NASA's SAFER using B-Method

Sylvain Verly

Supervisor: Andrew Ireland

Second reader: David Corne

Final Year Dissertation

Honours Degree

Introduction

The objective of the project is to develop an implementation of the SAFER[1] backpack propulsion system using formal methods. SAFER backpack was designed by NASA. It helps an astronaut, in case of accidental separation from the spacecraft during an EVA[2], to return to his space shuttle. This project aims to develop a working software system for the backpack from the NASA specification using B-method. The software should response properly (as stated in the specification) to inputs given by the astronaut.

A key objective is to develop a deep understanding of how the B-method works, its strengths and limitations. The SAFER specification has been previously analysed using two other formal methods, *i.e.* PVS and VDM. Using the SAFER example it will be possible to compare the results of using B-Method with the results of these two previous approaches carried out.

!!!!!!! STRUCTURE TO BE PUT HERE !!!!!!

[1] SAFER stands for *Simplified Aid For EVA Rescue*.
[2] EVA stands for *Extra-Vehicular Activity*, also called ‘spacewalk’.

Background

2.1 Formal Methods

2.1.1 Introduction

A formal method is a technique based on rigorous mathematics concepts for the development of hardware or software systems. It aims to demonstrate their validity over their respective specifications. Formal methods provide a high level of guarantee, assuring a system to be free of bugs.

However there are costly in terms of human resources and time. Indeed, high skilled persons are required, and the development of such systems is time-expensive. Although not 100% reliable, they provide much better level of trust than any informal methods. Therefore, they are used in the development of critical systems, *i.e.* systems that involve human lifes and/or a large amount of money (*e.g.* the launch of a space shuttle[3]).

There are numerous several methods, carrying out different approaches, and implementing different aspects of formal methods. The **subsection 2.1.3** will focus on Z notation, a method created by Jean-Raymond Abrial[4], PVS (Prototype Verification System) a method to formally specify and VDM (Vienna Development Model) a method whose Jean-Raymond Abrial has been inspired by to create the B-method.

2.1.2 Formal methods' approach

A standard development approach such as the use of informal methods, starts from a set of requirements, stated in natural language. It aims to reach a working system, through several steps, such as conception, modules description, etc. Amongst which, some verification steps are carried out, to check that everything matches the requirements.

A formal method's approach almost follow the same pattern. However it uses logic, as well as mathematical concepts and theories. Most important parts, and common to all formal methods are specification and, proof and verification.

[3] this case is interesting to underline, because of Ariane 5 Flight 501 failure, due to a software bug (see ESA report [7] for more information).

[4] Jean-Raymon Abrial is the initiator of the B-method.

- **Specification:** formal methods can be used to give a specification to a system. A formal specification is based on a formal language, peculiar to each method, with a precise semantic, avoiding misinterpretations. This formal description is used throughout the development process to ensure the final result satisfies the informal specification (translated into formal specification). A good formal specification is ideally abstract [6, page 6], *i.e.* doesn't consider implementation in a target system.

  Here is a simple example, with mathematic notation as formal language.

  - **Informal description:** every person loves the ones who love them.
  - **Formal description:**
    - $love \in PERSON \leftrightarrow PERSON$

      This means that $love$ is relation between two persons. The symbol $\leftrightarrow$ indicates that it could be any relation.
    - $\forall p, q \bullet (p \in PERSON \wedge q \in PERSON \wedge (p \mapsto q) \in love \Rightarrow (q \mapsto p) \in love$
    - $love = love^{-1}$

      Means that $love$ is equal to its own reverse (the relation described above "works in both way").

  Some formal methods (whose B-Method) include several different levels of abstraction in a specification. Each level aims to be proofed by formal verification and proof. The different levels go from abstraction to a more concrete model, called implementation. The step to go from the former to the latter is called *refinement*. We will discuss it in more details in an other section.

- **Verification and proof:** a formal specification aims to be proved, in order to check its consistency. We don't want an expression such as $0 = 1$ to sneak into our specification. Of course this example is pretty obvious, but it will rarely be the case.

  There are several ways to carry out the prooving process. This can be done by hand, despite being a long task. Z notation, that will see later on in this report, is a perfect example of formal method where proof are carried out by hand. Most common way to proove a system is to use automatic proover. A tool peculiar to the method used is needed. Given a set of axioms, and inference rules, such a tool can proof some properties of a specification. However it is rarely fully automatic. It needs some guidance from a human being. Indeed, a proover could check thousands of uninteresting states if not guided by human (who knows, those states of proof are irrelevant).

2.1.3 Presentation of some formal methods

As seen above there are numerous formal methods. All these don't necessarily go all the way to a complete implementation of a working system. Some of them only provide languages for specification and manual formal verification, some others provide computerized tools to verify consistency of abstract machines, some others offer tools to implement abstract machines into a target language, etc. This report will adress Z notation, PVS, VDM, and of course B-Method in which we are interested in.

In addition, as mentioned on DCSSI report [6, page 6] :

"*L'utilisation d'une méthodologie formelle de développement doit reposer sur l'utilisation d'un ensemble d'outils constituant une implémentation de la méthode formelle.*"

This can be translated as: "*The use of formal approaches has to rely on a set of tools, designed as an implementation of the formal method.*" This is important to bear in mind when choosing a formal approach for development. Even though it is possible to carry out verification by hand as proposed on Z notation, this solution is not viable on significant sized project.

2.1.3.1 Z notation

Z is a notation based on mathematics, to describe software or hardware system. It uses *set theory* and *predicate logic* to express that system, and therefore helps the formalization of it. A system described in Z comprises:

- a *state* of the system which tells us *what* the system is, through a collection of state variable and their values,
- a collection of *operations* that can change this state.

A Z specification is organised with schemas. The following figure demonstrate the structure of a schema and is used to define state or operations:

Schema Name

Declarations

Predicate

As an example, let's write a door system in Z notation:

- a door is in the state closed or opened,
- it can switch from one state to another respectively by operations open and close,
- a door is initially closed.

STATE

$doorState \in \{opened, closed\}$

This schema defines the set STATE defined by the two elements *opened* and *closed*.

Initialisation

$\Delta STATE$

$doorState' = closed$

This schema defines an initialisation operation. Δ $STATE$ defines a schema that is allowed to be modified, and *doorState'* represents the variable *doorState* after modification. Here, the schema $STATE$ is modified as we changed the state of the variable *doorState*.

Close

$\Delta STATE$

$doorState \neq closed\ doorState' = closed$

Open

$\Delta STATE$

$doorState \neq opened \; doorState' = opened$

In there two operations we had the notion of precondition[5], *i.e.* here we cannot open a door already opened, neither close a door already closed[6].

Z notation is not a method but just a notation. For a long time, it was used with just a piece of paper and a pencil. Today a set of tools exists for formatting, type-checking, and helping for proofs.

2.1.3.2 Prototype Verification System (PVS)

PVS is a computerized environment designed for formal specification and formal verification. It provides [8]:

- **a specification language:** it is based on predicate logic. A PVS specification is defined as a set of theories. A theory includes signature for types, constants and axioms, definitions and theorems associated with this signature.
- **predefined theories:** PVS includes a set of predefined theories. For instance, the theory *booleans* introduces:
  - the nonempty type `boolean`,
  - the elements `TRUE` and `FALSE`,
  - operators such as `AND`, `OR`, implies $\Rightarrow$, `WHEN` and equivalence $\Leftrightarrow$.
- **a type checker:** this tool is concerned with checking a specification is logically correct, and that no errors are found in the specification.
- **theorem proover:** a proof is made once a specification has been successfully type-checked. The PVS theorem proover is composed of a set of basic theorems and axioms. A proof is interactive and is carried out by the user who defines proof strategies through a set of commands which control the use of theorems and axioms.

PVS is distributed under General Public License, and is available on Linux, Mac and Solaris.

[5] This notion is largely used in B method, and in software development in general

[6] It doesn't really change anything to state the *doorState* variable become *opened* instead of *opened* in this case. We might think these preconditions are unecessary. However our door system, doesn't state that opening a door already open is allowed. Imagine this door system is designed for an aircraft! A precondition checking the variable values is then necessary.

2.1.3.3 Vienna Development Model (VDM)

The VDM development started in a Vienna IBM Laboratory [9] in 1970. It introduces a new development paradigm: the refinement. This step aims to transform an abstract specification (close to the problem domain) into a concrete specification (close to the machine or to the target language). VDM discerns two different refinement processes [5]:

- **Data reification:** the refinement of abstract data types (such as sets, sequences, etc.) into data structures (such as arrays, records, etc.).
- **Operations decomposition:** the refinement of operations and functions defined in the abstract specification. An abstract specification describes *what* do operations do rather than *how*. The refinement process, adds *explicit* information (such as algorithm) describing how an operation is carried out. This *explicit* information is close to a target implementation language.

The language of VDM is VDM-SL and supports VDM's different levels of abstraction.

2.1.4 Conclusion

As just adressed, there are many different formal methods. We were particularly interested in Z notation and VDM because the B-Method has been inspired by these two. PVS has been used to formally specify the SAFER system that will be adressed in an other section.

2.2 SAFER: Simplified Aid For EVA Rescue

2.2.1 Introduction

The SAFER is a streamlined version of MMU (Manned Maneuvering Unit). It allows a crewmember of a spacecraft, or space station to be engaged in EVA in a safer way. Compared to MMU, the SAFER is lighter, and designed for contingency only, *i.e.* as a self-rescue device. Spacewalk with SAFER are always tethered, i.e. the crewmember is attached to space station. However, for some numerous reasons, the tether can break and the crewmember will find himself floating away from his station, with no or little chance for him to come back. This is why SAFER has been designed.

It fits around the space suit[7] as shown on figure 2.1. A hand held controller unit is attached in front of the crewmember's EMU. It allows him to perform the six degrees of freedom[8] (see figure 2.2). The SAFER uses 24 gaseous-nitrogen (GN_2) thrusters, distributed all around.

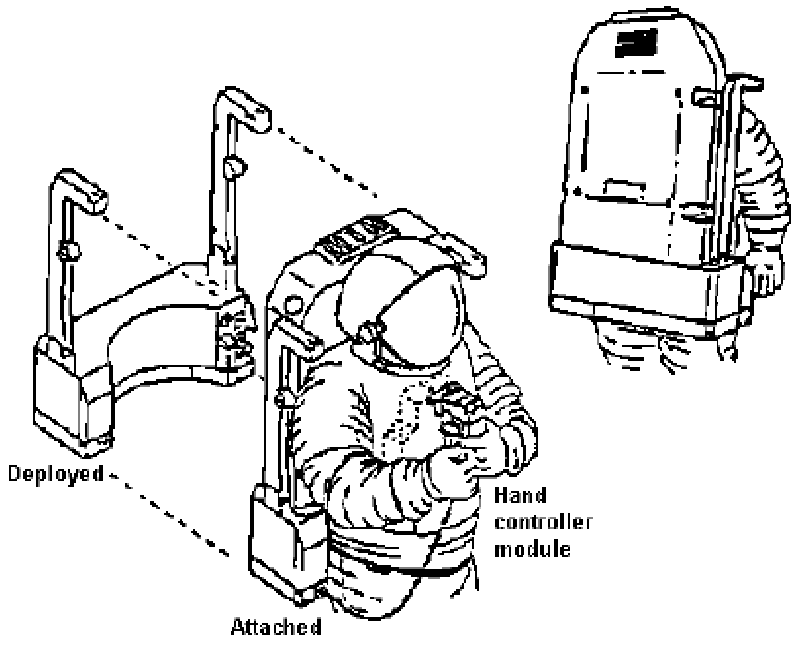


Figure 2.1 SAFER attached to Extravehicular Mobility Unit [3, figure C.1]

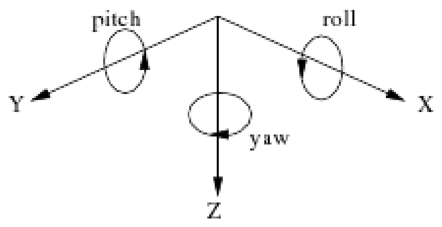


Figure 2.2 Six degrees of freedom [3, part of figure C.3]

2.2.2 System Software

The system software or avionics software is responsible for controlling the SAFER depending on crewmember commands. It is composed of two main subsystems:

[7] also called Extravehicular Mobility Unit (EMU)
[8] translation on each axis X, Y and Z, and rotation pitch, yaw and roll

- **Maneuvering control subsystem:** this system aims to receive input from the Hand Controller and output acceleration command to the thrusters. It comprises of two subsystems: the command interpreter subsystem that receive input from hand held controller, and an automatic attitude hold subsystem (AAH) that is designed to keep rotation rates close to zero. This is the combination of these two subsystems that will produce a single acceleration command. The AAH can be switched off at any time by the crewmember.

  However the thruster command output is subject to various constraints, *e.g.* only one translational axis at a time receives acceleration, four thrusters maximum turned on simultaneously, etc. These constraints are summed up into a selection logic table (see **table 2.1** and **table 2.2**) with three possible values for each axis: negative thrust -, positive thrust + or no thrust at all. The thrusters are shown in **figure 2.3**. Each thruster is designated by a letter indicating the direction of forces applied on the SAFER: U for Up, D for Down, B for Back, F for Forward.

- **Fault detection subsystem:** this system performs testing function to detect fault during and before an EVA and manages the display interface.

The **figure 2.4** shows the architecture and flows of the SAFER system.

X	Pitch	Yaw	Always turned on	On if no roll command
-	-	-	B4	B2 B3
-	-		B3 B4	
-	-	+	B3	B1 B4
-		-	B2 B4	
-			B1 B4	B2 B3
-		+	B1 B3	
-	+	-	B2	B1 B4
-	+		B1 B2	
-	+	+	B1	B1 B4
	-	-	B4 B1	
	-		B4 F2	
	-	+	B3 F2	
		-	B2 F1	
		+	B3 F4	
	+	-	B2 F3	
	+		B1 F3	
	+	+	B1 F4	
+	-	-	F1	F2 F3
+	-		F1 F2	
+	-	+	F2	F1 F4
+		-	F1 F3	
+			F2 F3	F1 F4
+		+	F2 F4	
+	+	-	F3	F1 F4
+	+		F3 F4	
+	+	+	F4	F2 F3

Table 2.1 Thruster selection table for
X, pitch and yaw commands [3, table C.2]

Y	Z	Roll	Always turned on	On if no pitch or yaw
-	-	-		
-	-			
-	-	+		
-		-	L1R	L1F L3F
-			L1R L3R	L1F L3F
-		+	L3R	L1F L3F
-	+	-		
-	+			
-	+	+		
	-	-	U3R	U3F U4F
	-		U3R U4R	U3F U4F
	-	+	U4R	U3F U4F
		-	L1R R4R	
		+	R2R L3R	
	+	-	D2R	D1F D2F
	+		D1R D2R	D1F D2F
	+	+	D1R	D1F D2F
+	-	-		
+	-			
+	-	+		
+		-	R4R	R2F R4F
+			R2R R4R	R2F R4F
+		+	R2R	R2F R4F
+	+	-		
+	+			
+	+	+		

Table 2.2 Thruster selection table for Y, Z and roll commands [3, table C.3]

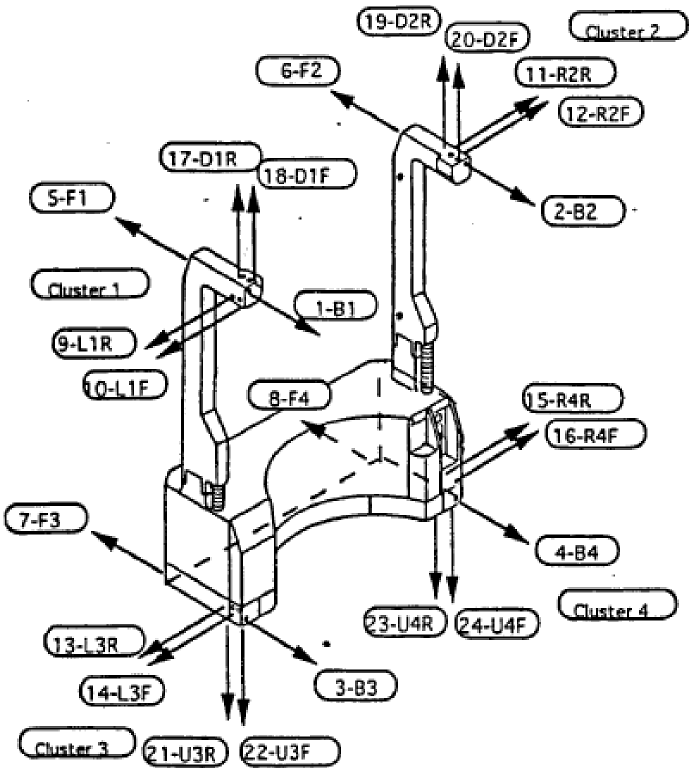


Figure 2.3 Thrusters designation [3, part of figure C.3]

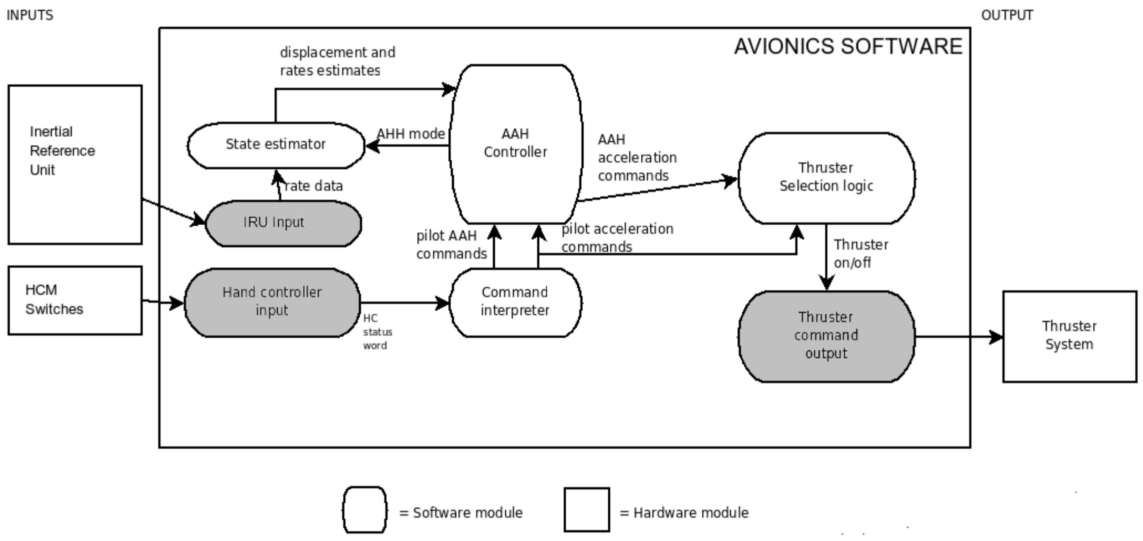


Figure 2.4 Architecture of the SAFER system [3, figure C.7]

B-method: a short description

3.1 Introduction

"**B** *is a method for specifying, designing, and coding software systems.*"

(Abrial, 1996, page xv)

Note: *this section relies for a major part on The B-Book [2] and Henri Habrias*[9] *handouts 1 and 2 [4].*

B-Method is a formal method introduced by Jean-Raymond Abrial. Many ideas of B-Method are derived from VDM as he had collaborated with C.B. Jones (who had been involved in the VDM project), and from Z, as J.R. Abrial is one of the originators of the notation. As he stated, the B method intends to include all steps from specification to code.

The term B includes several different aspects that need to be clearly differentiated:

- **Method:** B is a formal method. It provides an approach for "specifying, designing, and coding software systems", and also formal verification.
- **Language:** B-Method provides one common language for each step of the B approach (specifying, refining, implementing).
- **Set of tools:** as seen before, a good formal method includes a set of tools implementing it. B-Method provides proover and code generation tools.

Moreover the method proposed by Abrial, covers:

- **Formal specification:** expression of an informal statement into a formal one thanks to B language. Each module identified in the informal specification can be translated into one abstract machine. This includes specification of data (like in Z, it uses the set theory and predicate logic), and specification of operations. Each of them are defined by a precondition (without which the operation cannot be executed if the condition is not reached), and an action that describes *what*

[9] Henri Habrias was teaching B-Method in the University of Nantes (France).

the operation does. This last one is expressed as an atomic action and that does not contain sequencing and loops. It leaves a larger scope of decision for a later refinement.

- **Type-Checking:** *"any predicate involving set-theoritic constructs [shall] be* type-cheked *before being proved"* [2, page 64]. Type-checking is concerned with checking a formal statement and operations of an abstract machine "do make sense". It aims to forbid the manipulation of terms authorised by the B syntax.

- **Refinement and Proof:** a specification is intended to be 'translated' into an executable code. This is done by carrying on successive refinement steps. As while the refinement process goes, programming structures (loop, sequencing, etc.), and programmable types (arrays, files, etc.) are added to the refined machine, towards implementation machine. The refined machine, intermediate state between an abstract machine and implementation, is a cross between these two. It is not a mathematical model anymore, but not a programmable machine yet.

  Each step of the refinement process has to be prooved in order to show that it satisfies the specification. This operation is carried out by an automatic proover.

- **Code generation:** this is the last step of the method. It consists of translating (using an automatic tool), an implemented machine into an imperative programming language.

3.2 Different level of abstraction

3.2.1 Abstract machine

An abstract machine defines the boundaries in which operates the concepts of abstract specification. It identifies specification and operations. The key word of an abstract machine in B is `MACHINE`.

3.2.1.1 Structure

Every abstract machine structure in B-method follow the same pattern. A machine needs:

- **Basic sets:** these will be used to define the types of variables. For instance a set `SWITCH` could be defined by $\{on, off\}$. Such sets are *none empty* and incommensurable. This means that if A and B are two basic sets, writing something like $A \cup B$ or $A \subseteq B$ is forbidden. This is a typing error! However something like $A \times B$ or $A \leftrightarrow B$ and so on are allowed.

  Due to that incommensurability, an issue related to empty sets is raised. For instance, let's define two variables aa and bb, with $aa \subseteq A$, $bb \subseteq B$. If we initialise the two variables: $aa := \{\}$ and $bb := \{\}$, can we affirm that $aa = bb$? The answer is no! These are two different empty sets. A parallel can be done with really strong typed languages, if aa is of `float` and bb is of type `integer`, with $aa := 0.0$ and $bb := 0$, comparing aa and bb would not be allowed.

- **Variables:** a machine uses variables to formalize an informal specification. Variables are declared abstract by default, although it is possible to declare them concrete in an abstract machine. A concrete variable can only be declared given concrete types. `INTEGER`, `NATURAL` are examples of concrete types whereas basic sets for instance are abstract types. Concrete typing in a machine reduces the range of possibilities variables offers for specification[10], that's why they are abstract by default.

- **Invariant:** it defines types of variables, and properties of variables. For instance a variable $switch$. The typing invariant is $switch \in SWITCH$, indicating $switch$ is either on or off. $switch = on$ is an invariant that define a property of the variable $switch$, where switch can only take the value on. It has to be respected at *any* time throughout the software system, otherwise the invariant is violated, and the system will not be fully prooved.

- **Initialisation:** this part is compulsery for *every* variables declared. It makes possible variables to be given initial values. These, of course, must respect the invariant.

- **Operations:** as explained in the B-Book [2, page 230, section 4.3] "the role of an operation is to modify the state of an abstract machine, and this, of course, within the limits of the invariant." These are known as functions in c, or procedure in Ada for instance.

  An operation in B-method includes the concept of *precondition*. It aims to setup the conditions within which the operation should be used. These conditions must be checked *outside* the operation body. That is why there are called precondition.

[10] supposed ideally abstract, see section ??.

Actually this concept is well known through the programming community, as you can use them in any language, not formalised though. Indeed, in a language like Jave for instance, a precondition can be used within the function comments, setting up conditions of use. It is the programmer responsibility in such a language to make sure, these conditions of use are respected *before* the function is called. But, nothing prevent him from doing it wrong, forgetting to check those conditions by mistake or anything else.

This is not the case in B-method, and more largely in formal methods. It makes sure preconditions are checked within the code prior to the function calling. We will come to that in **subsection 3.2.3**.

These 5 concepts are respectively comprised within `SETS`, `VARIABLES`, `INVARIANT`, `INITIALISATION` and `OPERATIONS` clauses. These are the basic clauses that can be found within an abstract machine. However an abstract machine could contain, only the clause `SETS` for instance. It is useful when some basic sets are to be used throughout all the project. For instance, considering a door system it could be interesting to have machines to carry operations on, and a machine to store the set $STATE = \{opened, closed\}$ that might be used in these different machines. The machine should be linked, somehow, using a link clause. We will come to that later on this report.

3.2.1.2 Example of an abstract machine

Here is an example of an abstract machine. It contains all the basic clauses mentionned in the previous section. It is based on the door system previously mentionned in subsection ??.

`MACHINE`

$DOOR$

`SETS`

$STATE = \{opened, closed\}$

`VARIABLES`

$doorState$

`INVARIANT`

$doorState \in STATE$

`INITIALISATION`

$doorState := closed$

```
OPERATIONS
open =
PRE
doorState ≠ opened
THEN
doorState := opened
END;

   close =
PRE
doorState ≠ opened
THEN
doorState := closed
END
END
```

3.2.1.3 Important abstraction concepts

Abstraction level requires to understand several aspects specific to that level. Here are presented the two most important largely used within abstract machine.

- **Indeterminism:** this concept is used within operations body. It allow a certain laxity in the specification, that is, leaving some piece information to a lower level of abstraction like implementation for instance. A variable for example does not necessarily need to be explicitly affected to a particular value. There is just the need to make sure the substitution[11] holds the invariant. So a form of indeterminism would be something like: *for a variable k, which is a number from 0 to 9, k is being affected the value either 0, 1, 2,..., or 9.* This is formally written with the

[11] A substitution defines an atomic operation by which a variable is given a new value, the symbol of substitution within B-method is ':='. The common form of a substitution is $Variable := Expression$

symbol $:\in$, translated by 'becomes in'. So for the example above the substitution would be: $k :\in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

- **Parallel substitution:** this concept is a form of indeterminism but it addresses the problem of sequencing. Indeed sequencing, is prohibited within abstraction level. An abstract operation (operation defined within an abstract machine) does not specify in which order substitutions are made, so that the operation is the most abstract possible. Something like $aa := 2; bb := 3$[12] would be written $aa := 2 \parallel bb := 3$ where $\parallel$ is the parallelism symbol. This can be turn into $aa := 2; bb := 3$ or $bb := 3; aa := 2$ within an implementation.

3.2.2 Refinement

Refinement is there to help going from an abstract specification, to a specification (also called implementation) that takes into account computer constraints (like sequencing). It is concerns with telling more information, like removing indetermism, although this is not compulsery. A refinement can handle both abstract and some implementation concepts. The refinement concepts are contained into a refinement machine, whose keyword is `REFINEMENT`. It must include which machine (abstract or refined[13]) it refines by the keyword `REFINE`.

3.2.2.1 What does refinement do?

- **Weaken operation precondition:** it is concerns with removing more and more parts of a precondition until makes it disappear. Indeed precondition is supposed to be called *before* calling an operation. Precondition will be tested (by the programmer, not the compiler) within the calling operation.
- **Sequencing:** it starts being introduced, and is part of removing parallelism.
- **Concrete variables** start being introduced to refine abstract variables. These variables are linked with each other within the invariant (this is called the *link invariant*), usually using the equality

[12] ';' is the sequencing symbol
[13] indeed it is possible to refine a refinement! Notice that you cannot refine an implementation.

symbol '='. Notice that this is not an easy task, especially when abstract types are basic sets, but we will come to that later.

- **Proof:** of course, the refinement is concerned with checking it does what the abstract machine does.

3.2.2.2 Important refinement concepts

These concepts are used within the refinement process, and are important to bear in mind when developing a B project.

- **Constant signature:** this means a refinement machine (including implementation) has to keep the same signature of operations than abstract machine throughout the whole process of refinement. This causes some problem of operation parameters. Indeed if a signature has to be the same from abstraction to implementation, of what type should be the parameters and the return type (if applicable)? Well, return variables have to be concrete at all times, *i.e.* even in an abstract machine, whereas operation parameters can be either abstract or concrete.
- **Automatic refinement:** the process of refinement is usually a repetitive task. Therefore, developers of B-method have build a tool that allow a programmer to automatically develop refinement machines, given a set of rules and an abstract machine.

3.2.3 Implementation

Implementation is the final step of the refinement process. It refines either a machine or a refinement, but the implementation itself cannot be refined anymore. Notice that the language used within implementation machine is called BO.

3.2.3.1 What does implementation do?

Here are presented, some concepts as well as what an implementation does.

- **No abstract variables:** implementation does not have state of its own, that is no abstract variables,
- **Importation:** it has to import other machines, using their operations to implement the operations to be implemented.

- **Control structure:** operations have to be implemented in such a way that the control structure are easily translatable in an imperative language,
- **Variables:** they are now strictly limited to the use of concrete types,
- **Indeterminism and parallelism** are now *definitely* suppressed,
- **Preconditions:** they are completely removed. However the code is prooved, only if the calling preconditions are respected, as seen in **subsubsection 3.2.2.1**.

3.2.3.2 Example of an implementation

This example is based on the previous abstract machine. It gives at the same time an example of refinement (since implementation is a refinement), that was not given in **section 3.2.2**.

```
   IMPLEMENTATION
$DOOR\_imp$
REFINES
$DOOR$
VARIABLES
$doorState\_imp$
INVARIANT
$doorState\_imp \in 0, 1\&$
/* Link invariant */
$doorState\_imp \in ran(encode\_state)\&$
$doorState \in dom(encode\_state)$
INITIALISATION
$doorState := 0$
OPERATIONS
$open =$
BEGIN
$doorState := 1$
END;

   $close =$
BEGIN
```

$doorState := 0$

END

END

$encode - state$ is a relation links (encodes) a basic set to concrete variables. It is usually defined in another machine, imported (see **subsubsection 3.3.1.4**) by the implementation, but in order to simplify it has not been included in this example. Just notice that $encode\_state \in STATE \twoheadrightarrow \{0, 1\}$, where $\twoheadrightarrow$ represents a bijection.

3.2.4 Code generation

This is the last step of a B project. This is not done by the programmer, but automatically by the tool. An implementation machine is translated from the BO language to a target language. Currently, 4 languages are available: C, C++, Ada, and High Integrity Ada (HIA).

3.2.4.1 What does code generation do?

- **Unitary translation of an implementation:** it generates a compilable code from BO. Remember that an implementation machine was made with control structure, like `IF THEN ELSE`, or `WHILE`, thus a code is easily translated. Besides, the target type of code can conflict with the BO language. For instance B0 might impose some restriction regarding variables names, restriction that would not be the same in C or Ada. This situation can lead to conflict.

  This step is unitary because it is translating each implementation machine one by one. Each of them will produce a single file, as well as a header file.
- **Project linking:** as each implementation machine is translated separately, a mechanism to link all the machines with each other is needed. This is what project linking does. It creates the references, `#include` for C and C++, and `uses` for Ada and HIA for instance.

We saw in this section the main aspects of each level of abstraction (code generation is not really a level of abstraction but it was useful to mention it), and how to go from on to another. However this only addresses the problem with one machine (one abstract machine, its refinement(s) and implementation). For significantly sized project a good linking mechanism between machine is needed. The next section addresses this subject, with the main different compisition clauses.

3.3 Design an application with B

The B method contains numerous characteristics of design, due to the use of different level of abstraction. The design will be sensibly different from a level to another. These differences and main composition clauses (that is how the link keywords are called) will be discussed in this section.

3.3.1 Abstract design

The B method embeds several composition clauses in order to organize the abstract specification. These clauses do not describe the structure of the final implementation, and therefore the structure of the final software. The composition clauses just help organising an abstract specification. Besides the use of these clauses will result in a specific refinement and implementation strategy.

3.3.1.1 The SEES clause

A SEES clause can be used in the definition of SETS or CONSTANTS that have to be available in the whole application. This is one way of using the SEES clause. There are obviously some more.

The **figure 3.1** shows the relationships of a machine M1 seeing a machine M2. A machine M1 that sees an other machine M2, has only read access to the variables through its own operations or through read only access operations of the machine M2. The machine M1 has access to the SETS and CONSTANTS of M2.

A machine comprises basic sets used throughout the whole project would be *seen* by every machine.

3.3.1.2 The INCLUDES clause

A INCLUDES clause provides *extension* relationships between two or more machines. A machine M1 including a machine M2 will extends the operations of machine M2. An operation of M1 can call an operation of M2 or an operation of M2 can be *promoted* so it becomes accessible through M1.

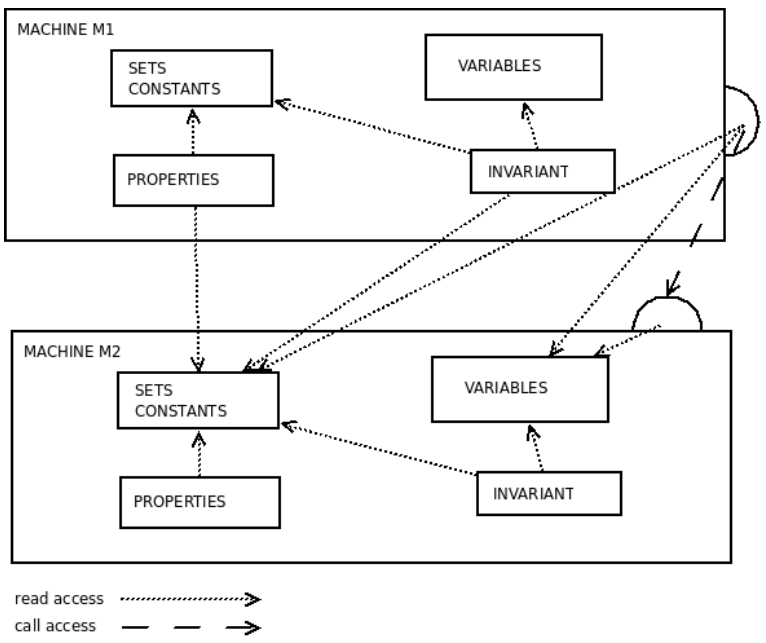


Figure 3.1 Sees clause and its relationships [10, figure 11.1]

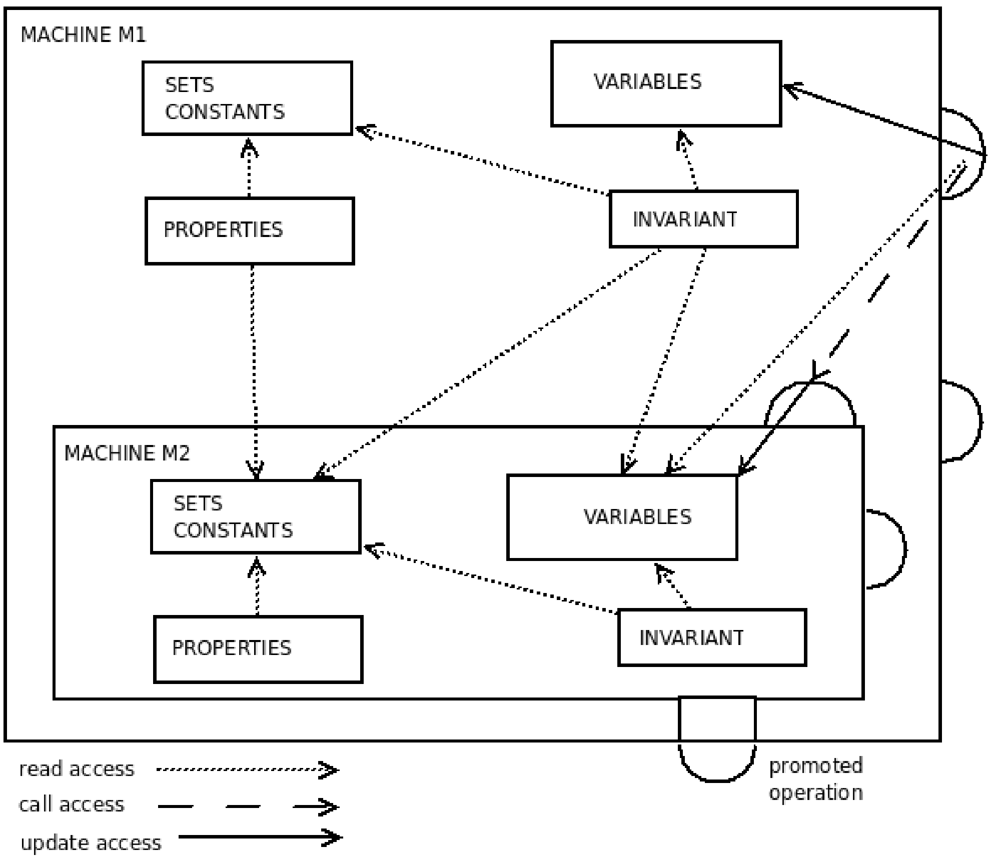


Figure 3.2 Includes clause and its relationships [10, figure 10.1]

However every operations of M1 must preserve its invariant (including possible promoted machines of M2). Of course every operations of M2 must preserve its own invariant as well.

If all operations are to be promoted, then we use the *EXTENDS* clause instead, because then M1 becomes an extension of M2, with a further invariant and operations of its own.

3.3.1.3 Refinement & implementation

Not every machine of a design aim to be refined. Indeed the refinement is done on operations. For instance a *seen* machine that contains only sets will not be refined and/or implemented. Composition clauses used for the abstract design can affect the refinement & implementation design.

On the other hand some, abstract machines may have operations, executing the instruction `skip`. This instruction does nothing! Why is that? For instance, we might need an operation 'main' to execute the program. This has no need for any abstract specification, however we need an implementation for it. There is not any implementation without an abstract machine. The abstract machine, here, serves as a 'header' file, containing operations definitions.

3.3.1.4 The IMPORTS clause

This clause is used for implementation machine, to help them using 'services' of other machines. A machine must be imported only once, though several instance of that machine are allowed. For instance an implementation I1 imports a machine M2. I1 will refer to M2 by adding a suffix: import1.M2 and import2.M2 for instance. The state of the machine identified by *import1* is different from import2. Every implementation machine will have IMPORTS, even to refer to basic libraries.

3.4 Basic libraries/machines

AtelierB provides some basic libraries to interact within the implementation level with the system. It provides for instance basic operations such as input or output of text. Such a machine is developed in a particular way. It has abstract specification in order to protect prohibited operations to be called within the implementation code (of another machine using this library), but has no implementation code itself. The code generation will produce operations headers without anything within their body. A developer can then fill the operations body with appropriate code.

AtelierB provides code for:

- BASIC_IO: this library handles input of text and output of text,
- BASIC_ARRAY_VAR: this library handles use of a one dimension array,
- BASIC_ARRAY_RGE: this library handles use of a two dimensions array.

There are only few available libraries due to a very specific (and thus very specific specification) need, different for every project. This means that this project might (actually, quite sure) need specific libraries to be developed with the method just described. The most common basic machines that will used and developed are machine implementing basic sets. Here is an example of a machine and its implementation implementing the `TRANS_AXIS` set:

```
/* BASIC_POWER_SWITCH
* Author: sylvain
* Creation date: Sun Jan 18 2009
*/
MACHINE
BASIC_TRANS_AXIS
SEES
SAFER_TYPE
CONSTANTS
encode_trans_axis, decode_trans_axis
PROPERTIES
encode_trans_axis ∈ TRANS_AXIS ⤖ (3,5) ∧
decode_trans_axis ∈ (3,5) ⤖ TRANS_AXIS ∧
dom(encode_trans_axis ▷ {3}) = ran({3} ◁ decode_trans_axis) ∧
dom(encode_trans_axis ▷ {4}) = ran({4} ◁ decode_trans_axis) ∧
dom(encode_trans_axis ▷ {5}) = ran({5} ◁ decode_trans_axis)
OPERATIONS
return ⟵ encode(axis) =
PRE
axis ∈ TRANS_AXIS
THEN
return :∈ (3,5)
END
END
```

```
   /* BASIC_TRANS_AXIS_imp
* Author:  sylvain
* Creation date:  Tue Jan 20 2009
*/
IMPLEMENTATION
BASIC_TRANS_AXIS_imp
REFINES
BASIC_TRANS_AXIS
SEES
SAFER_TYPE
VALUES
encode_trans_axis = {xx↦3,yy↦4,zz↦5};
decode_trans_axis = {3↦xx,4↦yy,5↦zz}
OPERATIONS
return ⟵ encode(axis)  =
BEGIN
return := encode_trans_axis(axis)
END
END
```

Here each element of the set is given an implementation value: an integer. The encoding is hided from user, who just uses the operation `encode`.

3.5 Code checking & proof

3.5.1 Type checking

This step is concerned with checking numerous things, mostly concerned with types of variables. Here is a non-exhaustive list of things type checking does:

- Verifies the data has been typed correctly and that the expressions used within predicates, expressions or substitution are of compatible types,
- Checks visibility rules: make sure access of data are allowed, depending if a machine has a write/read-only acccess to the data of another machine,
- Makes sure there is no ambiguity within the data.

These are the two main things type checking does. It precedes the generation of proof obligations (see **subsection 3.5.2**). Indeed it wouldn't make sense to generate proof obligations with mistyped code.

3.5.2 Proof obligations

Here is the definition gave by the author of the B-method[14]:

"A proof obligation is a mathematical formula to be proven, in order to ensure that a B component is correct."[1]

A proof obligations (also shortened with PO) is of the form, $H \Rightarrow P$, where H and P are predicate. This means P has to be proven under the hypotheses H. Notice that H is usually a concatenation of several predicate.

Let's illustrate that with an example. Here is the machine `example` (from [1]):

MACHINE

$Exemple$

VARIABLES

val

INVARIANT

$val \in \mathbb{N}$

OPERATIONS

$increment =$

$val := val + 1$

END

Notice that a proof obligation is generated if there are operations, since they are modifying the state of the machine. The proof obligation generated for this machine is: $val \in \mathbb{N} \Rightarrow val + 1 \in \mathbb{N}$. If it can be proved then this machine is labelled as correct.

[14] It is known in High Integrity Ada as verification condition.

Develop the SAFER sytem using B-Method

4.1 Introduction

As seen, the B-Method allows a system to be specified *and* implemented, thanks to the process of refinement. SAFER has been specified using PVS, which only allows formalisation of specification, as well as verification (see **subsection 2.1.3.2**). It is interseting to see how the B-Method can be used in this particular case that is SAFER.
Therefore this chapter will address the project itself, *i.e.* the development of the SAFER system using B-Method. Due to some problems in trying to use both B-method and tools associated, the initial version of the system was not developed. However the system produced resembles the original idea of the *Streamlined SAFER*, that is:

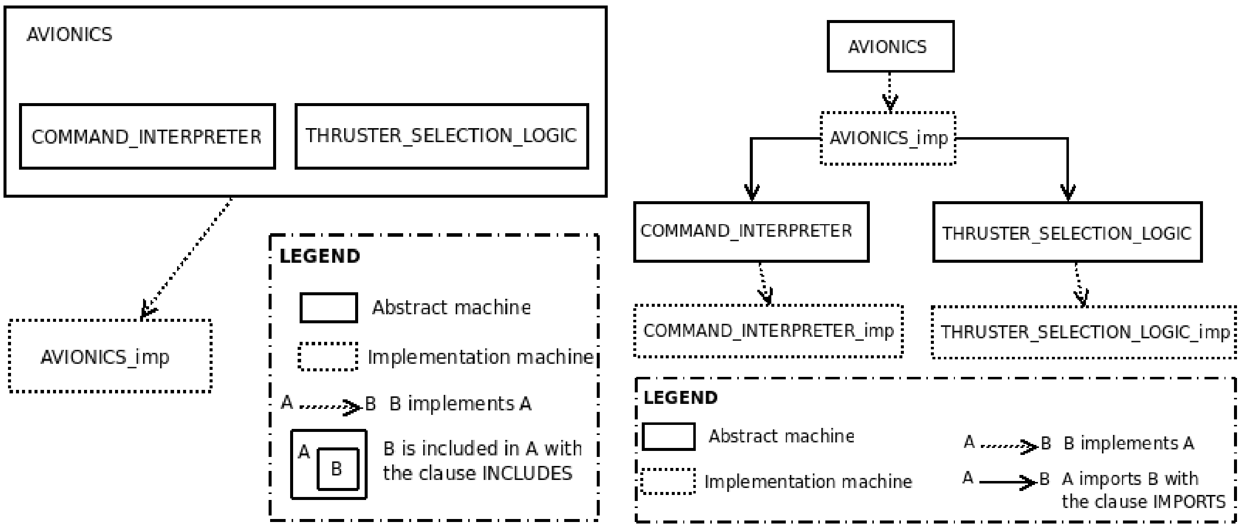
> The idea of the streamlined SAFER was to developed a correct code using B-method, in order to have a quick overview on the method, for a development process from specification with abstract machine, to code generation. This version of the SAFER includes only function to move forward and backward, and to switch on/off the HCM. Thus this SAFER has only one axis X, and 4 front, 4 back thrusters, as well as a switch on/off button. In order to simplify the development, the code is build around only one machine AVIONICS, that implements the functionalities described above. Indeed, the functionalities developed here were too simple to bother developing 4 machines (HCM_INPUT, COMMAND_INTERPRETER, THRUSTER_SELECTION_LOGIC, THRUSTER_OUTPUT). Besides the use of the different clauses to organize the specification is not obvious for who is not used to the B-method. The machine AVIONICS is of course implemented. This implementation uses numerous other machines in order to implement enumerated sets (such as those found in the machine SAFER_TYPE), one dimension array, etc.

The version produced will be presented in the next sections, as well as how it differs both from the Streamlined SAFER and the original one.

4.2 SAFER system

The **figure 2.4**, 4 modules are clearly visible within the avionics software. The system produced 'implements' only the *command interpreter* and the *thruster selection logic*. Modules highlighted in gray, are not explicitly present in the project as there are no hardware system to communicate with. For instance the *thruster command output* is replaced by a message giving which of the 24 thrusters are fired. From the Streamlined SAFER point of view, the system produced implements the same modules. However it tends to maximise the use of composition clauses such as `INCLUDES` or `IMPORTS`.

A B-method system is composed of abstract, refinement and implementation machines. Abstract machines aim to specified the system, and code is generated from implementation machines. Given the previous described modules of SAFER system, the task was to choose appropriate organisation and imbrication of machines. This is not an easy task, as a formal method like B-method doesn't follow a 'classical' type of organization due to the refinement process. A good idea is to be inspired by the modules given in the flowchart of the avionics software. Each module will be represented as an abstract machine. The problem is to decide at which level of abstraction a machine will use the other one. For instance, let's assume we have three machines `AVIONICS`, `COMMAND_INTERPRETER` and `THRUSTER_SELECTION_LOGIC`. There are roughly two types of organisation. They are presented in **figure 4.1**.



a: at abstract level b: at implementation level

Figure 4.1 Types of organisation

Organisation at implementation level seems more complicated as there are more machines. However the `AVIONICS` implementation machine will probably tend to be quite complex in organisation *(a)*. Therefore I chose to use the second one. This will lead to a significant amount of machines, but the code will probably find itself much more readable. Just like in object programming, a code is much more enjoyable when split into relevant and numerous classes rather than all in a main class.

4.2.1 SAFER architecture

Given the SAFER architecture in **section 2.2**, a design peculiar to the B-method was achieved (see **figure 4.2**). Several important things can be noticed on this figure:

- as shown in this figure, it is quite complicated. It just shows how machines are related to each other.
- the more 'basic' the machines are, the more used. Indeed a basic machine implements a basic set, used throughout the whole project, therefore they are used a lot.
- the schema cleary shows the main machine uses two modules: `COMMAND_INTERPRETER` and `THRUSTER_SELECTION`. We will see these two machines in details later on, as well as the other machines, in order to explain what they implement.

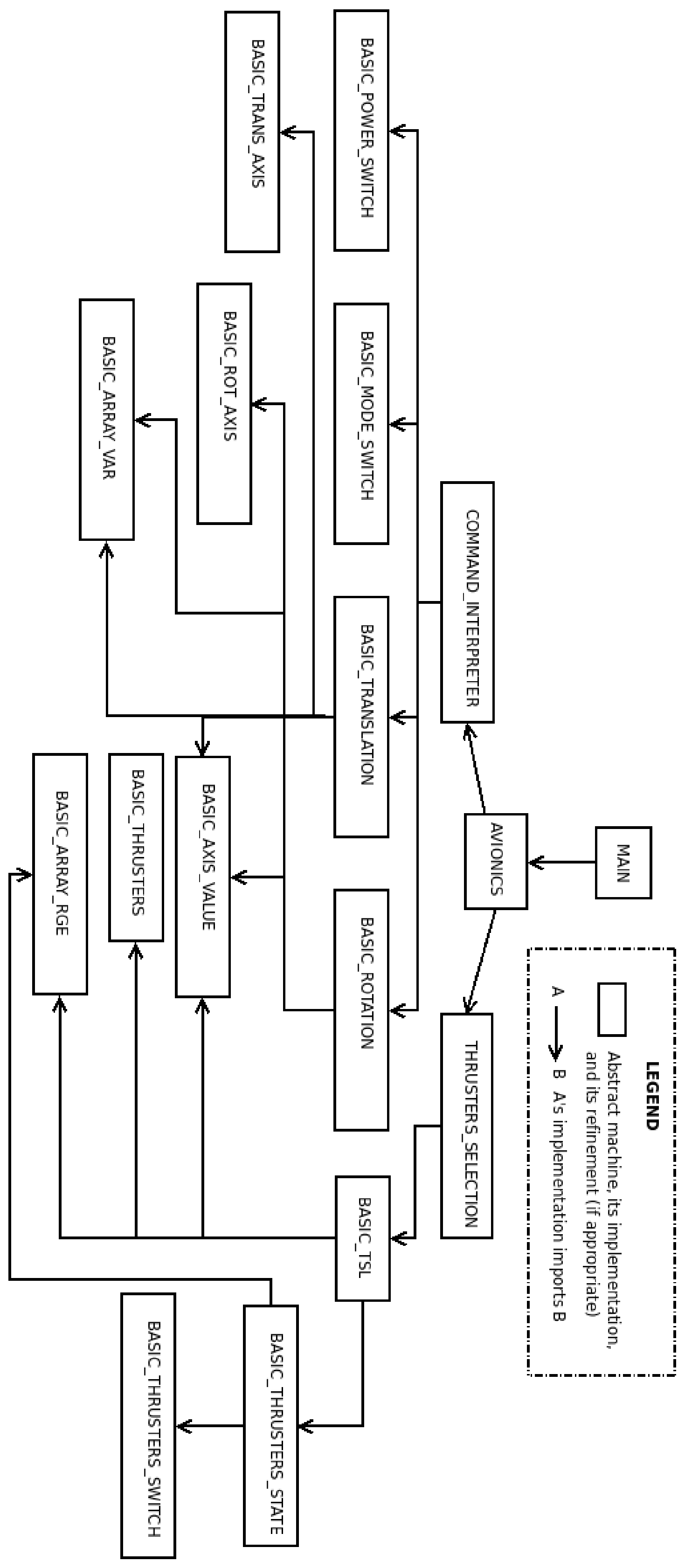


Figure 4.2 Simplified schema of the software design (see **appendix A** for the complete design with implementation machines.)

4.2.2 Previous planned design

Originally, it was planned to developed the machines presented below. The requirements that are present here refer to the requirement present in **appendix B**

- SAFER_TYPE: contains all types that will be used througout the software development such as the type of thrusters, the type of input and so on. All the machines described below see this machine.
- HCM_INPUT: is concerned with providing an interface between the machine COMMAND_INTERPRETER and the switches of the HCM.

  Specification: as stated in **appendix B**, requirement 11. That is, the avionics software shall accept the following data from the hand controller module.
- COMMAND_INTERPRETER: aims to translate commands received from the HCM_INPUT into appropriate output.

  Specification: as stated in **appendix B**, requirements 7 and 8.
- IRU_INPUT: is concerned with providing an interface between the machine STATE_ESTIMATOR and the IRU system that senses rates and temperatures of thrusters.

  Specification: as stated in **appendix B**, requirement 16.
- STATE_ESTIMATOR: aims to interpret the temperatures and rates values provided by the IRU_INPUT. **Specification:** as stated in **appendix B**, requirement 12.
- AAH: specifies the Automatic Attitude Hold.

  Specification: as stated in **appendix B**, requirements 3,4,5 and 6.
- THRUSTER_SELECTION: this machine aims to provide which thruster must be switch off or switch on, regarding the thruster selection table provided in the previous document (table 2.1 and 2.2).

  Specification: as stated in **appendix B**, requirements 9 and 10.
- THRUSTER_OUPUT: aims to provide an interface between the THRUSTER_SELECTION machine and the thruster system.

  Specification: as stated in **appendix B**, requirement 14.

For numerous reasons, this design, and therefore these machines were not all implemented. The two main we are interested in are THRUSTER_SELECTION and COMMAND_INTERPRETER. Besides all requirements haven't been all implemented in these machines. This is mainly due to problems encountered with AtelierB (see **section 4.4**) and mastering the B method.

4.3 Description of machines

This section will address important modules of the SAFER system that have been developed. It will not detailed all machines present in **figure 4.2**.

Note: *It is important to often look at* **figure 4.2** *in order to get a good overview of the relationships between machines, like importation for instance.*

4.3.1 SAFER TYPE

The `SAFER_TYPE` machine includes types, that is basic sets that will be used throughout the whole project. The sets are presented below, they are defined in extension, like every basic sets.
The sets below represent information related to axis. `TRANS_AXIS`[15] for information related to transational axis, and `ROT_AXIS` for rotational axis. `AXIS_VALUE` defines values of each axis as defined in **subsection 2.2.2**, that is positive, negative or no thrust at all. It will be useful to construct relations involving `TRANS_AXIS` and/or `ROT_AXIS` to formalize a specification, we will see how it is done later on.

- `AXIS_VALUE` $= \{zero, neg, pos\}$
- `TRANS_AXIS` $= \{xx, yy, zz\}$
- `ROT_AXIS` $= \{roll, pitch, yaw\}$

These two sets represent switches, therefore only two values define them.

- `POWER_SWITCH` $= \{on, off\}$
- `MODE_SWITCH` $= \{tran, rot\}$

Here are represented sets related to thrusters: first the set defining all thrusters by their name (see **figure 2.3**), and second the set that defines it a thruster is in state fire or not. Notice that we could have use the set `POWER_SWITCH` to implement that. However, despite the fact their carry the same information, they are not used within the same context. It is important to differentiate those two, in order to avoid confusion or even mistakes within the specification. This is usually a good policy. Besides a warning is to be given on big sets, like `THRUSTERS`, it can increase considerably the time of prooving, depending on the relation it is involved into.

- `THRUSTERS` $= \{$
  $b1, b2, b3, b4,$
  $f1, f2, f3, f4,$
  $l1r, l1f, r2r, r2f,$
  $l3r, l3f, r4r, r4f,$
  $d1r, d1f, d2r, d2f,$
  $u3r, u3f, u4r, u4f$
  $\}$
- `THRUSTERS_SWITCH` $= \{fired, notfired\}$

From now on every machine 'sees' `SAFER_TYPE` and therefore has read access to these sets.

[15] The axis are represented with double letters xx for the X axis, yy for Y axis and so on. This is just a restriction of the B syntax that forbids the use of single letter as identifier, variables, or even elements of basic set.

4.3.2 COMMAND INTERPRETER

This section will present the abstract machine, and its implementation, for the complete code see **appendix C.2** *and* **appendix C.3**.

As stated in the requirements the command interpreter machine shall respond properly to input given by the `HCM_INPUT` machine[16]. The machine includes several operations allowing to control the SAFER system, like `switch_on`, `switch_off`, etc.

- **Abstract machine**

  – **Variables:** this machine includes 4 variables representing different states of the machine: `power_switch`, `mode_switch`, `translation`, `rotation`. The two latter represent value of each axis. They are typed as follow:

  ⋆ `power_switch` $\in$ `POWER_SWITCH`
  ⋆ `mode_switch` $\in$ `MODE_SWITCH`
  ⋆ `translation` $\in$ `TRANS_AXIS` $\longrightarrow$ `AXIS_VALUE`
  ⋆ `rotation` $\in$ `ROT_AXIS` $\longrightarrow$ `AXIS_VALUE`

  `translation` $\in$ `TRANS_AXIS` $\longrightarrow$ `AXIS_VALUE` is a total function from $\{xx, yy, zz\}$ to $\{zero, pos, neg\}$. That means each axis is assigned a single value at every instance of time. The relation `rotation` works in a similar way.
  Six variables could have been used for the two latter variables: `translation_xx`, `translation_yy`, `translation_zz` and `rotation_xx`, `rotation_yy`, `rotation_zz`. Each of these variable would have been of the type `AXIS_VALUE`. However it has not been used because it is less powerful in terms of specification. It is much easier to manipulate a relation than 3 differents variables. Indeed much more constraints can be expressed, like cardinality for instance.

  – **Invariant:** variables types were addressed above The next part of the invariant is concerned with formalizing a specification:

  ⋆ If the power is switched off, then all commands have to be set to zero:
  (`power_switch = off` $\Rightarrow$ `ran(translation) =` $\{$`zero`$\}$ $\wedge$ `ran(rotation) =` $\{$`zero`$\}$)
  `ran` represents the range of a relation. This means that if `power_switch` is off, it implies that all translation and rotation are equal to zero.
  ⋆ If a rotational command is not null, then all translation commands have to be suppress
  $\forall$ `aa` $\in$ `ROT_AXIS` $\Rightarrow$ (`ran(`$\{$`aa`$\}$ $\lhd$ `rotation)` $\neq$$\{$`zero`$\}$ $\Rightarrow$ `ran(translation) =` $\{$`zero`$\}$))

[16] `HCM_INPUT` is not implemented here as it is a really simple machine, and because there is no proper input like they would be on an actual SAFER, the input are made within the main function.

For every rotational axis, if one does input a negative or positive thrust then all translation axis have to be equal to zero. $\lhd$ is a restriction on the domain of the relation given aa.

- ⋆ At most one translation command shall be on, with priority given in this order X, Y and Z: $(card(ran(translation \rhd \{pos, neg\})) <= 1)$
  card is the cardinality of a set and $\rhd$ is a restriction on the range of a relation.
  It is interesting to notice that not everything can be expressed in this abstract machine. This is the case of the priority requirement here. It is at a low level that it would be expressed, such as in implementation (not necessarily the implementation of that machine). A possible solution would be to use an `IF THEN ELSE` structure in the order given above.

Constraints related to this particular machine were written in the invariant clause, it now has to be respected by the following operations

— **Operations:**

- ⋆ `is_powered_off`: check whether the SAFER is switched off or not,
- ⋆ `is_mode_tran`: check whether the SAFER is in translation mode or not,
- ⋆ `switch_off`: switch off the SAFER if it is not already off,
- ⋆ `switch_on`: switch on the SAFER if it is not already on,
- ⋆ `switch_mode_tran`: switch to translation mode,
- ⋆ `switch_mode_rot`: switch to rotational mode,
- ⋆ `translate_on_xx(value)`: input a thrust zero, positive or negative for the X axis,
- ⋆ `translate_on_yy(value)`: input a thrust zero, positive or negative for the Y axis,
- ⋆ `translate_on_zz(value)`: input a thrust zero, positive or negative for the Z axis.

These operations hide how they are implemented. The action of switching on the SAFER will be hided from the calling operation, and so will it be for the switching off operations, and the other ones.
Moreover the two former operations `is_powered_off` and `is_mode_tran` aim to be used to check the preconditions of some operations. For instance the SAFER cannot be switched on if it is already on, and cannot be switched off if it is already off. These are the preconditions of use of these two operations. It is usually a good policy two prohibit things that would not be allowed within a given context, the use of the SAFER hand maneuvering control for instance.

- **Refinement machine**

The refinement machine here is quite straight forward and will not be detailed here. It just introduces sequencing in replacement of parallelism.

- **Implementation machine**

Here we will focus on how the above specification has been implemented. As seen in **chapter 3**:

— an implementation machine uses 'services' of other machines in order to implement its own operations

— an implementation machine has no state of its own.

Therefore in order to implement the various state holded by the four variables `power_switch`, `mode_switch`, `translation` and `rotation`, a number of basic machines have been developed. There are:

— `BASIC_POWER_SWITCH` and `BASIC_MODE_SWITCH`: they do more than just implementing basic sets into implementable values, they hold the state of the machine as well. That means for instance that the value of `power_switch` is contained into the machine `BASIC_POWER_SWITCH` and not within the implementation. That is why the operation `switch_on`, for example, of the implementation machine, calls operation `switch_on` from `BASIC_POWER_SWITCH` whereas it was modifying its own state in the abstract machine.
Let's clarify this! Here is the operation `switch_on` as it is in the abstract machine:

`switch_on` = PRE `power_switch` $\neq$ `on` THEN `power_switch` := `on` END

and here is the implemented operation:

`switch_on` = THEN `power.switch_on` END

`power` refers to the imported machine, declared as follows:

IMPORTS `power.BASIC_POWER_SWITCH`.

The implementation using `BASIC_MODE_SWITCH` will not be detailed here. Just notice the variable `mode_switch` is implemented in the same way as `power_switch` and so are operations related to it.

— `BASIC_TRANSLATION` and `BASIC_ROTATION` implement relation respectively from `TRANS_AXIS` and `ROT_AXIS` towards `AXIS_VALUE`. Only the first relation will be addressed, as they both work the same way (for the complete code of both machines see **appendix C**).
`BASIC_TRANSLATION` is just a simple machine implementing the following relation: `TRANS_AXIS` $\longrightarrow$ `AXIS_VALUE`. The two basic sets involved are used as they are, it is in the implementation of `BASIC_TRANSLATION` that they would be 'encoded' using `BASIC_TRANS_AXIS` and `BASIC_AXIS_VALUE`.
Now that we find a way to implement the two basic sets we need, a mechanism to implement a total function has to be found. This is where the basic libraries provided by the software Atelierb are to be used. The one that we are interested in is `BASIC_ARRAY_VAR`, it implements a total function with use of an array, indeed it is the perfect structure for that type of relation. The structure will of course depends on the type of relation. The more complicated the relation the harder to implement. For instance a partial function will not easily be implemented.

4.3.3 THRUSTER SELECTION

This section will present the abstract machine, and its implementation, for the complete code see **appendix C.4** *and* **appendix C.5**. *We will not address refinement step as there is not any in this module.*
This machine is concerned with dealing with input from the `COMMAND_INTERPRETER` machine. It has to rely on the thruster selection logic (see **table 2.1** & **table 2.2**) to fire proper thrusters (limited to four at any time).

It is quite hard to find a structure to express these two tables. A solution would be to go straight to implementation and make it easier, however the abstraction would loose a bit of its appeal. Another solution would be to use several relations overlapped with each other. Indeed for the selection logic, there are a source set, which is the combination of all possible input, and a target set which will be a combination of 4 thrusters. So the relation would be:

"Combination of input" $\longrightarrow$ "Combination of 4 thrusters"

Let's detail both sets:

- The combination of possible inputs is a component of 4 variables as seen in **chapter 2.2**. There are 3 set of variables: $\{$X,Pitch,Yaw,Roll$\}$, $\{$Y,Pitch,Yaw,Roll$\}$ and $\{$Z,Pitch,Yaw,Roll$\}$. The set of all possible inputs will depend on the thrust of each variable: positive, negative or no thrust. Thus the size of possible inputs for a set of variables is 81, and 243 for the whole range of inputs. The set would then be:

  $(\texttt{TRANS\_AXIS} \longrightarrow \texttt{AXIS\_VALUE}) \times (\text{pitch} \longrightarrow \texttt{AXIS\_VALUE}) \times (\text{yaw} \longrightarrow \texttt{AXIS\_VALUE}) \times (\text{roll} \longrightarrow \texttt{AXIS\_VALUE})$

  This is a *huge* set and it would be hard to verify that it respects the invariant, indeed the proover provided in the software might not be so powerful to deal with so many cases.

- The combination of thrusters could be the powerset of THRUSTERS. However each set has to be limited to 4 elements, and powerset does not provide such a guarantee. Besides it would include combination of thrusters that are not used anyway and would be a waste of time during the proof step. A solution is to explicitly define the set of possible combination of thrusters.

The set of elements from the relation being far too much big, leading to time of prooving considerably long, hours or maybe even days, it has been decided to implement only a part of the selection logic table. That is to only consider input on one axis at a time. For instance what thrusters should be fired if only X axis has an input from the COMMAND INTERPRETER? The selection logic considered is shown on **table 4.1**.

- **Abstract machine**

  Given what was said above, an abstract structure was 'implemented'. The variables tsl_axis_xx, tsl_axis_yy and tsl_axis_zz express that table with the mean of a total function. Notice that the set of combination of thrusters is reduced, for each variable, to the only possible set of thrusters that can be fired. This re-enforce the specification and make gain some time when prooving the machine.
  Now we need a structure to record the state of each thrusters (fired or not fired) just as it is in COMMAND INTERPRETER with TRANS_AXIS and AXIS_VALUE for instance. So we use the same relation, a total function, THRUSTERS $\longrightarrow$ THRUSTERS_SWITCH, on which we can apply the constraint "no more than 4 thrusters fired at any time".

X	Thrusters to fire
-	B1,B2,B3,B4
+	F1,F2,F3,F4
∅	

Y	Thrusters to fire
-	L1R,L3R,L1F,L3F
+	R2R,R4R,R2F,R4F
∅	

Z	Thrusters to fire
-	U3R,U4R,U3F,U4F
+	D2R,D1R,D1F,D2F
∅	

Table 4.1 Selection logic table implemented

Here express the specification in an abstract way was not easy, and some concession had to be made. Now we will see how the implementation of that machine was made.

- **Implementation**

  The structure to be used in the implementation machine was as well part of the decision to simplify the abstract structure of thruster selection logic. Indeed some basic libraries (usually provided with AtelierB) where not included in the software, which limited the use of libraries to `BASIC_ARRAY_VAR`, `BASIC_ARRAY_RGE` and `BASIC_IO`. It is too few to implement a correct system, although sufficient for this version of the SAFER.
  In order to implement the relation given by variables `tsl_axis_xx`, `tsl_axis_yy` and `tsl_axis_zz` a basic machine `BASIC_TSL` was developed, `TSL` stands for thruster selection logic. It uses `BASIC_ARRAY_RGE` in the implementation. The link invariant of `BASIC-TSL` is quite complicated and will not be addressed here (it can be found in appendix ??).

Normally `THRUSTER SELECTION` is supposed to pass information to `THRUSTER OUTPUT`. As no actual SAFER is available, the output which was meant to be thrusters firing will just be message displaying in the terminal.

4.3.4 AVIONICS

Now that the two most important machines have been addressed, it is necessary to see how their are effectively used within a calling machine. This is what the `AVIONICS` machine does. First of all you would notice that the code of the abstract machine is very short. Indeed `AVIONICS` is a mean to call operations from `THRUSTER SELECTION` and `COMMAND INTERPRETER`, this can be done only in an implementation. As can be seen on the code of the implementation machine there are a lot of control structure. Operations calling imported operations make sure the preconditions are respected before any usage. This is all there is to say about `AVIONICS`, it controls the good usage of services provided by `THRUSTER SELECTION` and `COMMAND INTERPRETER`.

An important point to notice is that the call of an operation `translate_on_AXIS`, changing the state of the machine `COMMAND INTERPRETER` is not linked to the operation `fire_thrusters_AXIS`, changing state of machine `THRUSTER SELECTION`. Both machines refer to 2 different states whereas they should share the same. For instance the machine `COMMAND INTERPRETER` modifies the state of the variable translation. This variable is bounded by an invariant so its consistency regarding the specification can be check throughout the whole project. A good thing to do would be to make `THRUSTER SELECTION` use this variable to decide which thrusters to fire regarding its thruster selection table, so that to make sure the invariant holds. However due to a problem related to the structure chosed at the beginning of the project, it was not possible to achieve that implementation. Indeed to imported machine have no access read or write to another imported machine, so `THRUSTER SELECTION` cannot access state of `COMMAND INTERPRETER`. The design chosed was probably not the right one.

4.4 B-Method tool: AtelierB

In order to develop a working SAFER system. A software able to handle B language, type-checking, obligation proofs generation[17], automatic and interactive proof, as well as code generation. AtelierB, the tool developed by ClearSy, includes these features. However the free version of this software was in beta development at the time I used it[18]. This led to some installation and usage problems, and considerabley slowed the developement process down.

4.5 Conclusion & errors

Several errors where made on this project. Just like every project carrying on a bit of research do not necessarily go well all the time. A first mistake was probably to rely too much on the AtelierB software, which was not really in an usable version[19]. Missing libraries, steady crashes did not help to perform a significant and satisfactory piece of work. The second mistake would be related with the B design sustained. It probably would have been more relevant to use the abstract level design using `INCLUDE` clauses for the reason explained in **section 4.3.4**. `INCLUDE` clause allow two machines included to have access to each other state.
Despite these mistakes the project carried out works fine, but could be easily improved.

[17] also called verification conditions
[18] which is not the case anymore at the time I am writing these lines
[19] This is still the case with the new release

Comparative works

Several team had worked on the SAFER project, and several different method have been used. No access to these studies were given, just their quick respective report allow to draw some conclusions here. The two method are PVS (see **section 2.1.3.2**) and VDM see **section 2.1.3.3**).

Conclusion & future work

References

[1] *Proof Obligations,Reference Manual v3.7*.

[2] J. Abrial, *The B-Book, Assigning programs to meanings*. (1996).

[3] J. Crow, *Formal methods specification and analysis guidebook for the verification of software and computer systems, volume ii: A practitioner's companion* Tech. Rep., NASA (1997).

[4] H. Habrias Spécifications avec b. (2007).

[5] A. Harry, *Formal methods, Fact File, VDM and Z*. (1996).

[6] E. Jaeger, *Remarques relatives à l'emploi des méthodes formelles (déductives) en sécurité des systèmes d'information* Tech. Rep., Direction centrale de la sécurité des systèmes d'information (2008).

[7] J. L. Lions, *Ariane 5, flight 501 failure, report by the inquiry board* Tech. Rep., European Space Agency (1996).

[8] S. Owre(2008).

[9] N. Plat and P. G. Larsen, *An overview of the iso/vdm-sl standard* Tech. Rep., University of Leicester (1992).

[10] S. Schneider, *The B-method, an introduction*. (2001).

APPENDICES
NASA's SAFER using B-Method

Sylvain Verly

Supervisor: Andrew Ireland

Second reader: David Corne

Final Year Dissertation

Honours Degree

Schema of the software design

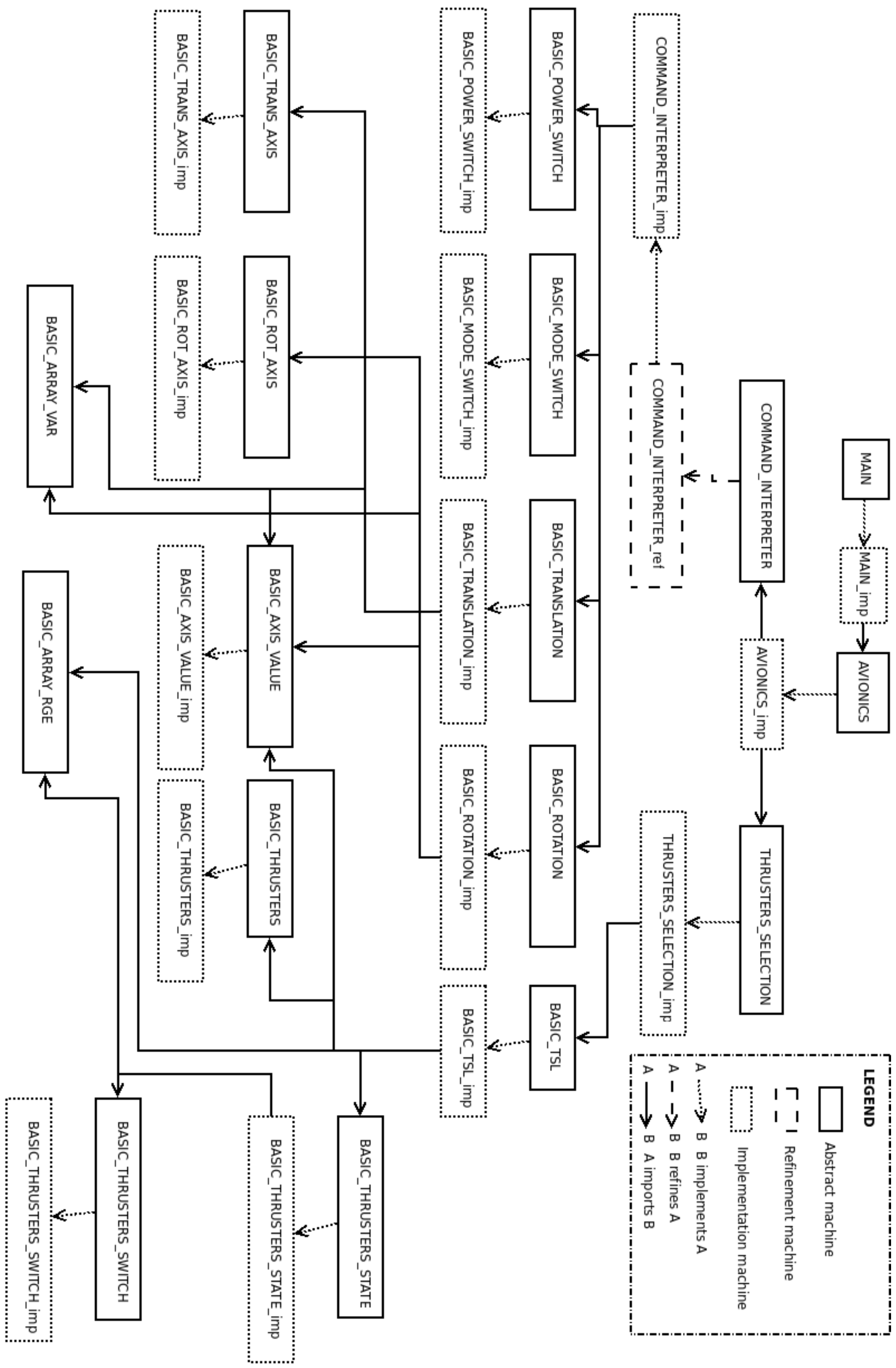


Figure A.1 Schema of the software design.

Requirements

Here are presented the requirements that were in the design document.

1. The avionics software shall reference all commands and maneuvers to the coordinate system defined in **figure 2.3**.
2. The avionics software shall provide a six degree-of-freedom maneuvering control capability in response to crewmember-initiated commands from the hand controller module.
3. The avionics software shall allow a crewmember with a single command to cause the measured SAFER rotation rates to be reduced to less than 0.3 degree per second in each of the three rotational axes.
4. The avionics software shall attempt to maintain the calculated attitude within ± 5 degrees of the attitude at the time the measured rates were reduced to the 0.3 degree per second limit.
5. The avionics software shall disable AAH on an axis if a crewmember rotation command is issued for that axis while AAH is active.
6. Any hand controller rotation command present at the time AAH is initiated shall subsequently be ignored until a return to the off condition is detected for that axis or until AAH is disabled.
7. Hand controller rotation commands shall suppress any translation commands that are present, but AAH-generated rotation commands may coexist with translations.
8. At most one translation command shall be acted upon, with the axis chosen in priority order X, Y, Z.
9. The avionics software shall provide accelerations with a maximum of four simultaneous thruster firing commands.
10. The avionics software shall select thrusters in response to integrated AAH and crew-generated commands according to **table 2.1** and **table 2.2**.
11. The avionics software shall accept the following data from the hand controller module:
    - + pitch
    - - pitch
    - + X, - X
    - + yaw or + Y, - yaw or - Y
    - + roll or + Z, - roll or - Z
    - Power switch
    - Mode switch

- Display proceed switch
- AAH pushbutton

12. The avionics software shall provide flight control for AAH using the IRU-measured rotation rates and rate sensor temperatures.
13. The avionics software shall accept the following data from the inertial reference unit:
    - Roll, pitch, and yaw rotation rates
    - Roll, pitch, and yaw sensor temperatures
    - X, Y, and Z linear accelerations
14. The avionics software shall provide the following data to the valve drive assemblies for each of the 24 thrusters:
    - Thruster on/off indications
15. The avionics software shall provide the following data to the HCM for display:
    - Rotation rates and displacements
    - Miscellaneous status messages
16. The avionics software shall provide the following data to the data recorder assembly:
    - IRU-sensed rotation rates
    - IRU-sensed linear accelerations
    - IRU rate sensor temperatures
    - Angular displacements
    - AAH command status

Requirements 1 to 14 are high level requirements whereas requirements 15 and 16 are low requirements.

Code

C.1 SAFER TYPE

```
/* SAFER_TYPE
* Author:  sylvain
* Creation date:  Wed Jan 14 2009
*/
MACHINE
SAFER_TYPE
SETS
AXIS_VALUE = {zero,neg,pos};
TRANS_AXIS = {xx,yy,zz};
ROT_AXIS= {roll,pitch,yaw};
POWER_SWITCH = {on,off};
MODE_SWITCH= {tran,rot};
THRUSTERS = {
b1,b2,b3,b4,
f1,f2,f3,f4,
l1r,l1f,r2r,r2f,
l3r,l3f,r4r,r4f,
d1r,d1f,d2r,d2f,
u3r,u3f,u4r,u4f
};
THRUSTERS_SWITCH = {fired,notfired}
END
```

C.2 COMMAND INTERPRETER

```
/* COMMAND_INTERPRETER
* Author:  sylvain
* Creation date:  Wed Jan 14 2009
*/
MACHINE
COMMAND_INTERPRETER
SEES
SAFER_TYPE
VARIABLES
```

```
power_switch, mode_switch,
translation, rotation
INVARIANT
/* Power switch button */
power_switch ∈ POWER_SWITCH  ∧
/* Mode switch button */
mode_switch ∈ MODE_SWITCH  ∧
/* A transition is either null, negative  ∨  positive on every axis */
translation ∈ TRANS_AXIS ⟶ AXIS_VALUE  ∧
/* A rotation is either null, negative  ∨  positive on every axis */
rotation ∈ ROT_AXIS ⟶ AXIS_VALUE  ∧

/* If the power is switched off, then all commands have to be set to zero */
(power_switch = off ⇒ ran(translation) = {zero}  ∧  ran(rotation) = {zero})  ∧
/* If a rotational command aa is ¬ null, then all translation commands have to be suppress */
∀ (aa).(aa:ROT_AXIS ⇒ (ran({aa} ◁ rotation) ≠ {zero} ⇒ ran(translation) = {zero}))  ∧
/* At most one translation command shall be on (with priority order X, Y, Z but this cannot be for-
malized here) */
(card(ran(translation ⩥ {zero})) ≤ 1)
INITIALISATION
power_switch :∈ POWER_SWITCH  ||
mode_switch :∈ MODE_SWITCH  ||
translation := {(xx↦zero),(yy↦zero),(zz↦zero)}  ||
rotation := {(roll↦zero),(pitch↦zero),(yaw↦zero)}

OPERATIONS
return ⟵ is_powered_off =
BEGIN
return := bool(power_switch = off)
END;

return ⟵ is_mode_tran =
BEGIN
return := bool(mode_switch = tran)
END;

/* Switch off operation */
switch_off =
PRE
power_switch ≠ off
THEN
translation := TRANS_AXIS * {zero}  ||
rotation := ROT_AXIS * {zero}  ||
power_switch := off
END;
```

```
/* Switch on operation */
switch_on =
PRE
power_switch ≠ on
THEN
power_switch := on
END;

/* Switch mode into translation mode */
switch_mode_tran =
PRE
mode_switch ≠ tran
THEN
mode_switch := tran
END;
/* Switch mode into translation mode */
switch_mode_rot =
PRE
mode_switch ≠ rot
THEN
mode_switch := rot
END;

/* Translation functions */
translate_on_xx(value) =
PRE
value ∈ AXIS_VALUE  ∧
value ≠ zero  ∧
power_switch = on
THEN
rotation := {(roll↦zero),(pitch↦zero),(yaw↦zero)}  ||
translation := {(xx↦value),(yy↦zero),(zz↦zero)}
END;

translate_on_yy(value) =
PRE
value ∈ AXIS_VALUE  ∧
value ≠ zero  ∧
power_switch = on  ∧
mode_switch = tran
THEN
rotation := {(roll↦zero),(pitch↦zero),(yaw↦zero)}  ||
translation := {(xx↦zero),(yy↦value),(zz↦zero)}
END;

translate_on_zz(value) =
PRE
value ∈ AXIS_VALUE  ∧
value ≠ zero  ∧
power_switch = on  ∧
mode_switch = tran
THEN
rotation := {(roll↦zero),(pitch↦zero),(yaw↦zero)}  ||
translation := {(xx↦zero),(yy↦zero),(zz↦value)}
END;

/* Rotate functions */
rotate_on_roll(value) =
PRE
value ∈ AXIS_VALUE  ∧
value ≠ zero  ∧
power_switch = on  ∧
mode_switch = rot
THEN
translation := {(xx↦zero),(yy↦zero),(zz↦zero)}  ||
rotation := {(roll↦value),(pitch↦rotation(pitch)),(yaw↦rotation(yaw))}
END;
rotate_on_pitch(value) =
PRE
value ∈ AXIS_VALUE  ∧
value ≠ zero  ∧
power_switch = on
THEN
translation := {(xx↦zero),(yy↦zero),(zz↦zero)}  ||
rotation := {(roll↦rotation(roll)),(pitch↦value),(yaw↦rotation(yaw))}
END;

rotate_on_yaw(value) =
PRE
value ∈ AXIS_VALUE  ∧
value ≠ zero  ∧
power_switch = on  ∧
mode_switch = rot
THEN
translation := {(xx↦zero),(yy↦zero),(zz↦zero)}  ||
rotation := {(roll↦rotation(roll)),(pitch↦rotation(pitch)),(yaw↦value)}
END
END
```

C.3 COMMAND INTERPRETER implementation

```
/* COMMAND_INTERPRETER_imp
* Author:  sylvain
* Creation date:  Tue Jan 20 2009
*/
IMPLEMENTATION
COMMAND_INTERPRETER_imp
REFINES
COMMAND_INTERPRETER_ref
bf >lor[Bkeyword]SEES
SAFER_TYPE
IMPORTS
power.BASIC_POWER_SWITCH,
mode.BASIC_MODE_SWITCH,
translation.BASIC_TRANSLATION,
rotation.BASIC_ROTATION

INVARIANT
translation.relation = translation_ref  ∧
rotation.relation = rotation_ref  ∧
power.switch = power_switch_ref  ∧
mode.switch = mode_switch_ref
INITIALISATION
power.switch_off;
mode.switch_tran;

translation.INIT_RELATION;
rotation.INIT_RELATION

OPERATIONS
return ⟵ is_powered_off =
VAR sw IN
sw ⟵ power.is_powered_off;
return := bool(sw = TRUE)
END;

return ⟵ is_mode_tran =
VAR md IN
md ⟵ mode.is_mode_tran;
return := bool(md = TRUE)
END;

/* Switch off operation */
switch_off =
BEGIN
translation.INIT_RELATION;
rotation.INIT_RELATION;
```

```
power.switch_off
END;

/* Switch on operation */
switch_on =
BEGIN
power.switch_on
END;

/* Switch mode into translation mode */
switch_mode_tran =
BEGIN
mode.switch_tran
END;
/* Switch mode into translation mode */
switch_mode_rot =
BEGIN
mode.switch_rot
END;

/* Translation functions */
translate_on_xx(value) =
BEGIN
rotation.INIT_RELATION;
translation.SET_AXIS_VALUE(yy,zero);
translation.SET_AXIS_VALUE(zz,zero);
translation.SET_AXIS_VALUE(xx,value)
END;

translate_on_yy(value) =
BEGIN
rotation.INIT_RELATION;
translation.SET_AXIS_VALUE(xx,zero);
translation.SET_AXIS_VALUE(zz,zero);
translation.SET_AXIS_VALUE(yy,value)
END;

translate_on_zz(value) =
BEGIN
rotation.INIT_RELATION;
translation.SET_AXIS_VALUE(xx,zero);
translation.SET_AXIS_VALUE(yy,zero);
translation.SET_AXIS_VALUE(zz,value)
END;

/* Rotate functions */
```

```
rotate_on_roll(value) =
BEGIN
translation.INIT_RELATION;
rotation.SET_AXIS_VALUE(roll,value)
END;
rotate_on_pitch(value) =
BEGIN
translation.INIT_RELATION;
rotation.SET_AXIS_VALUE(pitch,value)
END;

rotate_on_yaw(value) =
BEGIN
translation.INIT_RELATION;
rotation.SET_AXIS_VALUE(yaw,value)
END
END
```

C.4 THRUSTER SELECTION

```
/* THRUSTER_SELECTION
* Author:  sylvain
* Creation date:  Sun Feb 15 2009
*/
MACHINE
THRUSTER_SELECTION
bf >lor[Bkeyword]SEES
SAFER_TYPE
VARIABLES
tsl_axis_xx, tsl_axis_yy, tsl_axis_zz, thrusters_state_map
INVARIANT
tsl_axis_xx ∈ AXIS_VALUE ⟶ {∅,{b1,b2,b3,b4},{f1,f2,f3,f4}}
tsl_axis_xx =
{(zero↦∅),
(pos↦{f1,f2,f3,f4}),
(neg↦{b1,b2,b3,b4})}  ∧
tsl_axis_yy ∈ AXIS_VALUE ⟶ {∅,{l1r,l3r,l1f,l3f},{r2r,r4r,r2f,r4f}}  ∧
tsl_axis_yy =
{(zero↦∅),
(pos↦{r2r,r4r,r2f,r4f}),
(neg↦{l1r,l3r,l1f,l3f})}  ∧

tsl_axis_zz ∈ AXIS_VALUE ⟶ {∅,{d2r,d1r,d1f,d2f},{u3r,u4r,u3f,u4f}}  ∧
tsl_axis_zz =
{(zero↦∅),
(pos↦{d2r,d1r,d1f,d2f}),
(neg↦{u3r,u4r,u3f,u4f})}  ∧
thrusters_state_map ∈ THRUSTERS ⟶ THRUSTERS_SWITCH  ∧
/* No more than 4 thrusters fired at any instance of time */
card(dom(thrusters_state_map ▷ {fired})) ≤ 4
INITIALISATION
tsl_axis_xx :=
{(zero↦∅),
(pos↦{f1,f2,f3,f4}),
(neg↦{b1,b2,b3,b4})}  ||
tsl_axis_yy :=
{(zero↦∅),
(pos↦{r2r,r4r,r2f,r4f}),
(neg↦{l1r,l3r,l1f,l3f})}  ||
tsl_axis_zz :=
{(zero↦∅),
(pos↦{d2r,d1r,d1f,d2f}),
(neg↦{u3r,u4r,u3f,u4f})}  ||
thrusters_state_map := THRUSTERS * {notfired}
OPERATIONS
/*This function is just a sample of what should be the final function */
fire_thrusters_xx(axis_value) =
PRE
axis_value ∈ AXIS_VALUE
THEN
skip
END;

/*This function is just a sample of what should be the final function */
fire_thrusters_yy(axis_value) =
PRE
axis_value ∈ AXIS_VALUE
THEN
skip
END;

/*This function is just a sample of what should be the final function */
fire_thrusters_zz(axis_value) =
PRE
axis_value ∈ AXIS_VALUE
THEN
skip
/*thrusters_state_map := (THRUSTERS-tsl_axis_zz(axis_value))*{notfired} ∪
(tsl_axis_zz(axis_value)*{fired})*/
END
```

```
END
```

C.5 THRUSTER SELECTION implementation

```
/* THRUSTER_SELECTION_imp
* Author:  sylvain
* Creation date:  Sun Feb 15 2009
*/
IMPLEMENTATION
THRUSTER_SELECTION_imp
REFINES
THRUSTER_SELECTION
bf >lor[Bkeyword]SEES
SAFER_TYPE
IMPORTS
selection.BASIC_TSL
OPERATIONS
/* This function is just a sample of what should be the final function */
fire_thrusters_xx(axis_value) =
BEGIN
/* Reinitialise all thrusters to "notfired" */
selection.RESET_THRUSTERS_STATE_MAP;
IF axis_value ≠ zero THEN
selection.FIRE_THRUSTER_XX(axis_value,1);
selection.FIRE_THRUSTER_XX(axis_value,2);
selection.FIRE_THRUSTER_XX(axis_value,3);
selection.FIRE_THRUSTER_XX(axis_value,4)
END
END;

/*This function is just a sample of what should be the final function */
fire_thrusters_yy(axis_value) =
BEGIN
/* Reinitialise all thrusters to "notfired" */
selection.RESET_THRUSTERS_STATE_MAP;
IF axis_value ≠ zero THEN
selection.FIRE_THRUSTER_YY(axis_value,1);
selection.FIRE_THRUSTER_YY(axis_value,2);
selection.FIRE_THRUSTER_YY(axis_value,3);
selection.FIRE_THRUSTER_YY(axis_value,4)
END
END;

/*This function is just a sample of what should be the final function */
```

```
fire_thrusters_zz(axis_value) =
BEGIN
/* Reinitialise all thrusters to "notfired" */
selection.RESET_THRUSTERS_STATE_MAP;
IF axis_value ≠ zero THEN
selection.FIRE_THRUSTER_ZZ(axis_value,1);
selection.FIRE_THRUSTER_ZZ(axis_value,2);
selection.FIRE_THRUSTER_ZZ(axis_value,3);
selection.FIRE_THRUSTER_ZZ(axis_value,4)
END
END
END
```

C.6 BASIC TRANSLATION

```
/* BASIC_TRANSLATION
* Author: sylvain
* Creation date: Mon Feb 16 2009
*/
MACHINE
BASIC_TRANSLATION
SEES
SAFER_TYPE
VARIABLES
relation
INVARIANT
/* A relation to represent what value has each axis */
relation ∈ TRANS_AXIS ⟶ AXIS_VALUE
INITIALISATION
relation := TRANS_AXIS * {zero}
OPERATIONS

/* Reset all the value to zero */
INIT_RELATION =
BEGIN
relation := TRANS_AXIS * {zero}
END;
/* Modify the value trans with the given value */
SET_AXIS_VALUE(trans,value) =
PRE
trans ∈ TRANS_AXIS  ∧
value ∈ AXIS_VALUE
THEN
relation(trans) := value
```

```
END
END
```

C.7 BASIC TRANSLATION implementation

```
/* BASiC_TRANSLATION_imp
* Author:  sylvain
* Creation date:  Mon Feb 16 2009
*/
IMPLEMENTATION
BASIC_TRANSLATION_imp
REFINES
BASIC_TRANSLATION
SEES
SAFER_TYPE
IMPORTS
transaxis.BASIC_TRANS_AXIS,
tr_axisvalue.BASIC_AXIS_VALUE,
translation.BASIC_ARRAY_VAR((3,5),(0,2)),
basic_translation.BASIC_STRING
INVARIANT
/* A relation to represent what value has each axis */
dom(translation.arr_vrb) = ran(encode_trans_axis)  ∧
dom(encode_trans_axis) = dom(relation)  ∧
ran(translation.arr_vrb) ⊆ ran(encode_axis_value)  ∧
ran(relation) ⊆ dom(encode_axis_value)
INITIALISATION
VAR tmpaxis, tmpvalue IN
tmpvalue ← tr_axisvalue.encode(zero);
tmpaxis ← transaxis.encode(yy);
translation.STR_ARRAY(tmpaxis,tmpvalue);
tmpaxis ← transaxis.encode(zz);
translation.STR_ARRAY(tmpaxis,tmpvalue);
tmpaxis ← transaxis.encode(xx);
translation.STR_ARRAY(tmpaxis,tmpvalue)
END
OPERATIONS
/* Reset all the value to zero */
INIT_RELATION =
VAR tmpaxis, tmpvalue IN
tmpvalue ← tr_axisvalue.encode(zero);

tmpaxis ← transaxis.encode(yy);
translation.STR_ARRAY(tmpaxis,tmpvalue);
```

```
tmpaxis ⟵ transaxis.encode(zz);
translation.STR_ARRAY(tmpaxis,tmpvalue);
tmpaxis ⟵ transaxis.encode(xx);
translation.STR_ARRAY(tmpaxis,tmpvalue)
END;
/* Modify the value trans with the given value */
SET_AXIS_VALUE(trans,value) =
VAR tmpaxis, tmpvalue IN
tmpvalue ⟵ tr_axisvalue.encode(value);
tmpaxis ⟵ transaxis.encode(trans);
translation.STR_ARRAY(tmpaxis,tmpvalue);
IF value ≠ zero THEN
CASE trans OF
EITHER xx THEN
IF value = pos
THEN basic_translation.STRING_WRITE("Translation on XX POSITIVE")
ELSE basic_translation.STRING_WRITE("Translation on XX NEGATIVE")
END
OR yy THEN
IF value = pos
THEN basic_translation.STRING_WRITE("Translation on YY POSITIVE")
ELSE basic_translation.STRING_WRITE("Translation on YY NEGATIVE")
END
OR zz THEN
IF value = pos
THEN basic_translation.STRING_WRITE("Translation on ZZ POSITIVE")
ELSE basic_translation.STRING_WRITE("Translation on ZZ NEGATIVE")
END
END
END
END
END
END
```

C.8 BASIC ROTATION

```
/* BASIC_TRANSLATION
* Author: sylvain
* Creation date: Mon Feb 16 2009
*/
MACHINE
BASIC_ROTATION
SEES
SAFER_TYPE
```

```
VARIABLES
relation
INVARIANT
/* A relation to represent what value has each axis */
relation ∈ ROT_AXIS → AXIS_VALUE
INITIALISATION
relation := ROT_AXIS * {zero}
OPERATIONS

/* Reset all the value to zero */
INIT_RELATION =
BEGIN
relation := ROT_AXIS * {zero}
END;
/* Modify the value trans with the given value */
SET_AXIS_VALUE(rota,value) =
PRE
rota ∈ ROT_AXIS  ∧
value ∈ AXIS_VALUE
THEN
relation(rota) := value
END
END
```

C.9 BASIC ROTATION implementation

```
/* BASIC_ROTATION_imp
* Author:  sylvain
* Creation date:  Tue Feb 17 2009
*/
IMPLEMENTATION
BASIC_ROTATION_imp
REFINES
BASIC_ROTATION
SEES
SAFER_TYPE
IMPORTS
rotaxis.BASIC_ROT_AXIS,
rt_axisvalue.BASIC_AXIS_VALUE,
rotation.BASIC_ARRAY_VAR((6,8),(0,2)),
basic_rotation.BASIC_STRING
INVARIANT
/* A relation to represent what value has each axis */
dom(rotation.arr_vrb) = ran(encode_rot_axis)  ∧
dom(encode_rot_axis) = dom(relation)  ∧
ran(rotation.arr_vrb) ⊆ ran(encode_axis_value)  ∧
ran(relation) ⊆ dom(encode_axis_value)
INITIALISATION
VAR tmpaxis, tmpvalue IN
tmpvalue ⟵ rt_axisvalue.encode(zero);
tmpaxis ⟵ rotaxis.encode(pitch);
rotation.STR_ARRAY(tmpaxis,tmpvalue);
tmpaxis ⟵ rotaxis.encode(yaw);
rotation.STR_ARRAY(tmpaxis,tmpvalue);
tmpaxis ⟵ rotaxis.encode(roll);
rotation.STR_ARRAY(tmpaxis,tmpvalue)
END
OPERATIONS
/* Reset all the value to zero */
INIT_RELATION =
VAR tmpaxis, tmpvalue IN
tmpvalue ⟵ rt_axisvalue.encode(zero);

tmpaxis ⟵ rotaxis.encode(pitch);
rotation.STR_ARRAY(tmpaxis,tmpvalue);
tmpaxis ⟵ rotaxis.encode(yaw);
rotation.STR_ARRAY(tmpaxis,tmpvalue);
tmpaxis ⟵ rotaxis.encode(roll);
rotation.STR_ARRAY(tmpaxis,tmpvalue)
END;
/* Modify the value trans with the given value */
SET_AXIS_VALUE(rota,value) =
VAR tmpaxis, tmpvalue IN
tmpvalue ⟵ rt_axisvalue.encode(value);
tmpaxis ⟵ rotaxis.encode(rota);
rotation.STR_ARRAY(tmpaxis,tmpvalue);
IF value ≠ zero THEN
CASE rota OF
EITHER pitch THEN
IF value = pos
THEN basic_rotation.STRING_WRITE("Rotation on Pitch POSITIVE")
ELSE basic_rotation.STRING_WRITE("Rotation on Pitch NEGATIVE")
END
OR yaw THEN
IF value = pos
THEN basic_rotation.STRING_WRITE("Rotation on Yaw POSITIVE")
ELSE basic_rotation.STRING_WRITE("Rotation on Yaw NEGATIVE")
END
OR roll THEN
IF value = pos
```

```
THEN basic_rotation.STRING_WRITE("Rotation on Roll POSITIVE")
ELSE basic_rotation.STRING_WRITE("Rotation on Roll NEGATIVE")
END
END
END
END
END
END
```

C.10 BASIC TSL

```
/* BASIC_TSL
* Author:  sylvain
* Creation date:  Sun Feb 15 2009
*/
MACHINE
BASIC_TSL
SEES
SAFER_TYPE
VARIABLES
tsl_axis_xx, tsl_axis_yy, tsl_axis_zz, thrusters_state_map
INVARIANT
tsl_axis_xx ∈ AXIS_VALUE ⟶ {∅,{b1,b2,b3,b4},{f1,f2,f3,f4}}
tsl_axis_xx =
{(zero↦∅),
(pos↦{f1,f2,f3,f4}),
(neg↦{b1,b2,b3,b4})}  ∧
tsl_axis_yy ∈ AXIS_VALUE ⟶ {∅,{l1r,l3r,l1f,l3f},{r2r,r4r,r2f,r4f}}  ∧
tsl_axis_yy =
{(zero↦∅),
(pos↦{r2r,r4r,r2f,r4f}),
(neg↦{l1r,l3r,l1f,l3f})}  ∧

tsl_axis_zz ∈ AXIS_VALUE ⟶ {∅,{d2r,d1r,d1f,d2f},{u3r,u4r,u3f,u4f}}  ∧
tsl_axis_zz =
{(zero↦∅),
(pos↦{d2r,d1r,d1f,d2f}),
(neg↦{u3r,u4r,u3f,u4f})}  ∧
thrusters_state_map ∈ THRUSTERS ⟶ THRUSTERS_SWITCH
INITIALISATION
tsl_axis_xx :=
{(zero↦∅),
(pos↦{f1,f2,f3,f4}),
(neg↦{b1,b2,b3,b4})}  ||
tsl_axis_yy :=
{(zero↦∅),
(pos↦{r2r,r4r,r2f,r4f}),
(neg↦{l1r,l3r,l1f,l3f})}  ||
tsl_axis_zz :=
{(zero↦∅),
(pos↦{d2r,d1r,d1f,d2f}),
(neg↦{u3r,u4r,u3f,u4f})}  ||
thrusters_state_map := THRUSTERS * {notfired}
OPERATIONS
RESET_THRUSTERS_STATE_MAP =
BEGIN
skip
END;
FIRE_THRUSTER_XX(axis_value,thruster_number) =
PRE
axis_value ∈ AXIS_VALUE  ∧
axis_value ≠ zero  ∧
thruster_number ∈ (1,4)
THEN
skip
END;
FIRE_THRUSTER_YY(axis_value,thruster_number) =
PRE
axis_value ∈ AXIS_VALUE  ∧
axis_value ≠ zero  ∧
thruster_number ∈ (1,4)
THEN
skip
END;
FIRE_THRUSTER_ZZ(axis_value,thruster_number) =
PRE
axis_value ∈ AXIS_VALUE  ∧
axis_value ≠ zero  ∧
thruster_number ∈ (1,4)
THEN
skip
END
END
```

C.11 BASIC TSL implementation

```
/* BASIC_SET_VAR
* Author:  sylvain
* Creation date:  Sun Feb 15 2009
*/
IMPLEMENTATION
BASIC_TSL_imp
REFINES
BASIC_TSL
SEES
SAFER_TYPE
IMPORTS
axisvalue.BASIC_AXIS_VALUE,
thrusters.BASIC_THRUSTERS,
thrusters_state.BASIC_THRUSTERS_STATE,
selection_xx.BASIC_ARRAY_RGE((1,2),(1,4),(1,24)),
selection_yy.BASIC_ARRAY_RGE((1,2),(1,4),(1,24)),
selection_zz.BASIC_ARRAY_RGE((1,2),(1,4),(1,24))
INVARIANT
/* Selection XX */
ran(selection_xx.arr_rge(1)) ⊆ ran(encode_thrusters)  ∧
ran(selection_xx.arr_rge(2)) ⊆ ran(encode_thrusters)  ∧
tsl_axis_xx(zero) ⊆ dom(encode_thrusters)  ∧
tsl_axis_xx(pos) ⊆ dom(encode_thrusters)  ∧
tsl_axis_xx(neg) ⊆ dom(encode_thrusters)  ∧
ran(encode_axis_value)-{0} = dom(selection_xx.arr_rge)  ∧
dom(tsl_axis_xx) = dom(encode_axis_value)  ∧
/* Selection YY */
ran(selection_yy.arr_rge(1)) ⊆ ran(encode_thrusters)  ∧
ran(selection_yy.arr_rge(2)) ⊆ ran(encode_thrusters)  ∧
tsl_axis_yy(zero) ⊆ dom(encode_thrusters)  ∧
tsl_axis_yy(pos) ⊆ dom(encode_thrusters)  ∧
tsl_axis_yy(neg) ⊆ dom(encode_thrusters)  ∧
ran(encode_axis_value)-{0} = dom(selection_yy.arr_rge)  ∧
dom(tsl_axis_yy) = dom(encode_axis_value)  ∧
/* Selection ZZ */
ran(selection_zz.arr_rge(1)) ⊆ ran(encode_thrusters)  ∧
ran(selection_zz.arr_rge(2)) ⊆ ran(encode_thrusters)  ∧
tsl_axis_zz(zero) ⊆ dom(encode_thrusters)  ∧
tsl_axis_zz(pos) ⊆ dom(encode_thrusters)  ∧
tsl_axis_zz(neg) ⊆ dom(encode_thrusters)  ∧
ran(encode_axis_value)-{0} = dom(selection_zz.arr_rge)  ∧
dom(tsl_axis_zz) = dom(encode_axis_value)  ∧
/* THRUSTERS STATE */
thrusters_state.relation = thrusters_state_map
OPERATIONS
RESET_THRUSTERS_STATE_MAP =
BEGIN
thrusters_state.INIT_RELATION
```

```
END;
FIRE_THRUSTER_XX(axis_value,thruster_number) =
VAR tmpvalue, tmpdecode, tt IN
tmpvalue ⟵ axisvalue.encode_nozero(axis_value);
tt ⟵ selection_xx.VAL_ARR_RGE(tmpvalue,thruster_number);
tmpdecode ⟵ thrusters.decode(tt);
thrusters_state.SET_THRUSTER(tmpdecode,fired)
END;
FIRE_THRUSTER_YY(axis_value,thruster_number) =
VAR tmpvalue, tmpdecode, tt IN
tmpvalue ⟵ axisvalue.encode_nozero(axis_value);
tt ⟵ selection_xx.VAL_ARR_RGE(tmpvalue,thruster_number);
tmpdecode ⟵ thrusters.decode(tt);
thrusters_state.SET_THRUSTER(tmpdecode,fired)
END;
FIRE_THRUSTER_ZZ(axis_value,thruster_number) =
VAR tmpvalue, tmpdecode, tt IN
tmpvalue ⟵ axisvalue.encode_nozero(axis_value);
tt ⟵ selection_xx.VAL_ARR_RGE(tmpvalue,thruster_number);
tmpdecode ⟵ thrusters.decode(tt);
thrusters_state.SET_THRUSTER(tmpdecode,fired)
END
END
```

C.12 AVIONICS

```
/* AVIONICS_imp
* Author: sylvain
* Creation date: Sun Jan 18 2009
*/
MACHINE
AVIONICS
SEES
SAFER_TYPE
OPERATIONS
power_toggle_switch =
BEGIN
skip
END;
mode_toggle_switch =
BEGIN
skip
END;
```

```
move_xx(value) =
PRE
value ∈ AXIS_VALUE
THEN
skip
END;

move_yy(value) =
PRE
value ∈ AXIS_VALUE
THEN
skip
END;
move_zz(value) =
PRE
value ∈ AXIS_VALUE
THEN
skip
END;

move_roll(value) =
PRE
value ∈ AXIS_VALUE
THEN
skip
END;
move_pitch(value) =
PRE
value ∈ AXIS_VALUE
THEN
skip
END;

move_yaw(value) =
PRE
value ∈ AXIS_VALUE
THEN
skip
END
END
```

C.13 AVIONICS implementation

```
/* AVIONICS_imp
* Author:  sylvain
* Creation date:  Sun Jan 18 2009
*/
IMPLEMENTATION
AVIONICS_imp
REFINES
AVIONICS
SEES
SAFER_TYPE
IMPORTS
cmd.COMMAND_INTERPRETER,
selection.THRUSTER_SELECTION
OPERATIONS
power_toggle_switch =
VAR pp IN
pp ⟵ cmd.is_powered_off;
IF pp = TRUE /*If SAFER is off*/
THEN
cmd.switch_on
ELSE
cmd.switch_off
END
END;
mode_toggle_switch =
VAR mm IN
mm ⟵ cmd.is_mode_tran;
IF mm = TRUE /*If SAFER is on mode tran*/
THEN
cmd.switch_mode_rot
ELSE
cmd.switch_mode_tran
END
END;

move_xx(value) =
VAR pp IN
pp ⟵ cmd.is_powered_off;
IF
value ≠ zero  ∧
pp = FALSE /* Machine Power switch ON */
THEN
cmd.translate_on_xx(value)
END;
selection.fire_thrusters_xx(value)
END;
```

```
move_yy(value) =
VAR pp,mm IN
pp ⟵ cmd.is_powered_off;
mm ⟵ cmd.is_mode_tran;
IF
value ≠ zero  ∧
/* Machine Power switch ON and Mode switch TRAN */
pp = FALSE  ∧  mm = TRUE
THEN
cmd.translate_on_yy(value)
END;
selection.fire_thrusters_yy(value)
END;
move_zz(value) =
VAR pp,mm IN
pp ⟵ cmd.is_powered_off;
mm ⟵ cmd.is_mode_tran;
IF
value ≠ zero  ∧
/* Machine Power switch ON and Mode switch TRAN */
pp = FALSE  ∧  mm = TRUE
THEN
cmd.translate_on_zz(value)
END;
selection.fire_thrusters_zz(value)
END;

move_roll(value) =
VAR pp,mm IN
pp ⟵ cmd.is_powered_off;
mm ⟵ cmd.is_mode_tran;
IF
value ≠ zero  ∧
/* Machine Power switch ON and Mode switch ROT */
pp = FALSE  ∧  mm = FALSE
THEN
cmd.rotate_on_roll(value)
END
END;
move_pitch(value) =
VAR pp IN
pp ⟵ cmd.is_powered_off;
IF
value ≠ zero  ∧
/* Machine Power switch ON */
```

```
pp = FALSE
THEN
cmd.rotate_on_pitch(value)
END
END;

move_yaw(value) =
VAR pp,mm IN
pp ⟵ cmd.is_powered_off;
mm ⟵ cmd.is_mode_tran;
IF
value ≠ zero  ∧
/* Machine switch ON and Mode switch ROT */
pp = FALSE  ∧  mm = FALSE
THEN
cmd.rotate_on_yaw(value)
END
END
END
```