Electronic Notes in Theoretical Computer Science 51 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume51.html 20 pages

A methodological process for the design of a large system: two industrial case-studies.

Nestor Lopez, Marianne Simonot, Véronique Viguié Donzeau-Gouge

Cedric Research Laboratory, Conservatoire National des Arts et Métiers, 292 Rue Saint-Martin, 75141 Paris Cedex, France.
email: (donzeau, lopezne, simonot)@cnam.fr

Abstract

This paper presents two examples taken from industrial case-studies that have been specified using an event system approach. Component specifications, taking the form of pre-post formula, have been derived. Constraints which ensure the correctness of the whole process are given.

1 Introduction

Goals: Our purpose is to present a methodological process which can be used for an *entire* system development. Our goal is to define a system according to the properties it must satisfy and express them in a formal way. This process leads us to build two sorts of mathematical models : event models and operational models. They define the formal specification of the entire system. These models are proved to be consistent and the correctness of the whole process is ensured, i.e. the correctness of the implementation of each component and the correctness of the implementation of the entire system.
Context: The applications we have in mind heavily interact with their environment. They may use hardware or software components that already exist.
Addressed issues: A first issue is the big quantity of information required by this kind of specification. It is therefore necessary to introduce a certain hierarchy. This is done in two ways: by initially defining the more abstract aspects and gradually focusing upon details; by sectorizing each specific category of property (properties which describe the interactions between components and with the environment, properties which are local to one component, like functional properties). We use refinement techniques [Back 88], [Morg 90], [Abr 96-1], [But 96] to support the abstraction i.e. to take account of more details and express some design decisions.
Another issue is how to specify interactions of a system with its environment. The trend is to build a closed model [Abr 96-2] which includes the whole environment. However, this model can be extremely complex. In our approach,

we develop a closed model which specifies only the interactions of the system with the environment.
Furthermore, refining a closed system might lead to a program failing to implement the specification we have in mind [Lop,Sim,Don 00].
Choices: The formal specification language must allow the expression of non-deterministic specifications and has to be supported by efficient proof assistants. We neither combine different logics nor introduce new ones. We largely rely on existing proven methods backed by tools. We adopt the set theory as well as the refinement technics as they are encapsulated in the actual tools [Atelier B], [B Toolkit].
Process: We start from an informal specification written in a natural language which describes all the properties the system must satisfy. Domain experts participate in the elaboration and validation of this document. When system details are introduced, some implementation choices can be made: for example re-using existing devices with well-defined interfaces.
Then we build, step by step, an event model which only describes the interactions of the system with its environment and between the different components. It expresses the hypothesis that we make on the environment. We only keep from the initial properties those that express interactions. This process is controlled by event refinement techniques as defined in [Lam,Sha 90], [But 96], [Abr 96-2], [Back,Kur 88].
After this, the event model is transformed into a shared module which is used in the specifications of the components. This module provides the external and internal communications mechanisms of the system. It is called *interface module*. Roughly speaking, by module we mean a set of program specifications which can be refined to sequential implementations. Notions like shared modules or logical variables [1] [Aba,Lam 88] are not supported by the actual tools.
Finally, the specification of each component is written separately. It incorporates the properties of the informal specification related to only to the component, and which therefore, have not yet been taken into account. We obtain *the specification* of the component. These specifications are used as the starting point for the usual refinement process for sequential programs [Back 88], [Abr 96-1].

The correctness of the whole process is presented in [Lop 02]. In this article, the conditions that ensure this correctness will only lightly be touched upon.
In the following part of this paper, we treat two examples extracted from industrial case-studies which are specified and partially proven using [Atelier B]. They are presented in the syntax of this tool and some comments are added to help the reader.

However, the B method does not include two important notions we use:

[1] Logical variables are also called auxiliary variables.

shared modules and auxiliary variables. These notions are introduced and formally defined in [Lop 02]. They tend to extend the utilization of the B method to concurrent systems and distributed systems.

The first example illustrates the way we use to specify a large system.

The second example illustrates the way we use to derive components specifications. The event specification of this example is detailed in [Lop,Sim,Don 00].

2 An event based specification: The passenger exchange function of the Météor Metro Line

Meteor is a new and entirely automated metro line in Paris. The line is formed by trains and arrival/departure platforms. Trains and platforms have doors. The passenger exchange function [2] must ensure the passengers' safety. The safety is built upon four independant principles that can be expressed as follows:

P0 An exchange cannot generate an unsafe situation.

P1 To guarantee the passengers' safety, it is necessary to ensure the train's immobility during these exchanges.

P2 Passengers are safe if these areas (i.e. trains and platforms) are closed universes.

P3 If a dangerous situation occurs, the passengers should have the possibility of leaving the train -at any time- for a safe environment.

Several kinds of passenger exchanges can be identified: the usual ones, the emergency passenger evacuations which happen when a dangerous situation is detected. Such an evacuation has to be launched as soon as a danger occurs [3].

2.1 Recalls

Events. An event specification takes the form $\exists a.\ G(x,a) \wedge A(x,a,x')$. The guard contraints the occurrence of the event and the action specifies its effect. The a variable allows to express the external non-determinism i.e. external values provided by the environment. The guard $G(x,a)$ defines the states in which the event can be observed, the action $A(x,a,x')$ relates system states before and after the observation. The concrete syntax of an event (in [Abr 96-2]) is $\textsf{any}\ a\ \textsf{where}\ G(x,a)\ \textsf{then}\ S_{x,a}$ where $S_{x,a}$ is the generalized substitution which is a translation of $A(x,a,x')$. For instance $x' = x \cup a$ is translated into $x := x \cup a$.

Event Refinement. Let $E_1(x,x')$ be of the form $\exists a.\ G_1(x,a) \wedge A_1(x,a,x')$ and $E_2(w,w')$ be of the form $\exists a.\ G_2(w,a) \wedge A_2(w,a,w')$, the specifications of

[2] This case-study has been proposed by the RATP and Matra Transport International within the framework of an agreement between the CNAM computer research center (lab. CEDRIC), RATP, MTI(SIEMENS), and STERIA under the contract DJ.0246/98.

[3] This presentation is a simplified version of this problem -a more detailed description is provided in [Lop 02], [Lop 99].

two events. They are defined on two different state spaces. Let $J(x, w)$ be a total relation defined between these two spaces, $E_1 \sqsubseteq_J E_2 =_{def}$
$\forall x, w, w'. \exists a. (J(x, w) \wedge G_2(w, a) \wedge A_2(w, a, w')) \Rightarrow \exists a, x'. G_1(x, a) \wedge A_1(x, a, x') \wedge J(x', w')$ This formula expresses the fact that all state changes observed with E_2 are also observed with E_1.

Event System. An event system $Ss = (x, IS, C, E_i)$ is formed by a shared variable x that represents the system state, an invariant IS that expresses static properties of the system, the specification C that initializes the system and the set of events E_i for $i \in 1..n$ that defines the dynamics. Such systems are transition systems, whose initial states are defined by the initialization, and whose transitions are defined by the events.

System Consistency. An event system is said to be consistent if the invariant is established by the initialization and preserved by each event.

System Refinement. Let S_1 and S_2 be two event systems, let $J(x, w)$ a total relation between their variables, S_1 is refined by S_2 ($S_1 \sqsubseteq_J S_2$) if each event of S_2 is a refinement of the corresponding event of S_1, and so it is for the initializations. The proof obligations are generated by the actual tools.

Adding new events. According to [Abr 96-2], [Lop 02], it is possible during a refinement:

- to add new events. For this, we have just to assume that the new event is present in the abstract system with $true \wedge skip$ as a specification. $skip$ means that the new event does not modify any variable of the abstract system. This means that when a new variable is introduced, the refinement must include all the events which modify this variable. This constrains the way events are introduced.
- to split an event. This decomposition allows us to observe an event with greater precision. In order to do this, it suffices to assume that in the abstract event system, the abstract event E is duplicated as many times as necessary and that each event in the decomposition must be a refinement of E.

Once these constraints are satisfied, the order in which events are introduced during the refinement does not matter. The final event system does not depend on the way in which refinements are done.

2.2 First model

This model introduces the main notions of the system i.e. the trains tt, the platforms qq and their possible states (variables): unsafe (td, qd) or safe ($tt - td$, $qq - qd$). At this level, only the property P0 can be expressed.

We can observe five events:
- `UnsafeTrain`, `SafeTrain` which modify the variable td.
- `UnsafePlatf`, `SafePlatf` which modify the variable qd.
- `Transfer` which has no effect upon the environment and can be activated at any time.

```
SYSTEM Meteor1.1
SETS   tt; qq   /* Trains, platforms of the line */
VARIABLES   td, qd   /* Unsafe trains and platforms */
INVARIANT   td ⊆ tt  ∧  qd ⊆ qq
INITIALISATION   td, qd := ∅, ∅ /* The line is safe */
EVENTS
Transfer ≙
  SELECT TRUE   /* An exchange is always observable. */
  THEN   SKIP   /* The safety is not modified (P0). */
  END;
UnsafeTrain ≙ /* A safe train becomes unsafe */
  ANY t1 WHERE t1 ∈ tt ∧ t1 ∉ td
  THEN   td := td ∪ {t1}
  END;
SafeTrain ≙ /* An unsafe train becomes safe */
  ANY t1 WHERE t1 ∈ td
  THEN   td := td − {t1}
  END;
UnsafePlatf ≙ /* q1 becomes unsafe */
  ANY q1 WHERE q1 ∈ qq ∧ q1 ∉ qd
  THEN   qd := qd ∪ {q1}
  END;
SafePlatf ≙ /* q1 becomes safe */
  ANY q1 WHERE q1 ∈ qd
  THEN   qd := qd − {q1}
  END
```

Note that transfer can occur at any time and not only in a safe situation. Note also that the property P0 is expressed by the fact that the action of `Transfer` does not modify the system state.

The proofs of consistency of this model are generated and discharged by the Atelier B. We can prove additional properties using this model. For example, deadlock freeness is implied by the fact that under the invariant, the disjunction of the guards is always true. We can also prove properties of event traces the system should allow: for instance, `UnsafeTrain` may be observed at any time.

2.3 Second model

This model introduces the notion of *locked train* needed to express property P1. A locked train is totally stopped and to start moving, some actions need to be executed before.

(i) The variable ti models the set of locked trains. Hence, two new events `LockedTrain` and `UnlockedTrain` are introduced, which modify this variable.

(ii) At this level, we want to observe separately two kinds of exchanges : normal exchanges and other cases. To differentiate these two cases, we categorize within the subset of locked trains those trains which are doing a passenger exchange in the normal conditions: variable ttv. This partition is expressed by decomposing the event `Transfer` of the first model into two events `EndPassExch` (normal exchange) and `Transfer1`. The event `Transfer1` allows to observe all the other exchanges. It can be activated at any time for any train into $ti - ttv$.

An event, `StartPassExch`, must be added to modify ttv. It allows to observe the beginning of a normal passenger exchange.

(iii) A normal exchange is done when a train is in ttr. Hence, ttv only includes trains which satisfy this property. To express this, we introduce a partial function t_q from trains to platforms, the variable qtv denotes the set of trains which are doing a passenger exchange in the normal conditions. `StartPassExch` is activated if the train is stopped in front of a platform, and hence, t_q appears in its guard. The events which modify this variable are introduced: `DepartureFromPlatf` and `ArrivalToPlatf`.

The property P1 is an invariant of the system. The first four events of the system $Meteor1.1$ are included in addition to the events that we have just introduced. The first four events are still not synchronized. The new events are synchronized as follows:

`LockedTrain` → `Transfer1` → `UnlockedTrain`
`ArrivalToPlatf` → `LockedTrain` → `StartPassExch` → `EndPassExch` → `UnlockedTrain` → `DepartureFromPlatf`.
`LockedTrain` → `UnlockedTrain`

The assurance of this synchronization is obtained by proof obligations involving only guards.

SYSTEM $Meteor1.2$

VARIABLES

td, qd, /* Unsafe trains and platforms */

t_q, /* Trains in front of platforms */

ttr, /* Trains doing an exchange */

ttv, **qtv**, /* Trains and platforms doing a passenger exchange */

ti /* Locked trains */

INVARIANT

$td \subseteq tt \ \wedge \ qd \subseteq qq \ \wedge$	
$\mathbf{t\_q} \in \mathbf{tt} \rightarrowtail\!\!\!\!\!+\ \mathbf{qq} \ \wedge$	$\mathbf{ttv} \subseteq \mathbf{ttr} \subseteq \mathbf{ti} \subseteq \mathbf{tt} \wedge$ /* includes **(P1)** */
$\mathbf{ttv} \in \mathbf{dom}(\mathbf{t\_q}) \wedge$	$\mathbf{qtv} = \mathbf{t\_q}[\mathbf{ttv}] \wedge$

INITIALISATION $t\_q, ttr, ttv, td, ti, qtv, qd := \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset$

EVENTS

UnsafeTrain $\hat{=} \ldots$	UnsafePlatf $\hat{=} \ldots$
SafeTrain $\hat{=} \ldots$	SafePlatf $\hat{=} \ldots$

StartPassExch $\hat{=}$ /* $t1$ starts a Transfer */
ANY $t1$ WHERE
$t1 \in ti \wedge t1 \notin ttr \wedge$
$t1 \in dom(t\_q) \wedge t\_q(t1) \notin qtv$
THEN $ttv, qtv, ttr := ttv \cup \{t1\},$
$qtv \cup t\_q[t1], ttr \cup \{t1\}$
END;

EndPassExch $\hat{=}$ /* $t1$ finishes a Transfer */
ANY $t1$ WHERE
$t1 \in ttv$
/* $t1$ is in Transfer */
THEN $ttv, qtv, ttr := ttv - \{t1\},$
$qtv - t\_q[t1], ttr - \{t1\}$
END;

Transfer1 $\hat{=}$
ANY $t1$ WHERE $t1 \in ti \wedge t1 \notin ttv$ /* $t1$ is not in Transfer */
THEN $SKIP$ /* end of the Transfer */
END;

LockedTrain $\hat{=}$
/* An unlocked train becomes locked */
ANY $t1$ WHERE $t1 \in tt \wedge t1 \notin ti$
THEN $ti := ti \cup \{t1\}$
END;

UnlockedTrain $\hat{=}$
/* A locked train becomes unlocked */
ANY $t1$ WHERE $t1 \in ti \wedge t1 \notin ttr$
THEN $ti := ti - \{t1\}$
END;

ArrivalToPlatf $\hat{=}$
/* A (free and moving)train arrives to a (free) platform */
ANY $t1, q1$ WHERE $t1 \in tt \wedge$
$q1 \in qq \wedge t1 \notin dom(t\_q) \wedge$
$q1 \notin ran(t\_q) \wedge t1 \notin ti$
THEN $t\_q := t\_q \cup \{t1 \mapsto q1\}$
END;

DepartureFromPlatf $\hat{=}$
/* A train is leaving a platform */
ANY $t1$ WHERE
$t1 \in dom(t\_q) \wedge$
$t1 \notin ti$
THEN $t\_q := \{t1\} triangleleftrt\_q$
END;

2.4 Third model

In this model, we observe the normal openings of the doors. This model introduces the safety property P2 which can be rewritten as follows: (P2.1) a train which is not closed is in a passenger exchange state, (P2.2) a platform which is not closed is a platform on which a passenger exchange is currently taking place or which is in a safe state. We introduce open trains to and open platforms qo as well as four events `OpenedTrain`, `ClosedTrain`, `OpenedPlatf`, `ClosedPlatf`, which modify the values of these variables. These events detail the transition `StartPassExch` $\rightarrow$ `EndPassExch` by introducing the events which manage the opening and shutting of the train and platform doors. The guards of these new events and the modification of the guard of `EndPassExch` define the following synchronization:

$$\texttt{StartPassExch} \rightarrow \left| \begin{array}{c} \texttt{OpenedTrain} \\ \\ \texttt{OpenedPlatf} \end{array} \right| \rightarrow \left| \begin{array}{c} \texttt{ClosedTrain} \\ \\ \texttt{ClosedPlatf} \end{array} \right| \rightarrow \texttt{EndPassExch}$$

The doors of a platform are opened only if this platform is safe. Here, we introduce the variable qs and, therefore, the two events `SafeZone` and

`UnsafeZone` which allow to modify this variable. These events detail the opening of a platform. `SafeZone` must occur before `OpenPlatf` and `UnsafeZone` before `ClosedPlatf`.

SYSTEM $Meteor1.3$

VARIABLES

td, qd, t_q, ttr, ttv, qtv, ti,

to qo, /* Opened trains and platforms */

qs, /* safe platform zones */

INVARIANT

$td \subseteq tt \wedge qd \subseteq qq \wedge$

$ttv \subseteq ttr \subseteq ti \subseteq tt \wedge$ /* includes **(P1)** */

$\mathbf{to} \subseteq \mathbf{ttr} \subseteq \mathbf{tt} \wedge$ /* includes **(P2.1)** */

$\mathbf{qs} \subseteq \mathbf{qq} \wedge$

$t\_q \in tt \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow qq \wedge$

$ttv \in dom(t\_q) \wedge qtv = t\_q[ttv] \wedge$

$\mathbf{qo} \subseteq \mathbf{qq} \wedge$

$\mathbf{qo} \subseteq \mathbf{qtv} \cup \mathbf{qs}$ /* **(P 2.2) */**

INITIALISATION $t\_q, to, ttr, ttv, td, ti, qo, qtv, qd, qs := \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset$

EVENTS

UnsafeTrain $\widehat{=}$ ...

SafeTrain $\widehat{=}$ ...

StartPassExch $\widehat{=}$ ...

Transfer1 $\widehat{=}$ ...

LockedTrain $\widehat{=}$ ...

ArrivalToPlatf $\widehat{=}$ ...

OpenedTrain $\widehat{=}$ /* $t1$ becomes opened */

ANY $t1$ WHERE $t1 \in ttr \wedge t1 \notin to$

THEN $to := to \cup \{t1\}$

END;

OpenedPlatf $\widehat{=}$ /* $q1$ becomes opened */

ANY $q1$ WHERE $q1 \in qtv \cup qs \wedge q1 \notin qo$

THEN $qo := qo \cup \{q1\}$

END;

SafeZone $\widehat{=}$ /* $q1$ zone becomes safe */

ANY $q1$ WHERE $q1 \in qq \wedge q1 \notin qs$

THEN $qs := qs \cup \{q1\}$

END;

UnsafePlatf $\widehat{=}$ ...

SafePlatf $\widehat{=}$ ...

EndPassExch $\widehat{=}$

ANY $t1$ WHERE $t1 \in ttv \wedge$ /* $t1$ is in a transfer: */

$t1 \notin to \wedge t\_q(t1) \notin qo$ /* train and platform closed */

THEN $ttv, qtv, ttr := ttv - \{t1\}$,

$qtv - t\_q[t1], ttr - \{t1\}$ /* End of the transfer */

END;

UnlockedTrain $\widehat{=}$ ...

DepartureFromPlatf $\widehat{=}$ ...

ClosedTrain $\widehat{=}$ /* $t1$ becomes closed */

ANY $t1$ WHERE $t1 \in to$

THEN $to := to - \{t1\}$

END;

ClosedPlatf $\widehat{=}$ /* $q1$ becomes closed */

ANY $q1$ WHERE $q1 \in qo$

THEN $qo := qo - \{q1\}$

END;

UnsafeZone $\widehat{=}$ /* $q1$ zone becomes unsafe */

ANY $q1$ WHERE $q1 \in qs$

THEN $qs := qs - \{q1\}$

END;

2.5 Fourth model

In this model, we introduce the property P3 and observe an emergency passenger evacuation. When the passengers have to leave their train, the state of this train must allow the opening of its doors. An access to a circulation zone which is safe must also be provided. All of the trains which are in an emergency evacuation are modeled by the variable teu. We have two events which allow us to observe the beginning and the end of an emergency evacuation: `StartEva` and `EndEva`.

New notions: the evacuation zones ze, and among them, the subset of evacuation zones which are safe zs. We must be able to locate a train with respect to the evacuation zones; this is the role of the function t_z. The events which modify t_z (`ArrivingZoneEva` and `LeavingZoneEva`) and zs, (`SafeZoneEva` and `UnsafeZoneEva`) are introduced. The localization of platforms into the evacuation zones is defined by the relation q_z.

As previously, the event `Transfer1` is decomposed into two events : `EndEva` and `Transfer2`. `EndEva` models the end of an emergency evacuation, the event `StartEva` puts the train in a state of evacuation (teu is modified).

And just like the previous systems, the guards ensure the expected synchronization.

SYSTEM $Meteor1.4$

SETS **ze** /* Evacuation zones */

CONSTANTS **q_z** /* Localization of platforms into zones */

PROPERTIES $\mathbf{q\_z} \in \mathbf{qq} \leftrightarrow \mathbf{ze}$

VARIABLES

td, qd, t_q, ttr, ttv, qtv	ti,$t0$, $q0$, qs,
t_z /* Localization of trains */	**teu**, **zs** /* Urgent evacuations and evacuation zones */

INVARIANT

$td \subseteq tt \wedge qd \subseteq qq \wedge t\_q \in tt \rightarrowtail\!\!\!\!\!\!\rightarrow qq \wedge$	$ttv \subseteq ttr \subseteq ti \subseteq tt \wedge$ /* includes **(P 1)** */
$ttv \in dom(t\_q) \wedge qtv = t\_q[ttv] \wedge$	$to \subseteq ttr \subseteq tt \wedge$ /* includes **(P 2.1)** */
$qo \subseteq qq \wedge$	$qs \subseteq qq \wedge$
$qo \subseteq qtv \cup qs$ /* **(P2.2) */**	$\mathbf{t\_z} \in \mathbf{tt} \leftrightarrow \mathbf{ze} \wedge \mathbf{dom(t\_z)} = \mathbf{tt} \wedge$
$\mathbf{t\_q^{-1}}; \mathbf{t\_z} \subseteq \mathbf{q\_z}$	$\mathbf{teu} \subseteq \mathbf{ttr} \wedge$
$\mathbf{ttv} \cap \mathbf{teu} = \emptyset \wedge$	$\mathbf{teu} \subseteq \mathbf{td} \wedge$
$\mathbf{zs} \subseteq \mathbf{ze} \wedge$	$\mathbf{teu} \subseteq \mathbf{t\_z^{-1}}[\mathbf{zs}]$ /* **(P 3.1) */**
$\mathbf{ran(q\_z)} = \mathbf{ze}$ /* **(P 3.2) */**	

INITIALISATION $t\_q, t\_z, to, ttr, ttv, teu, td, ti, qo, qtv, qd, qs, zs$:

$t\_q = \emptyset \wedge t\_z \in tt \leftrightarrow ze \wedge dom(t\_z) = tt \wedge to = \emptyset \wedge ttr = \emptyset \wedge ttv = \emptyset \wedge teu = \emptyset \wedge td = \emptyset \wedge$
$ti = \emptyset \wedge qo = \emptyset \wedge qtv = \emptyset \wedge qd = \emptyset \wedge qs = \emptyset \wedge zs = \emptyset$

EVENTS

UnsafeTrain $\widehat{=} \ldots$	UnsafePlatf $\widehat{=} \ldots$	SafeTrain $\widehat{=} \ldots$	SafePlatf $\widehat{=} \ldots$
StartPassExch $\widehat{=} \ldots$	EndPassExch $\widehat{=} \ldots$	LockedTrain $\widehat{=} \ldots$	UnlockedTrain $\widehat{=} \ldots$
ArrivalToPlatf $\widehat{=} \ldots$	DepartureFromPlatf $\widehat{=} \ldots$	OpenedTrain $\widehat{=} \ldots$	ClosedTrain $\widehat{=} \ldots$
OpenedPlatf $\widehat{=} \ldots$	ClosedPlatf $\widehat{=} \ldots$	SafeZone $\widehat{=} \ldots$	UnsafeZone $\widehat{=} \ldots$

StartEva $\widehat{=}$ /* $t1$ starts an evacuation */
ANY $t1$ WHERE $t1 \in ti \wedge t1 \in td \wedge$
$t1 \notin ttr \wedge t\_z[t1] \subseteq zs$
THEN $teu, ttr := teu \cup \{t1\}, ttr \cup \{t1\}$
END;

EndEva $\widehat{=}$ /* an evacuation is finished */
ANY $t1$ WHERE $t1 \in teu \wedge$
$t1 \notin to \wedge t1 \notin td$
THEN $teu, ttr := teu - \{t1\}, ttr - \{t1\}$
END;

Transfer2 $\hat{=}$
ANY $t1$ WHERE $t1 \notin ttv \cup teu \ \wedge t1 \in ti$ /* $t1$ is not in transfer */
THEN $SKIP$ /* The transfer is finished */ END;

ArrivingZoneEva $\hat{=}$ /* a train arrives in a zone */
ANY $t1, z1$ WHERE $t1 \in tt \wedge$
$t1 \notin ti \ \wedge z1 \notin t\_z[t1] \ \wedge z1 \in ze \ \wedge z1 \notin zs$
THEN $t\_z := t\_z \cup \{t1 \mapsto z1\}$ END;

SafeZoneEva $\hat{=}$ /* $z1$ becomes safe */
ANY $z1$ WHERE $z1 \in ze \ \wedge z1 \notin zs$
THEN $zs := zs \cup \{z1\}$ END;

LeavingZoneEva $\hat{=}$ /* $t1$ leaves $z1$ */
ANY $t1, z1$ WHERE $t1 \in tt \wedge$
$t1 \notin ti \ \wedge z1 \in t\_z[t1] \ \wedge z1 \notin zs$
THEN $t\_z := t\_z - \{t1 \mapsto z1\}$ END;

UnsafeZoneEva $\hat{=}$ /* $z1$ becomes unsafe */
ANY $z1$ WHERE $z1 \in zs \ \wedge z1 \notin t\_z[td]$
THEN $zs := zs - \{z1\}$ END;

The other kind of passenger exchanges are introduced using a similar process.

2.6 Conclusion

What we obtain at this point is a formal description of the required behaviour of the system. It takes the form of an event model.

This model describes an automaton representing the external behaviour. Some safe properties model specify a part of the static of the system, the others specify the dynamic. Here, P1, P2 and P3 are static, P0 is dynamic. Static properties are embedded in the invariant. The final specification has a lot of events which have been gradually introduced, thanks to the event refinement mechanism. This top down analysis leads us to introduce system details, though the order of insertion between properties which are at a similar level in the abstraction is more or less arbitrary. For instance, the introduction of the emergency evacuation could be done before or at the same time as the normal exchange. Furthermore, the refinement process allows us to split the proofs of consistency in the final event system. The synchronization is distributed through the guards.

This application has been developed under AtelierB. Almost all the generated proof obligations have been automatically discharged.

However, we do not yet have the internal architecture of the system. The isolation of each component has not yet been done and, moreover, for each component, we do not have an implementation. In order to illustrate the whole process, we will present a second application which has been taken to the point of implementation.

3 Flight Warning System (FWS): The event model

3.1 Introduction

This case-study - the flight warning system (FWS) used in the airbus A340 aircraft- was proposed by the Aerospatiale Company. FWS's role is to monitor aeroplane subsystems. When an abnormal situation appears, FWS must decide on when, and how, to emit warning signals. One of the difficulties of this study comes from an imposed constraint on the final architecture: the system must be formed by two cyclic concurrent processes. The first one, `P1`, deals with examining all the alarms. If an alarm is detected "present", this process confirms it after a pre-defined period of time. If an alarm is detected 'absent', it removes it from the set of confirmed alarms. The second process, `P2`, has also to examine all the alarms. It is charged with emitting signals associated to the alarms which have been confirmed by the first process.

Since the two cycles are concurrent it is not possible to specify the application as being the following sequence: `P1 ; P2`. We have to allow for the fact that an alarm is examined by `P2` before `P1`, therefore, in this case, no signal will be emitted. Let us imagine one alarm a being activated at a time `t`. Six cases are possible:

(i) `t ...P1 ...P2`: the alarm a 'happens' before its examination by `P1` and `P2`. The alarm is treated by `P1` during its running cycle and is treated by `P2`. A signal is emitted [4].

(ii) `t ...P2 ...P1`: the alarm a happens before its examination by `P1` and `P2`. The alarm is treated by `P2`. As a has not yet been examined by `P1`, no signal is emitted during the running cycle of `P2`. A signal for a will be emitted during the next cycle of `P2` except if `P1`, during its own next cycle, examines a before `P2` and detects that a is absent.

(iii) `P1 ...P2 ...t` or `P2 ...P1 ...t`: a will be treated by the two processes during their next cycles.

(iv) `P1 ...t ...P2`: same as situation 3.

(v) `P2 ...t ...P1`: during its running cycle, `P1` examines and confirms a. `P2` will treat a during its next cycle (unless `P1` detects absence of a).

3.2 The Event model

Here, we present a simplified version of the case study which does not cover details concerning the flying phases and the signal composition. A complete description is given in [Lop 96-1], [Lop 96-2].

The role of the event specification is to model the interactions between components and the environment. To do so, we need the set of alarms WW and the set of signals Ss, and the following variables:
wp: the set of emitted alarms,
se: the set of emitted signals,
wc: the set of confirmed alarms,
$We1$, $Wp1$: the alarms examined and detected *present* by `P1`
$We2$, $Wc2$ the alarms examined and detected *confirmed* by `P2`.
$Num1$: counter. Number of executed cycles of `P1`.

[4] In the real system the signal can be emitted but there is no guarantee of this as the aeroplane can be in a state where many alarms are activated, so there is a competition [Lop 96-1].

$Num2$: counter. Number of executed cycles of `P2`.

With this model, we observe the following:
the beginning of presence (`NewWarning`) and the end of presence (`EndWarning`) of a warning situation (interaction with the environment),
the beginning of the emission (`EmittedSignal`) and the end (`EndSignal`) of a signal (interaction between `P2` and the environment),
the confirmation (`ConfirmWarning1`) and its end (`AbsentWarning1`) of an alarm (interaction between `P1` and `P2`),
the examination (`ExamineWarning1`) of an alarm by `P1` (interaction between `P1` and the environment),
the examination (`ExamineWarning2`) of an alarm by `P2` (interaction between `P2` and the environment),
the beginning of a new cycle of `P1` (`BeginCycle1`). It is not an interaction, but, since it modifies variables of this level so it needs to be here,
the beginning of a new cycle of `P2` (`BeginCycle2`).

This event model is built as explained in the previous section.

```
SYSTEM FWS2
SETS Ww, Ss
VARIABLES   wp, se, wc, Num1, Wp1, We1, Wc2, We2, Num2
INVARIANT   wp, wc, Wp1, We1, Wc2, We2 ⊆ Ww ∧ se ⊆ Ss ∧ Num1, Num2 ∈ NAT
INITIALIZATION   wp, se, wc, Num1, Wp1, We1 Wc2, We2,Num2 : (wp = ∅ ∧ se = ∅ ∧ wc = ∅ ∧ Num1 = 0
    ∧Wp1 = ∅ ∧ We1 = ∅ ∧ Wc2 = ∅ ∧ We2 = Ww ∧ Num2 = 0)
EVENTS
NewWarning ≙ /* A new alarm is présent */
  ANY wx WHERE wx ∈ Ww ∧ wx ∉ wp
  THEN   wp := wp ∪ {wx} END;
EmittedSignal ≙ /* A new signal is emitted */
  ANY sx WHERE sx ∈ Ss ∧ sx ∉ se
  THEN   se := se ∪ {sx} END;
ExamineWarning1 ≙ /* P1 is examining an alarm */
  ANY wx WHERE wx ∈ Ww ∧ wx ∉ We1
/* wx has not yet been examined */
  THEN
    IF wx ∈ wp THEN /* The alarm is present */
      Wp1, We1 := Wp1 ∪ {wx}, We1 ∪ {wx}
    ELSE We1 := We1 ∪ {wx} END
  END;
BeginCycle1 ≙ /* New cycle for P1 */
  SELECT We1 = Ww
  THEN We1, Wp1, Num1 := ∅, ∅, Num1 + 1
/* All the alarms will be examined */
  END;

EndWarning ≙ /* End of an alarm */
  ANY wx WHERE wx ∈ wp
  THEN   wp := wp − {wx} END;
EndSignal ≙ /* Extinction of a Signal */
  ANY sx WHERE sx ∈ se
  THEN   se := se − {sx} END;
ConfirmWarning1 ≙ /* P1 confirms an alarm */
  ANY wx WHERE wx ∈ Ww ∧ wx ∉ wc
  THEN   wc := wc ∪ {wx} END;

AbsentWarning1 ≙ /* P1 indicates that wx is absent */
  ANY wx WHERE wx ∈ Ww
  THEN   wc := wc − {wx} END;

BeginCycle2 ≙ /* New cycle for P2 */
  SELECT We2 = Ww
  THEN We2, Wc2, Num2 := ∅, ∅, Num2 + 1
/* All the alarms will be examined */
  END;

ExamineWarning2 ≙ /* P2 is examining an alarm */
  ANY wx WHERE wx ∈ Ww ∧ wx ∉ We2 /* wx has not yet been examined : wx ∉ We2 */
  THEN IF wx ∈ wc
      THEN Wc2, We2 := Wc2 ∪ {wx}, We2 ∪ {wx} /* The alarm is confirmed */
    ELSE We2 := We2 ∪ {wx} END /* The alarm is not confirmed */
  END;
```

Note that interactions between `P1` and `P2` are achieved via the variable wc which is modified by `P1` and is accessed by `P2`.
Note also that the events `BeginCycle1` and `BeginCycle2` remove at once all the elements of respectively $we_1,\ wp_1$ and $we_2,\ wc_2$: they model the beginning

of each cycle. As indicated in the guards ($we_1 = Ww$ and $we_2 = Ww$), these events can be activated only when all alarms have been examined by each process.

4 From event models to modules: Introduction

The event system FWS2 models the interactions of the system with the environment and between the two processes P1 and P2.
We will now illustrate, along with this example, the end of the process: going from this event system to the specification of each component and to the specification of the interface module.

In order to do so, we have first to transform the event system into a module which will be shared by the components. Its role is to provide each component with the operations it needs in order to interact: it is the interface module.
The specification of each component is not directly derived from the event model. It must be elaborated separately and it must contain the properties characterizing the component on its own. Don't forget that these properties have not been taken into account in the event model. But, a component specification can use some variables of the interface module. By doing so, we can relate the local properties of the component to the expected behaviour of the whole system. This specification can later be refined in order to obtain its implementation.

We will now illustrate this part of the process step by step.
Modules and operations. A module specification $MS = (x, IM, A, O_i)$ is formed by a shared variable x that represents the module state, an invariant IM that expresses static properties of the module, the specification A that initializes the module and the set of operations O_i for $i \in 1..n$ that defines the dynamics. Each operation O_i takes the form of a *pre-post specification* $(P_i(x), Q_i(x, x'))$. In the syntax of the tool, it takes the form `Pre` $P(x)$ `THEN` `S` where `S` is a generalized substitution. As there is a translation of generalized substitution into before-after predicates [Abr 96-1], it corresponds to usual pre-post formulation of program specification. Such modules are transition systems, whose initial states are defined by the initialization, and whose transitions are defined by the operations. From this semantic point of view, modules and event systems are equivalent.
Consistency of a module. A module is said to be consistent if the invariant is established by the initialization and preserved by each operation.
Refinements. A module M is refined by $M1$ if each operation of M is refined by the corresponding operation of $M1$ and so is for the initialization. An operation, that is to say, a program specification (P, Q), is refined by another one (R, S) if for all program t, $\{R\}\ t\ \{S\}$ implies $\{P\}\ t\ \{Q\}$. These proof obligations are generated by the tools. Here is the semantic difference between an operation and an event: refining a program specification allows to weaken the pre-condition whereas refining an event allows to strengthen the guard.

Module implementations. Let M be a module (x, IM, A, O_i) . An implementation of M is a set of programs $t_0, t_1, ..., t_n$ such that:
$\{true\}\ t_0\{A \wedge IM\}$, and $\{IM \wedge P_i\}\ t_i\ \{IM \wedge Q_i\}$.
Those conditions plus the consistency of M, implies the following property:
For all $\mathcal{M}$ we have $\{true\}\ t_0; \mathcal{M}(t_1, \ldots, t_n)\{IM\}$ where $\mathcal{M}$ is a combination with if then else, while, sequence (;) and operations O_i. In other words, the invariant is satisfied by all module states.
Module importation. We can import a module to implement another one [Abr 96-1].

Let M be the same module as before and M_1 be (x, IM_1, A_1, V_i) (for $i \in 1..k$). Each V_i is of the form (R_i, S_i). To implement M_1 through an importation of M, we have:

(i) to provide for each V_i an expression $\mathcal{M}_i(O_1, ..., O_n)$ built with combinations of calls of O_i,

(ii) to prove that $V_i \sqsubseteq \mathcal{M}_i(O_1, ..., O_n)$.

If the imported module M has an implementation i.e. a set of programs $t_0, t_1, ..., t_n$ such that: $\{true\}\ t_0\{A \wedge IM\}$, and $\{IM \wedge P_i\}\ t_i\ \{IM \wedge Q_i\}$, the conditions above ensure that, for all $\mathcal{M}$:
$\{true\}\ t_0; \mathcal{M}(\mathcal{M}_1(t_1, \ldots, t_n), \ldots, \mathcal{M}_k(t_1, \ldots, t_n))\{IM_1\}$.

5 Towards an interface module: first step

5.1 Transformation of an event into an operation

An event system specification allows us to observe a complete system - a system which does not interact with the observer. So, a system specification models a closed universe. A program taking inputs and producing outputs is not a closed universe. The way to connect these two approaches is to consider the event system as a tool to *observe* the program behaviour. This is the case if each transition allowed by an operation is also a possible transition of the event which corresponds to this operation [Lop,Sim,Don 00], [Lop 02].
Then, to transform a system into a module, it suffices to transform each event:
$E\ =$ any a where $G(x, a)$ then $S_{x,a}$ into the operation
$[res \longleftarrow]op(a)\ =$ pre $G(x, a)$ then $S_{x,a}$.

We can freely add output parameters. It is obvious that in this case, the event system and its corresponding module define the same transition system.

More complex translations can be done between events and operations. In these cases, some proof obligations have to be generated and discharged. For more details, see [Lop 02], chapter 5.7.

5.2 Application to FWS

The module interface is obtained from FWS2 following the translation defined above.

```
MACHINE FWS_ENV
VARIABLES  wp, se, wc, Num1, Wp1, We1, Wc2, We2, Num2
INVARIANT wp, wc, Wp1, We1, Wc2, We2 ⊆ Ww ∧ se ⊆ Ss ∧ Num1, Num2 ∈ NAT
INITIALIZATION  wp, se, wc, Num1, Wp1, We1 Wc2, We2,Num2 :
   (wp = ∅ ∧ se = ∅ ∧ wc = ∅ ∧ Num1 = 0 ∧ Wp1 = ∅ ∧ We1 = ∅ ∧
   Wc2 = ∅ ∧ We2 = Ww ∧ Num2 = 0)
OPERATIONS
NewWarning(wx) ≙
  PRE wx ∈ Ww ∧ wx ∉ wp
  THEN wp := wp ∪ {wx} END;
EmittedSignal(sx) ≙
  PRE sx ∈ Ss ∧ sx ∉ se
  THEN se := se ∪ {sx} END;
EndSignal(sx) ≙
  PRE sx ∈ Se
  THEN se := se − {sx} END;
bres ←− ExamineWarning2(wx) ≙
  PRE wx ∈ Ww ∧ wx ∉ We2
  THEN
    IF  wx ∈ wc
    THEN Wc2, We2, bres :=
    Wc2 ∪ {wx}, We2 ∪ {wx}, TRUE
    ELSE We2, bres :=
      We2 ∪ {wx}, FALSE END
  END;
BeginCycle2 ≙
  PRE
    We2 = Ww
  THEN We2, Wc2, Num2 :=
  ∅, ∅, Num2 + 1 END

EndWarning(wx) ≙
PRE wx ∈ wp
THEN wp := wp − {wx} END;
ConfirmWarning1(wx) ≙
  PRE wx ∈ Ww ∧ wx ∉ wc
  THEN wc := wc ∪ {wx} END;
AbsentWarning1(wx) ≙
  PRE wx ∈ Ww
  THEN wc := wc − {wx} END;
bres ←− ExamineWarning1(wx) ≙
  PRE wx ∈ Ww ∧ wx ∉ We1
  THEN
    IF wx ∈ wp
    THEN Wp1, We1, bres :=
    Wp1 ∪ {wx}, We1 ∪ {wx}, TRUE
    ELSE We1, bres :=
      We1 ∪ {wx}, FALSE END
  END;
BeginCycle1 ≙
  PRE
    We1 = Ww
  THEN We1, Wp1, Num1 :=
∅, ∅, Num1 + 1 END;
```

6 Toward an interface module: second step

6.1 Logical and abstract variables

All the module variables are used to express properties. Some of them model objects which will be effectively implemented by the program -they are named *abstract variables*, and have an operational content. Others are only logical -they are called *logical or auxiliary variables* [Aba,Lam 88]. Abstract variables are the only ones transformed during the module refinement process. They are implemented as well as data-refined.

The logical variables will not appear in the final implementation.

This step is concerned with separating the module variables between these two species.

Let M be a module which contains abstract variables (a) and logical variables (l). Let M_1 be the module obtained from M when all the expressions where logical variables appear, are eliminated. We want to take, as an implementation of M, any implementation of $M1$. This is possible if each operation O_M of M is of the following form:
$O_M = \ (p(a) \ \wedge \ pl(a,l) \ , \ q(a,a') \wedge ql(a,l,l'))$
and satisfies :
$\forall a, a', l. \ (p(a) \wedge pl(a,l) \wedge q(a,a') \ \rightarrow \ \exists l'. \ ql(a,l,l')).$
Consider now the corresponding operation in M_1. It takes the form: $op_{M_1} =$

$(p(a)\ ,\ q(a,a'))$.

So we have:

(i) as soon as O_M can be activated, O_{M_1} can also be activated (the pre-condition has been constrained).

(ii) abstract variables a, a' do not depend on logical variables (but l may depend on a).

(iii) under the pre-condition of O_M, all the states which can be reached by O_{M_1} are also attainable by O_M. This means that, during the refinement process, whatever implementation choice is made which reduces the internal non-determinism of O_{M_1}, it will realize the post-condition of O_M.

These constraints ensure that any implementation of M_1 is also an implementation of M [Lop 02].

From what we have just said, we can roughly consider M as an overloading of M_1.

Logical variables allow to express logical properties, which involve notions not directly present in M. These variables also allow to constrain the use of operations of M_1.

6.2 Application to FWS

In the interface module `FWS_ENV`, abstract variables are those used to communicate with the environment. Other variables are logical:

```
VARIABLES    wp, se, wc
LOGICAL VARIABLES    Num1, Wp1, We1, Wc2, We2, Num2
```

The base module corresponding to `FWS_ENV` i.e. `FWS_ENV` without logical variables is:

```
MACHINE FWS_BASE
VARIABLES wp, se, wc
INITIALIZATION wp, se, wc: wp = ∅ ∧ se = ∅ ∧ wc = ∅
OPERATIONS
NewWarning(wx) ≙
  PRE wx ∈ Ww ∧ wx ∉ wp
  THEN wp := wp ∪ {wx} END;
EmittedSignal(sx) ≙
  PRE sx ∈ Ss ∧ sx ∉ se
  THEN se := se ∪ {sx} END;
EndSignal(sx) ≙
  PRE sx ∈ Se
  THEN se := se − {sx} END;
bres ←− ExamineWarning2(wx) ≙
  PRE wx ∈ Ww
  THEN
    IF   wx ∈ wc
    THEN bres := TRUE
    ELSE bres := FALSE END
  END;
EndWarning(wx) ≙
  PRE wx ∈ wp
  THEN wp := wp − {wx} END;
ConfirmWarning1(wx) ≙
  PRE wx ∈ Ww ∧ wx ∉ wc
  THEN wc := wc ∪ {wx} END;
AbsentWarning1(wx) ≙
  PRE wx ∈ Ww
  THEN wc := wc − {wx} END;
bres ←− ExamineWarning1(wx) ≙
  PRE wx ∈ Ww
  THEN
    IF wx ∈ wp
    THEN bres := TRUE
    ELSE bres := FALSE END
  END;
```

This interface module is the effective interface module. `NewWarning` and `EndWarning` are implemented by the devices which detect warning situations (sensors and other electronic devices); `EmittedSignal` and `EndSignal` are implemented by output devices; `ExamineWarning1` is implemented by a mechanism provided by the environment to obtain inputs and so on...

Note that `BeginCycle1` and `BeginCycle2` have disappeared. They are, in fact, logical operations which formally express the necessity of examining all the alarms. They have a logical control role.

7 Component specification

7.1 Sharing

At this point, we have an interface module which provides the communication operations of the system. We have now to write the component specifications. In this application, we want:
1- to have the interface module E=(x, IE, A, V_i).
2- to model each component in a separate and independent module: $M_1 = (x, y, IM_1, A_1, W_i)$ and $M_2 = (x, z, IM_2, A_2, S_i)$. By independent, we mean that the consistency is proved independently for each module. It also means that each module has to be refined independently.
3- to force each module to be implemented using an import of the *same* interface module E. E becomes a shared module.
4- to preserve the correctness. By correctness, we mean that every interleaving of calls of programs realizing M_1 and M_2 operations (which modify interface variables through operation calls of E) establishes *both* the invariants of M_1 and M_2.
Formally: for every program t_{V_j} which realizes V_j and for every combination $\mathcal{N}_1$ of t_{V_j} and $\mathcal{N}_2$ of t_{V_j} we must have:
$\{true\}\ IE; A_1; A_2; (\mathcal{N}_1 || \mathcal{N}_2)\ \{IM_1 \wedge IM_2\}$.
As E needs to be shared by M_1 and M_2, the correctness cannot be expressed locally on each component module.

This is clearly not true in general: as M_1 and M_2 can share variables of E, an operation of $M2$ can break the invariant of $M1$ and vice-versa.

To avoid this problem, one solution is the following:
• For each component M_j we must identify the set of operations O_j of E which are exclusively used by M_j. O_j is the set of operations that M_j will be allowed to use.
• Let R_j be the set of variables modified by O_j, $R_j \subseteq x$. R_j is the set of variables that can be used by M_j.

7.2 Application to FWS

We are able now to model the two processes `P1` and `P2`. We focus on `P2`, the model of `P1` can be made in the same way. First we have to extract

the FWS_ENV operations which will only be used by the process P2. They are EmittedSignal, EndSignal, ExamineWarning2, BeginCycle2. So the variables of P2 coming from FWS_ENV are se,$Wc2$,$We2$,$Num2$. It will also include its proper variables in order to express its own local properties. The proper variables are: wa, wni, $wnis$, sa, snc, they have been introduced to formalize the following local properties:
1. Confirmed warnings not inhibited by the current fly phase are activated warnings. $Wc2 \cap wni \subseteq wa$.
2. No cancelled signals, associated with activated warnings, are activated signals.
3. Activated warnings are confirmed warnings.
4. Activated signals are associated to activated warnings.
5. Emitted signals are activated signals.
6. The crew is always able to determine the warnings associated to emitted signals.

Another property is, that during each cycle, all the alarms must be examined. This is expressed by $We2 = Ww$. We use here logical variables. As the process P2 is cyclic, the module includes only one operation Cycle2 with a $true$ pre-condition and which models a step of the cycle. All the properties of the module are expressed into the invariant - the post-condition of Cycle2 just establishes the invariant and counts the number of steps.

MACHINE P2 Processus 2.
SETS Ww, Ss
CONSTANTS s_w, c_w
PROPERTIES $s\_w \in Ss \to Ww \ \wedge c\_w \in Cc \leftrightarrow Ww$
VARIABLES se, $Wc2$, $We2$, $Num2$, wa, wni, $wnis$, sa, snc
INVARIANT
$se \subseteq Ss \ \wedge Wc2 \subseteq Ww \ \wedge We2 \subseteq Ww \ \wedge \ Num2 \in NAT \ \wedge wa \subseteq Wc2 \ \wedge$
$wni \subseteq Ww \ \wedge wnis \subseteq wa \ \wedge sa \subseteq Ss \ \wedge \ se \subseteq sa \ \wedge snc \subseteq Ss \ \wedge$
$Wc2 \cap wni \subseteq wa \ \wedge$ **(Property 1.)**
$s\_w^{-1}[wa] \cap snc \subseteq sa \ \wedge$ **(Property 2.)**
$wa \subseteq Wc2 \ \wedge$ **(Property 3.)**
$s\_w[sa] \subseteq wa \ \wedge$ **(Property 4.)**
$se \subseteq sa \ \wedge$ **(Property 5.)**
$wnis \subseteq ran(c\_w) \ \wedge$ **(Property 6.)**
$We2 = Ww$ **communication property**
INITIALIZATION
se, $Wc2$, $We2$, $Cc2$, $Ce2$, $Pc2$, $Pe2$, $Num2$
wa, wni, $wnis$, sa, snc: $INVARIANT$
OPERATIONS
Cycle2 $\hat{=}$
se, $Wc2$, $We2$, $Num2$, wa, wni, $wnis$, sa, snc:
$(INVARIANT \ \wedge Num2 = Num2\_0 + 1)$
END

Note that the variable $Num2$, which is incremented by each cycle, forces each implementation of Cycle2 to behave correctly: if this increment does not occur, a refinement of Cycle2 by skip would still be correct.
When $Num2$ is incremented, the following happens:
$Num2$ is a variable “coming” from FWS_ENV, hence each implementation of

`Cycle2` will call `BeginCycle2` as it is the only operation of `FWS_ENV` which modifies $Num2$. Each implementation must reestablish the invariant, as it is the post-condition of `Cycle2`. And, in particular, it must establish $We2 = Ww$. This can only be done by as many calls to `ExamineWarning2`, as the number of alarms in Ww.

8 Conclusion

A first result of this work is to propose a method which takes into account the entire development of a system. This method has been used to model two industrial case studies. We have used the existing tools (in our case Atelier B) to formally prove a large part of the proof obligations generated by the method and which is actually supported by these tools. Almost all the generated proof obligations have been automatically discharged (around 85 %). The new notions : logical variables, shared modules and event-to-operation transformations, generate new proof obligations. They are not supported by the actual tools ; these proof obligations have been proved "by hand". The fact that it has been used on industrial cases, shows that the proposed method is suitable to treat large-scale systems. Secondly, what is interesting is that these new notions solve the problems met in previous works, when we tried to model applications of this kind with only the notion of module [Lop 96-1]. Another point of interest is that these extensions do not represent a profound modification to the existing theory. Hence, their incorporation into existing tools can be achieved.

Further work has to be done to improve (to refine) the conditions we have formulated in order to ensure the correctness of the whole process.

`Acknowledgements`: The authors want to express their gratitude to the persons which have made this work possible: J.R. Abrial, A. Burlureau, P.Desforges, the Matra team and Aerospatiale.

References

[Atelier B] ClearSy, http://www.atelierb.societe.com/index.html

[B Toolkit] B-Core http://www.b-core.com/btoolkit.html

[Aba,Lam 88] M. Abadi and L. Lamport, *The existence of refinement mappings*, Technical Report. Digital Systems Research Center, Palo Alto, California, 1988.

[Abr 96-1] Abrial J.R., *The B Book. Assigning programs to Meanings*, Cambridge University Press, 1996.

[Abr 96-2] Abrial J.R., *Extending B without changing it, for Developing Distributed Systems*, in First B conference, H.Habrias editor, 169-190, Nantes, 1996.

[Back 88] Back R.J., *A calculus of refinements for program derivations*, Acta Informatica 25, 593-624, 1988.

[Back,Kur 88] Back R.J.R., Kurki-Suonio R., *Distributed cooperation with action systems*, in ACM Transactions on Programming Languages and Systems, vol. 10, No. 4, pages 513-554, ACM, 1983.

[But 96] Butler M.J., *Stepwise refinement of communicating systems*, Science of Computer Programming 27, 139-173, 1996.

[Lam,Sha 90] Lam S.S., Shankar U., *A Relational Notation for State Transition Systems*, in IEEE Transactions on Software Engineering, Vol 16, No 7, 755-775, 1990.

[Lop 96-1] Lopez N., *Construction de la specification formelle d'un systeme complexe*, in First B conference, H.Habrias editor, 63-119, Nantes, 1996.

[Lop 96-2] Lopez N., *Construction de la specification formelle d'un systeme complexe*, Memoire d'ingenieur CNAM, 1996.

[Lop 99] Lopez N., *An 'event based B' industrial experience*, in the proceedings of the B user Group Meeting, edited by Ken Robinson, Applying B in an industrial context, World Congress on Formal Methods 1999.

[Lop,Sim,Don 00] Lopez N., Simonot M., Donzeau-Gouge V., *Deriving software specifications from event-based models*, in the proceedings of the 1st International Conference of B and Z users, edited by Jonathan P. Bowen, Steve Dunne, Andy Galloway, Steve King, Lecture Notes in Computer Science 1878, 209-229, Springer, 2000.

[Lop 02] Lopez N., *Spécification Formelle de Systèmes Complexes, Méthodes et Techniques"*, These CNAM, 2002.

[Morg 90] Morgan C., *Programming from Specifications*, Prentice-Hall International. 1990.