligation>
```

where $m(a,t)$ is the list of concrete events refining a according to Definition 15 and $XML\_Semantic\_Subs$ denotes the XML substitution $Semantic\_Subs$ made up from the concatenation of substitutions in $m(a,t)$ by the $ref\_comp(t)$ operator according to Definition 16 and XML_a and XML_J are the XML abstract event a and gluing invariant J.

From Definition 17, by instantiation of t to the values 1, 2 or 3, we can verify that the `Proof_Obligation` element codes the semantics of the AND, OR and MILESTONE refinement involving the Substitution Calculus as stated in their corresponding Definitions 10, 11 and 12. We can note that in fact, the Proof Obligations generated in this way corresponds to the data refinement of the abstract event by the composition of the concrete events.

A similar approach was used to generate the Proof Obligations of the establishment of guards in the MILESTONE refinement of Definition 12 as it involves the Substitution Calculus. However, a simplified form was used to generate the Proof Obligation related to the proof of the disjunction of guards in Definition 11 as only predicates were involved.

3.3 Adaptations to Formod Tool

From the B perspective of the Formod tool, the Goal and Domain models of the Formose methodology are instantiated into the B methodology. In order to instantiate these models, an empty B project must be created to manage the Formose B components issued from the instantiation process. After instantiation, the B components can be edited either from the Atelier B or the Formod tools. When the edition of B components is terminated, the process of type checking, proof obligation generation and proof is made in the Atelier B tool.

In order to explore the feedback to be passed to the Formod Tool, in the next sections we develop the first steps of an example issued from a Case Study where we show the instantiation of simple Domain and Goal models into a B model and how this B model is proved in the Atelier B Tool. Then we explore how the Formod tool can display the results of activity of proof.

3.3.1 Goal and Domain Diagrams

The Case Study discussed in this section is a controller computing outputs from the state of its inputs. Therefore, the Main Goal, named *Saturn*, is naturally the intended behavior: "compute outputs as the result of the application of a control function to the inputs of the controller".

The first step to formalize the Main Goal is the elaboration of a Domain Diagram in the Formod tool where the figure 3.1 gives a representation of that diagram.

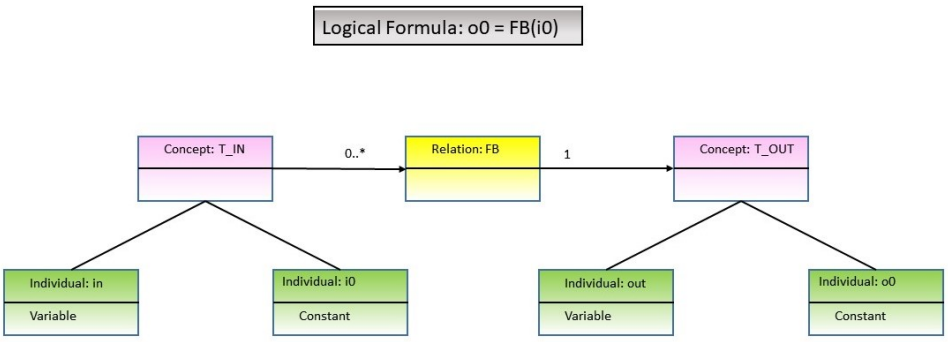

Figure 3.1: Domain Diagram corresponding to the Main Goal

The instantiation of the Main Goal and the Domain Diagram in the B perspective of the Formod Tool gives us two B components. After completion

of the Saturn event, the two components are showed in the figure 3.2

```
SYSTEM Ctx0
SETS
 T_IN;
 T_OUT;
CONSTANTS
 i0,
 o0,
 FB
PROPERTIES
 i0:T_IN &
 o0:T_OUT &
 FB:T_IN -> T_OUT
END
```

```
SYSTEM ProjectL0
SEES Ctx0
VARIABLES
 in,
 out
INVARIANT
 in:T_IN &
 out:T_OUT
INITIALISATION
 in := i0 ||
 out := o0 ||
EVENTS
 Saturn =
  SELECT 0=0
  THEN out:(out = FB(in))
  END
END
```

Figure 3.2: B Model associated to the First Level of the Specification

The B model in figure 3.2 declares two variables `in` and `out` and a constant `FB` modeling a control function and states, in the postcondition of the Main Goal that `out = FB(in)`, that is, the intended meaning of the controller.

The idea in the Second Level of specification is the introduction of "remote" and "local" inputs. The remote inputs are grabbed by input sensors in the application field, and then transmitted to controller to its local inputs to compute the outputs. The Domain Diagram associated to this specification level is sketched in figure 3.3. In this diagram two new variables `in_l` and `in_r` are introduced and a "Logical Formula" `in = in_l` is introduced to state that abstract variable `in` is refined (that is *glued*) with the concrete one `in_l`.

The Second Level of the specification is completed with the Goal Diagrams of figure 3.4. In this diagram, the Main Goal *Saturn* is AND Refined by subgoals *Get* and *Control*. The intended meaning of the subgoal *Get* is to copy the remote inputs `in_r` into the local inputs `in_l` and the meaning of the *Control* subgoal is to compute the outputs by the application of the

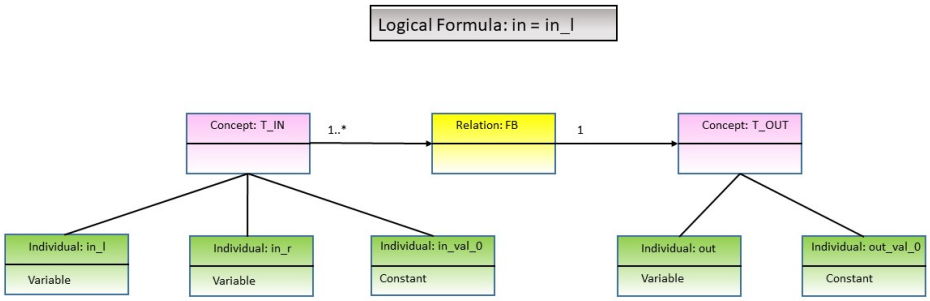

Figure 3.3: Domain Diagram of the Second Level of Specification

control function `FB` to the local inputs `in_l`.

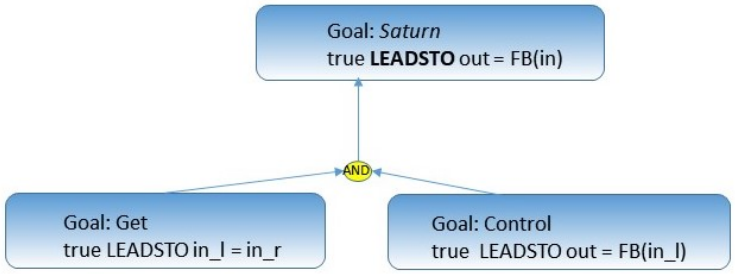

Figure 3.4: Goal Diagram of the Second Level of Specification

After instantiation of these diagrams to the B methodology and the update of the B Components with the body of the subgoals, we get the B component depicted in figure 3.5.

3.3.2 Proof Activity and Feedback to Formod

When the `ProjectL1.ref` component is successfully type checked we can generate the Proof Obligations required to validate the refinement. After the

```
REFINEMENT ProjectL1
REFINES ProjectL0
SEES Ctx0
ABSTRACT_VARIABLES
 in_r,
 in_l,
 out
INVARIANT
 in_r:  T_IN &
 in_l:  T_IN &
 out :  T_OUT &
 in_l= in
```

```
EVENTS
 Control ref_andSaturn =
  SELECT 0 = 0
  THEN
   out:(out = FB(in_l))
  END;
 Get ref_andSaturn =
  SELECT 0 = 0
  THEN
   in_l:(in_l= in_r)
  END
END
```

Figure 3.5: B Model associated to the First Level of the Specification

generation, the component is passed through the automatic provers of the Atelier B. The result of the proof activity can be displayed in the Atelier B Editor as shown in the figure 3.6.

In the figure 3.6 at the left of the window we distinguish two columns. In the first column we have the number of each line in the component and the second column we have the results of the prove activity. The result of the proof activity is displayed in selected lines corresponding to the lines involved in the Proof Obligations. The result of the proof activity is displayed as a fraction x/y where y denotes the number of Proof Obligations where the formula at the concerned line is involved and x, where $x \leq y$, denotes how many Proof Obligations have been successfully proved. In this way, for example, we have at line 8 the fraction 1/2 indicating that variable `in_l` is referenced by two Proof Obligations and only one of proof was successfully proved; at line 14, we have the fraction 0/1 indicating that no Proof Obligation concerning the formula `in_l = in` was proved. When the mouse is over the fraction 1/2 of line 8, we have at the top left window that Proof Obligations `AND_Refinement_Saturn.2` and `AND_Refinement_Saturn.4` are referencing `in_l` variable. Moreover, the green color of Proof Obligation number 2 indicates that the Proof Obligation has been proved whereas the red color in Proof Obligation 4 indicates that it has not been proved. If the mouse is set

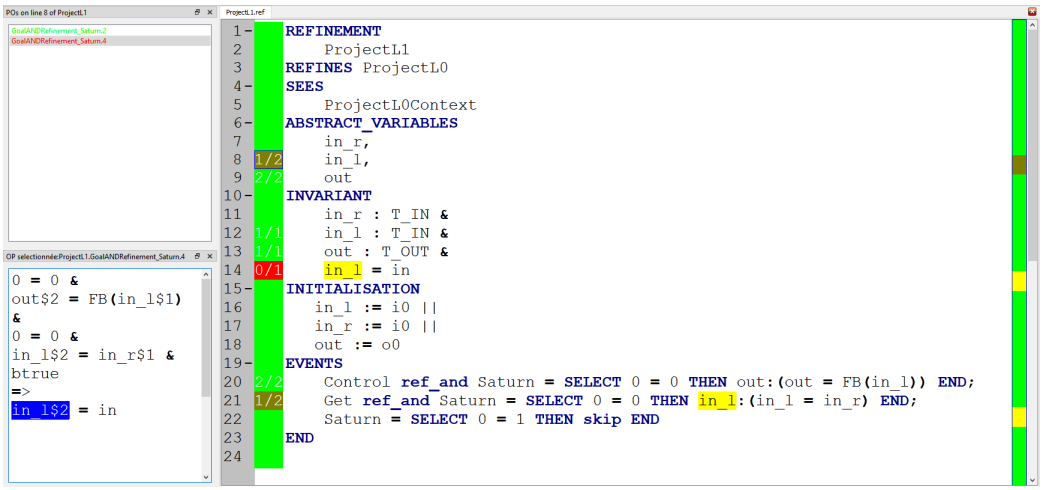

Figure 3.6: Atelier B Editor with Unproved Proof Obligations

over the fraction 0/1 in red color of line 15, we could see that the top left window displays in red color the Proof Obligation `AND_Refinement_Saturn.4`.

The bottom left window of the Atelier B editor displays the Proof Obligation selected in the top left windows. In figure 3.6, the Proof Obligation `AND_Refinement_Saturn.4` is selected and then we can inspect why the Proof Obligation cannot be proved. Without further analysis, the Atelier B Editor shows by its code of colors the status of the Proof Obligations and indicates the formula, by its line number, involved in the proof. In particular, we have at line 15 the proof status 0/1 indicating a non discharged Proof Obligation involving the gluing invariant `in_l = in` and at line 21 the status 1/2 indicating that the event `Get` of the AND Refinement of the Main Goal has an unproved Proof Obligation. At this time the Formod tool has not been feed back with this status, and we propose to add this status information in the Goal and Domain Diagrams as the sources of this error in the form of a check box as showed in figure 3.7.

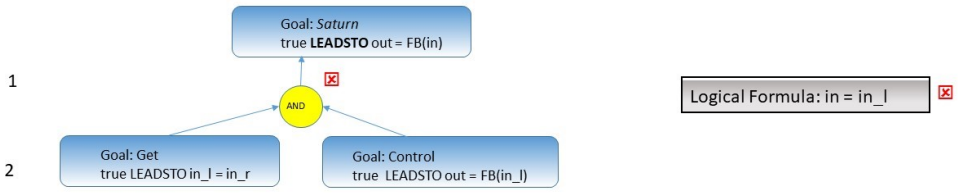

Figure 3.7: Feed Back of Undischarged Proof Obligations

Analyzing the Proof Obligation in the bottom left window of the Atelier B Editor in figure 3.6, we can conclude that we cannot claim that the abstract variable `in` refines the concrete variable `in_l` because this last variable is updated by `Get` event whereas the abstract variable is not modified. It suggests to change the gluing invariant by `in_r = in` in the Domain Diagram and change the AND Refinement by a MILESTONE Refinement in the Goal Diagram. After the update of the B Model, the result of the proof activity in the Atelier B Editor is showed in figure 3.8.

```
1-  REFINEMENT
2       ProjectL1
3   REFINES ProjectL0
4-  SEES
5       ProjectL0Context
6-  ABSTRACT_VARIABLES
7       in_r,
8   1/1 in_l,
9   2/2 out
10- INVARIANT
11      in_r : T_IN &
12  1/1 in_l : T_IN &
13  1/1 out : T_OUT &
14      in_r = in
15- INITIALISATION
16     in_l := i0 ||
17     in_r := i0 ||
18     out := o0
19- EVENTS
20  1/1 Get ref_milestone Saturn = SELECT 0 = 0 THEN in_l:(in_l = in_r) END;
21  2/2 Control ref_milestone Saturn = SELECT 0 = 0 THEN out:(out = FB(in_l)) END;
22      Saturn = SELECT 0 = 1 THEN skip END
23  END
24
```

Figure 3.8: Atelier B Editor with All Proof Obligations Proved

This positive result of the proof activity in the Atelier B tool can now be feed back to the Formod tool as illustrated in figure .

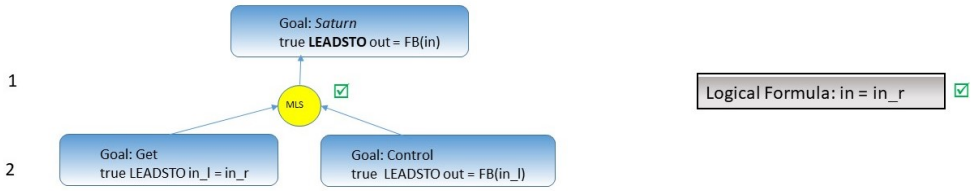

Figure 3.9: Feed Back of Undischarged Proof Obligations

The Atelier B Prover renames systematically the variables in the Proof Obligation, and this rename does not correspond to any formula in the Domain Diagram. For example, let us suppose that we have the AND Refinement of figure 3.4, but we change the gluing invariant of the Domain Diagram

in figure 3.3 by `in_r = in`. In this case, the Atelier B Editor will show the status of the figure 3.10.

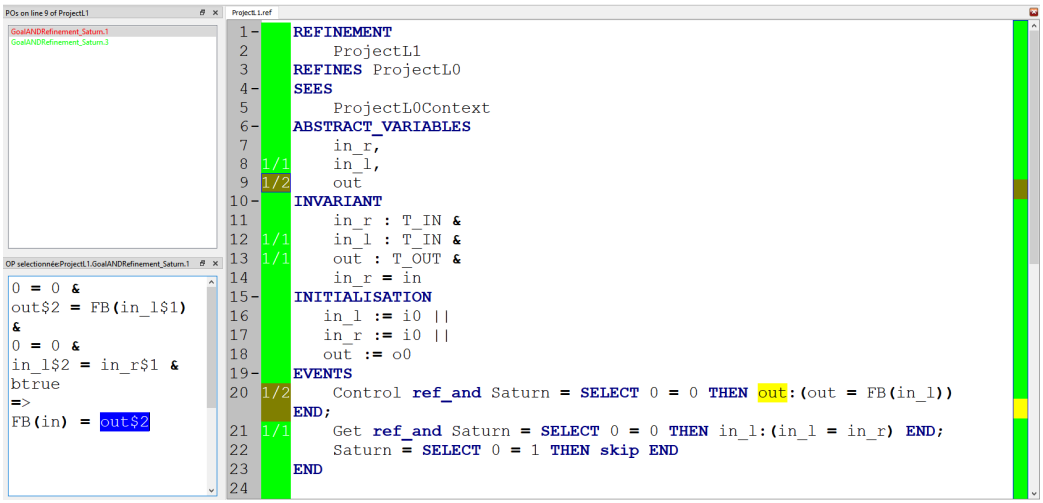

Figure 3.10: Editor with Unproved Proof Obligations in Implicit Invariant

In the Proof Obligation displayed in the bottom left window of the Atelier B Editor, we have `FB(in)=out$2` as the consequent of the Proof Obligation. In this case, the Atelier B prover has renamed the `out` variable in the refinement to `out$2` in order to avoid a clash with the name of the abstract variable and then it introduces an implicit gluing invariant `out = out$2`. According to the Proof Obligation of the AND Refinement, this gluing invariant must be preserved by the parallel composition of `Control` and `Get` events, but this is not possible as we have as antecedent of the Proof Obligation the predicate `out$2=FB(in_l$1)` and there is no way to prove that `in_l$1` is equal to `in`. Therefore, we can signal by a negative mark the Goal Diagram of the AND Refinement, as we do in figure 3.7, but we do not have a corresponding gluing invariant `out = out$2` in the Domain Diagram.

Chapter 4

Conclusions

In this report we have described the theoretical and practical aspects of the instantiation of the Formose methodology into Event-B in order to use the Atelier B tool to prove the coherence of the models. This instantiation has been done at three levels: syntax, semantics and implementation. Moreover, we have explored the way to feed back the Formod tool with the proof results made by the Atelier B.

At the syntactic level, the contents of Formose B components was stated. It was indicated that the component at the highest abstraction level must contain only one event and the `ref` keyword was extended to support the Formose AND, OR and MILESTONE Refinements; the original `ref` keyword is still used to denote data refinements. Although the syntax of the events in the Formosa B component has not been changed with respect to the Event B event syntax, it was precised that any event in Formose B components must be specified by a SELECT substitution having simple or "becomes such that" substitutions in its body.

At the semantic level, the meaning of events in Formose B components was given in an axiomatic way in terms of substitutions. AND, OR and MILESTONE Refinements were defined in terms of parallel, bounded choice and sequential substitution respectively. Moreover, taking into account the constraints imposed in the syntax of events, it was proved by a theorem that the semantics of the AND Refinement, given in terms of parallel substitutions, is equivalent to the non deterministic choice of sequential executions of these substitutions. Finally, the substitutions defining the semantics of the Formose refinements were expanded into Proof Obligations, in order to show the pertinence of these definition with respect to the original work in [3].

At the implementation level, it was explained the adjustments made in the Atelier B to support the instantiation of the Formose methodology. It was indicated the resources to be set in the Atelier B in order to activate the

processing of Formose B components. Moreover, it was explained how the Proof Obligation Generator of the Atelier B was parametrized to generate the Proof Obligation of the Formose refinements. Finally it was explored the feedback of the results of the proof activity into the Formod tool through a simple example.

Appendix A

Proof of Theorem 1

In order to prove Theorem 1 we prove, for any postcondition Q, the following:

$$[G1 \parallel G2]Q \Leftrightarrow [((G1 \mathbin{;} G2) \mathbin{[\!]} (G2 \mathbin{;} G1))]Q$$

The proof shows that both substitutions $[((G1 \mathbin{;} G2) \mathbin{[\!]} (G2 \mathbin{;} G1))]Q$ and $[G1 \parallel G2]Q$ are equivalent to:

$$\begin{aligned} &\forall x', y' \cdot (Guard_1(x) \wedge Guard_2(y) \wedge Post_1(x') \wedge Post_2(y') \\ &\quad \Rightarrow [x := x'][y := y']Q \end{aligned} \tag{A.1}$$

First part of the proof:

$$
\begin{aligned}
& [G1 \parallel G2]Q \\
\Leftrightarrow \quad & \{\ \mathsf{trm}(G1) = \mathsf{trm}(G2) = \mathit{true},\ \parallel \text{property and substitution calculus}\ \} \\
& \mathit{Guard}_1(x) \wedge \mathit{Guard}_2(y) \Rightarrow [x : (\mathit{Post}_1(x)) || y : (\mathit{Post}_2(y))]Q \\
\Leftrightarrow \quad & \{\ y \setminus (\mathit{Guard}_1(x), \mathit{Post}_1(x)) \text{ and } x \setminus (\mathit{Guard}_2(y), \mathit{Post}_2(y))\ \} \\
& \mathit{Guard}_1(x) \wedge \mathit{Guard}_2(y) \Rightarrow [x, y : (\mathit{Post}_1(x) \wedge \mathit{Post}_2(y))]Q \\
\Leftrightarrow \quad & \{\ [z : (P(z))]R \Leftrightarrow \forall z' \cdot (P(z') \Rightarrow [z := z']R)\ \} \\
& \mathit{Guard}_1(x) \wedge \mathit{Guard}_2(y) \Rightarrow \forall x', y' \cdot (\mathit{Post}_1(x') \wedge \mathit{Post}_2(y') \\
& \Rightarrow [x, y := x', y']Q) \\
\Leftrightarrow \quad & \{\ x \setminus y \text{ and } z \setminus (x, y, x', y', Q)\ \} \\
& \mathit{Guard}_1(x) \wedge \mathit{Guard}_2(y) \Rightarrow \forall x', y' \cdot (\mathit{Post}_1(x') \wedge \mathit{Post}_2(y') \\
& \Rightarrow [z := y'][x := x'][y := z]Q) \\
\Leftrightarrow \quad & \{\ x \setminus y',\ z \setminus x',\ z \setminus Q\ \} \\
& \mathit{Guard}_1(x) \wedge \mathit{Guard}_2(y) \Rightarrow \forall x', y' \cdot (\mathit{Post}_1(x') \wedge \mathit{Post}_2(y') \\
& \Rightarrow [x := x'][y := y']Q) \\
\Leftrightarrow \quad & \{\ (x', y') \setminus (\mathit{Guard}_1(x), \mathit{Guard}_2(y)\ \} \\
& \forall x', y' \cdot (\mathit{Guard}_1(x) \wedge \mathit{Guard}_2(y) \wedge \mathit{Post}_1(x') \wedge \mathit{Post}_2(y') \\
& \Rightarrow [x := x'][y := y']Q)
\end{aligned}
$$

Second part of the proof:

$$\forall x', y'.(Guard_1(x) \wedge Post_1(x') \wedge Guard_2(y) \wedge Post_2(y')$$
$$\Rightarrow [x := x'][y := y']Q)$$

$\Leftrightarrow$ { $y \setminus x'$ and $x \setminus y'$ implies $[x := x'][y := y']Q \Leftrightarrow [y := y'][x := x']Q$ }

$$\forall x', y'.(Guard_1(x) \wedge Post_1(x') \wedge Guard_2(y) \wedge Post_2(y')$$
$$\Rightarrow [x := x'][y := y']Q) \wedge$$
$$\forall y', x' : (Guard_2(y) \wedge Post_2(y') \wedge Guard_1(x) \wedge Post_1(x')$$
$$\Rightarrow [y := y'][x := x']Q)$$

$\Leftrightarrow$ { $(x', y') \setminus Guard_1(x)$, $(x', y') \setminus Guard_2(y)$, $y' \setminus Post_1(x')$ and $x' \setminus Post_2(y')$ }

$$(Guard_1(x) \Rightarrow \forall x'.(Post_1(x') \Rightarrow \forall y'.(Guard_2(y) \wedge Post_2(y')$$
$$\Rightarrow [x := x'][y := y']Q))) \wedge$$
$$(Guard_2(y) \Rightarrow \forall y' : (Post_2(y') \Rightarrow \forall x'.(Guard_1(x) \wedge Post_1(x')$$
$$\Rightarrow [y := y'][x := x']Q)))$$

$\Leftrightarrow$ { $(x, y') \setminus Guard_2(y)$, $(y, x') \setminus Guard_1(x)$, $x \setminus Post_2(y')$ and $y \setminus Post_1(x')$ }

$$(Guard_1(x) \Rightarrow \forall x'.(Post_1(x')$$
$$\Rightarrow [x := x'](Guard_2(y) \Rightarrow \forall y'.(Post_2(y') \Rightarrow [y := y']Q)))) \wedge$$
$$(Guard_2(y) \Rightarrow \forall y'.(Post_2(y')$$
$$\Rightarrow [y := y'](Guard_1(x) \Rightarrow \forall x'.(Post_1(x') \Rightarrow [x := x']Q))))$$

$\Leftrightarrow$ { Substitution "becomes such that" and definition of $G1$ and $G2$ }

$$(Guard_1(x) \Rightarrow \forall x'.(Post_1(x') \Rightarrow [x := x'][G2]Q)) \wedge$$
$$(Guard_2(y) \Rightarrow \forall y'.(Post_2(y') \Rightarrow [y := y'][G1]Q))$$

$\Leftrightarrow$ { Substitution "becomes such that" and definition of $G1$ and $G2$ }

$$([G1][G2]Q) \wedge ([G2][G1]Q)$$

$\Leftrightarrow$ { Substitution $[\!]$ }

$$[(G1 \,;\, G2) \,[\!]\, (G2 \,;\, G1)]Q$$

End Of Proof

Appendix B

Proof of Derivation 2.1

Let a, c_1 and $c_2]$ be the following events:

$$
\begin{aligned}
a &= \text{SELECT } Guard_a(x) \text{ THEN } x : (Post_a(x)) \text{ END} \\
c_1 &= \text{SELECT } Guard_{c_1}(y_{c_1}) \text{ THEN } y_{c_1} : (Post_{c_1}(y_{c_1})) \text{ END} \\
c_2 &= \text{SELECT } Guard_{c_2}(y_{c_2}) \text{ THEN } y_{c_2} : (Post_{c_2}(y_{c_2})) \text{ END}
\end{aligned}
$$

The Proof Obligation $C \wedge I \wedge D \wedge J \Rightarrow wp\_seq(E, \neg[a]\neg J)$ is equivalent to:

$$
\begin{aligned}
&\forall y'_{c_1}, y'_{c_2} \cdot (C \wedge I \wedge D \wedge J \wedge Guard_{c_1} \wedge Post_{c_1}(y'_{c_1}) \\
&\quad \wedge [y_{c_1} := y'_{c_1}] Guard_{c_2} \wedge [y_{c_1} := y'_{c_1}] Post_{c_2}(y'_{c_2}) \\
&\quad \Rightarrow \\
&\quad Guard_a \wedge \exists x' \cdot (Post_a(x') \wedge [y_{c_1} := y'_{c_1}][y_{c_2} := y'_{c_2}][x := x']J))
\end{aligned}
$$

Proof

$$
\begin{aligned}
& C \wedge I \wedge D \wedge J \\
\Rightarrow & \\
& wp\_seq([c_1, c_2], \neg[a]\neg J) \\
\Leftrightarrow & \qquad \{ \text{ Definition of } wp\_seq \text{ case non empty list } \} \\
& [c_1] wp\_seq([c_2], \neg[a]\neg J) \\
\Leftrightarrow & \qquad \{ \text{ Definition of } wp\_seq \text{ case non empty list } \} \\
& [c_1][c_2] wp\_seq([], \neg[a]\neg J) \\
\Leftrightarrow & \qquad \{ \text{ Definition of } wp\_seq \text{ case empty list } \} \\
& [c_1][c_2]\neg[a]\neg J \\
\Leftrightarrow & \qquad \{ \text{ Substitution calculus } \neg[a]\neg J \ \} \\
& [c_1][c_2] Guard_a \wedge \exists x' \cdot (Post_a(x') \wedge [x := x']J) \\
\Leftrightarrow & \qquad \{ \text{ Substitution Calculus } [c_2]Q \ \} \\
& [c_1] \forall y'_{c_2} . (Guard_{c_2} \wedge Post_{c_2}(y'_{c_2}) \Rightarrow \\
& Guard_a \wedge \exists x' \cdot (Post_a(x') \wedge [y_{c_2} := y'_{c_2}][x := x']J)) \\
\Leftrightarrow & \qquad \{ \text{ Substitution Calculus } [c_1]Q \ \} \\
& \forall y'_{c_1}, y'_{c_2} . (Guard_{c_1} \wedge Post_{c_1}(y'_{c_1}) \\
& \wedge [y_{c_1} := y'_{c_1}] Guard_{c_2} \wedge [y_{c_1} := y'_{c_1}] Post_{c_2}(y'_{c_2}) \\
& \Rightarrow Guard_a \wedge \exists x' \cdot (Post_a(x') \wedge [y_{c_1} := y'_{c_1}][y_{c_2} := y'_{c_2}][x := x']J))
\end{aligned}
$$

End Of Proof

Bibliography

[1] J.-R. Abrial, *The B-Book, Assigning Programs to Meanings*, Cambridge University Press, 1996.

[2] J.-R. Abrial, *Modelling in Event-B, System and Software Engineering*, Cambridge University Press, 2010.

[3] A. Matoussi, F. Gervais, R. Laleau, *An Event-B formalization of KAOS goal refinement patterns*, Technical Report TR-LACL-2010-1, Université Paris 12, January 2010.

[4] http://relaxng.org/compact-20021121.html, November 2002.