EPFL - IC FACULTY
RIGOROUS SYSTEMS DESIGN LAB
OPTIONAL PROJECT REPORT

Definition and Implementation of Validation Strategies for a Nanosatellite Flight Control Software Model

Author:
Alexandre SIKIARIDIS

Supervised by:
Joseph SIFAKIS
Simon BLIUDZE
Anton IVANOV

January 2016

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Contents

1 Introduction 3

2 Validation Strategies 4

3 A Practical Case: Memory Overflow 5
- 3.1 Adapting the Model to nuSMV 5
- 3.2 Verification by Construction 6
  - 3.2.1 Modelling Memory Overflow Handling 6
  - 3.2.2 Generalising the Process 8

4 Reducing Complexity 9
- 4.1 Motivation 9
- 4.2 Design of the Mode Control Model 9
- 4.3 Implementation 12
- 4.4 Handling Errors 15
- 4.5 Properties of the Mode Control Model 16
- 4.6 Relation to the Original Model 18

5 Future Work 20

6 Conclusion 20

1 Introduction

Based on an existing model for the flight control software of cubETH previously developed [1] using the BIP framework [2], the goal of this project was to determine some essential properties the satellite was expected to have, and to review the model so as to verify that those properties were respected.

In order to do so, translations and simplifications of the model were first attempted - those are described in section 3.1. The model was then adjusted such that some properties were exhibited by construction. We describe this in section 3.2.1. Thirdly, an abstraction of the model was developed, which reduces the system's complexity by clearly differentiating modes of behaviour the satellite is expected to adopt; formal verification of properties based on those modes can then be achieved. Explanations regarding design and implementation in BIP are given in section 4.

This project builds upon a model programmed with BIP2 (RC5 version) [3], in an attempt to provide a verifiable flight control system for the SwissCube project [4]. Most important for this project is the structure of the system, which is described in Figure 1.1.

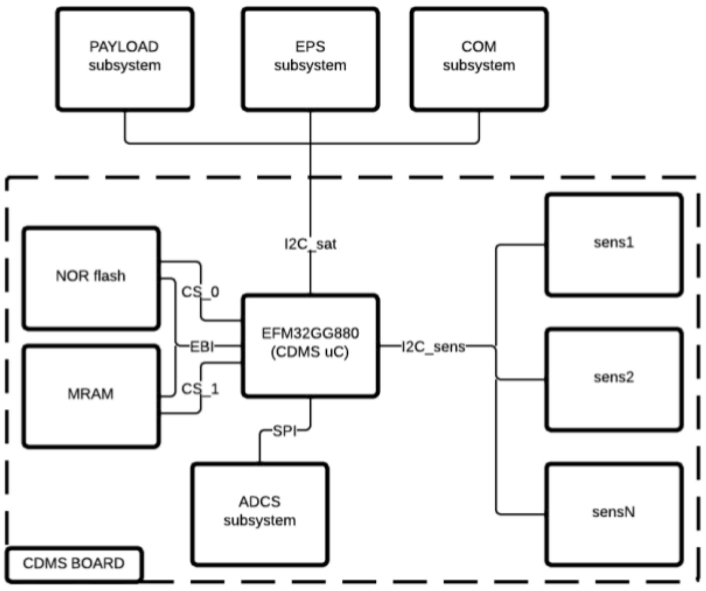


Figure 1.1: Architecture of the flight control system.

2 Validation Strategies

Several validation strategies were considered and explored. The first one described in this report attempted to apply verification tools to the model, as several ones supporting BIP code exist. Those have the advantage of allowing validation of complex properties through computational power.
Properties discussed in section 4 make use in particular of *bipchecker* [5], a tool which looks for execution paths leading to a given combination of component states. One advantage of bipchecker is its support of current BIP syntax, which makes it possible to run tests on BIP code with minimal modifications.
Another tool which we hoped to use is nuSMV, which, given a model, attempts to validate temporal logic rules[6]. While it does not support BIP code, a translator called *bip2smv* has been developed at RiSD. Unfortunately, it only handles old BIP syntax, therefore the SwissCube model had to first be adapted to the old BIP syntax, as is described in section 3.1.

Another validation strategy considered was the modification of the model, such that it would exhibit execution paths that were otherwise left to the implementation, and thus giving the means to verify properties by construction. One example in particular is discussed in section 3.2.1, and the process is then generalised to properties of a similar nature.

Finally, an abstraction of the SwissCube model was developed in BIP. In this new model, all components are simplified, so as to draw focus on interactions between them. This gives us a better view of the way they communicate, and how failures are handled. This is particularly useful given the I2C communication bus used on the satellite, which is considered to be fault-prone.
The abstraction also introduces a new functionality, which enables and disables certain subsystems of the satellite according to modes of operation. This in turn provides a higher-level control on the behaviour of the satellite, and allows us to verify properties based on those modes, related for instance to power consumption and scheduling of operations.

3 A Practical Case: Memory Overflow

In this section, we describe two attempts at at verifying the following property:

> *"When the memory is full, any write operation is interrupted while the error is logged and older data erased. The write operation then resumes."*

This property is crucial to the model, because memory content will often be pictures taken by the satellite which can individually fill up the non-volatile memory. Having cases of memory overflow alert ground control allows for an appropriate reaction. Of course, ensuring the property is respected also means we are certain of the consequences of memory overflows (i.e. older memory will be overwritten; the system will not, for example, overwrite recent data instead, or worse, crash).

In order to verify the property, two different strategies were followed: first, translating the BIP model to nuSMV syntax in the hope of applying the verification tool to it is discussed in section 3.1. The second strategy consisted in modifying the model to exhibit the property by construction; this is discussed in section 3.2

3.1 Adapting the Model to nuSMV

In order to run the BIP model in nuSMV, two steps had to be achieved: first a simplification of the model - which would be too broad in its complete form for any non-trivial property to be verified; and secondly a translation to the old BIP syntax, which is supported by the bip2smv tool.

Simplification

The model was simplified by removing any component unrelated to the memory overflow property described above. The components that were retained are listed here:

- All CDMS components: Flash_mem, I2C_sat, I2C_sens, and Error_log
- Minimal TC components to generate data reads and writes to the memory: CCSDS, tcReceiver, service 13, and service 15_9
- All housekeeping components - kept so as to minimise modification of the behaviour, those could likely be removed as well if necessary.
- Payload and sensor-related components (I2C_sens, sens1 and sens2) to generate heavy data writes to the memory.

Translation

The translation process from new syntax to old syntax is described here point by point:

1. Replace keyword *atom* by *atomic*.
2. Remove empty parentheses ("()") at the end of atom and compound declarations.

3. Replace *package* keyword at beginning of BIP file by *model*.
4. Add declaration of root compound component C at end of file as "*component C start*".
5. Inline all imported files - keyword *use* is not supported.
6. Flatten the model - hierarchical connectors are not supported.
7. Remove all use of C code and includes at beginning of file.

A short shell script was written to automate points 1 to 4. Point 5 can easily be achieved by hand. Points 6 and 7 are more complicated to automate, and are non-trivial to apply without modifying the behaviour of the model. A tool for flattening BIP files does exist - it was not however necessary in this project as the simplified model had a low enough complexity to allow for flattening by hand. Removal of C code however proved much more complicated, and was never fully completed due to lack of time - in particular, validating maintenance of the model's original behaviour is a significant challenge.

Because of this last point, complete translation of the model to old BIP syntax proved extremely time costly and error-prone. Instead, exhibition of the property upon the model itself was therefore undertaken.

3.2 Verification by Construction

Given properties the system is expected to exhibit, having the model present those properties directly allows for a verification by construction, rather than going through tools such as nuSMV or bipchecker, that would require modifications to the model's underlying BIP code such as those discussed previously.

3.2.1 Modelling Memory Overflow Handling

In the SwissCube model, all write operations are handled through the *flash_mem* component. Its interface allows other components to send write requests, which are then handled as per the left-hand side of the FSM described in Fig. 3.1.

Given the writing process modelled here, the point at which writing might have to be interrupted due to a memory overflow is in state *STATUS_WRITE*. Indeed, at that point, the current state of the memory is evaluated, and we can expect the previous *'write();'* statement to have set the status to *FULL* if such was the case. The solution we wish to implement, is to verify the state of the memory, attempt to correct the issue - in this case, to free some memory -, and then to repeat the write operation by returning to the *WRITE_BUFFER* state.

This is first done by modifying the model as is depicted in orange on Fig. 3.1. From *STATUS_WRITE* we add a transition - activated when the memory is detected to be full (encoded as *'status == FULL'*) - which leads to the *FULL* state. From there, three transitions can be followed, activated according to the status variable. An internal transition attempts to free memory, and updates the status variable. One transition named *free_failure* is available if the status variable indicates a permanent failure to free memory, and leads to the original *FAIL* state. Finally, an internal transition *internal_not_full* can be fired once the status variable indicates the memory has been freed, and returns to the *WRITE_BUFFER* state from which the write operation can be resumed.

As will be explained in section 4 - dedicated to the mode control model -, we guarantee the error will be reported by synchronising the *full* transition with the error log on the CDMS. The model then guarantees that the log will eventually be reported back to ground control.

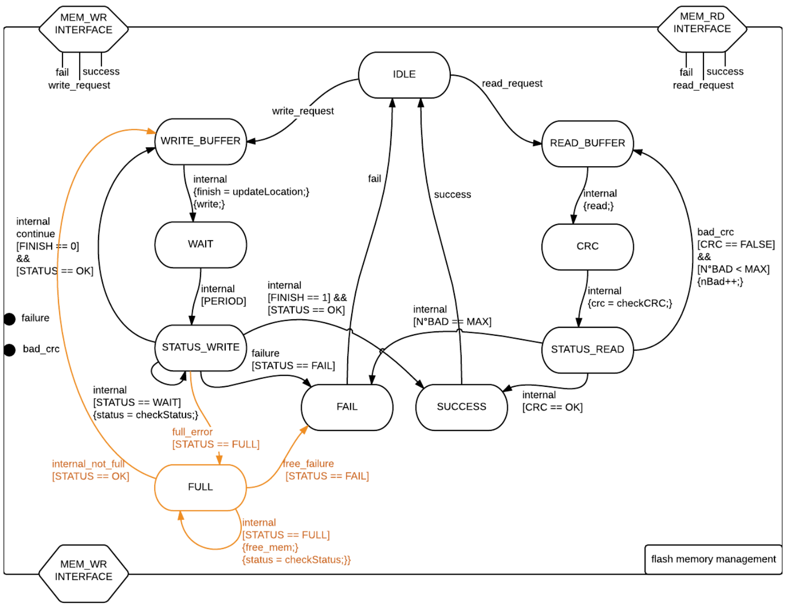

MEM_WR INTERFACE
fail
success
write_request
MEM_RD INTERFACE
fail
success
read_request
IDLE
read_request
write_request
WRITE_BUFFER
READ_BUFFER
internal {finish = updateLocation;} {write;}
fail
success
internal {read;}
WAIT
CRC
internal continue [FINISH == 0] && [STATUS == OK]
bad_crc [CRC == FALSE] && [N*BAD < MAX] {nBad++;}
internal [PERIOD]
internal [FINISH == 1] && [STATUS == OK]
internal [N*BAD == MAX]
internal {crc = checkCRC;}
failure
bad_crc
STATUS_WRITE
failure [STATUS == FAIL]
STATUS_READ
internal [STATUS == WAIT] {status = checkStatus;}
FAIL
SUCCESS
internal [CRC == OK]
full_error [STATUS == FULL]
internal_not_full [STATUS == OK]
free_failure [STATUS == FAIL]
FULL
internal [STATUS == FULL] {free_mem;} {status = checkStatus;}
MEM_WR INTERFACE
flash memory management

Figure 3.1: FSM for the volatile memory component. Modifications to support detection of memory overflow are drawn in orange.

While these modifications ensure memory overflows are detected and logged, an important part of the write process is still left to the implementation. Indeed, consequences of such failures are unclear: returning to the WRITE_BUFFER state does not ensure that the failed write operation will be repeated. If the data had been separated in batches fitting the I2C bus, this might just continue writing the next batch; or even worse, if this was the last batch, the current implementation might lead to unexpected errors.
We can summarise the current handling of a write request in four steps:

1. Receive the write request
2. Separate the data in batches
3. Handle the next batch (or fail). Atomicity is ensured here by the WAIT state.
4. Repeat step 2 or complete the write request.

The issue we have is linked to the fact that the second point is not modelled. Because of that, the *internal_not_full* transition breaks the order of operations, going from step 3 to *something* in between steps 2 and 3.

We fix this by making the batch process explicit as well; the *internal_not_full* transition now goes to a new state, immediately after the current batch has been selected. This is shown on figure 3.2, and ensures that the same batch write will be attempted by staying in step 3.

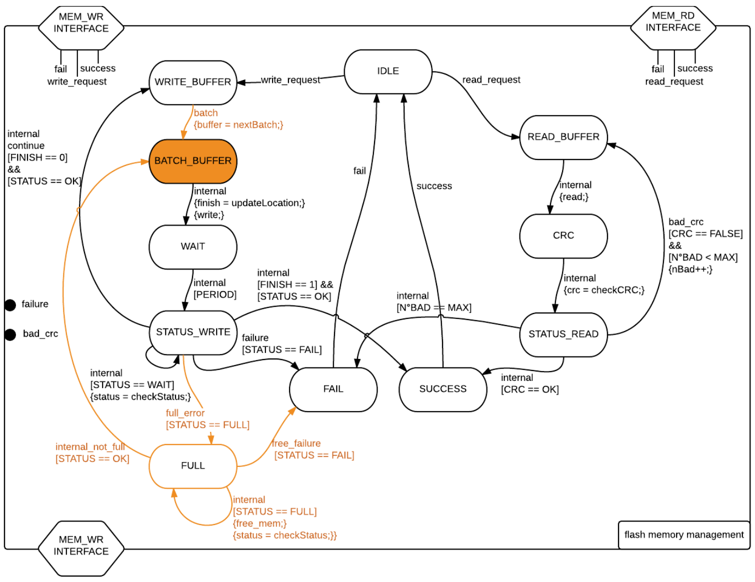


Figure 3.2: Final FSM for the volatile memory component, with modifications ensuring failed writes are repeated after memory overflows are detected.

3.2.2 Generalising the Process

The sequence of operation described in section 3.2.1 is entirely possible in the original model; the *STATUS_WRITE* state has a *failure* transition which should detect memory overflows and handle them. However, modifying the model and making the whole process a separate path in the FSM ensure the property is respected directly upon the model itself, without any regard to the underlying software. In particular, this avoids issues such as unhandled exceptions going unnoticed because the model assumes the software will take care of it.

This process can be generalised to any property relying on values that can be encoded in variables. Indeed, the link between a model and its underlying software can be represented by status variables; if all possible consequences of an action are identified, they can be encoded into a variable, and a test (i.e. a transition) can be dedicated to each possible outcome. This guarantees that none of the identified properties are ignored by the model.
In this precise case, the handled outcomes of the write operation - that is, the values the STATUS variable could take - were WAIT, FAIL, and OK. Separating the new FULL value from the FAIL value means the model now detects memory overflows. Adding the BATCH_BUFFER state then allows us to store the current batch in the *buffer* variable, which allows us to refine the exact way overflow failures are handled by providing more details regarding the *write()* operation in the original model.

4 Reducing Complexity

In order to reduce the system's complexity, an abstract version of the model was designed, which controls each of the main subsystems on the satellite in terms of individual modes of operation. The motivations behind the abstraction are described in this section, as well as the design process and assumptions made while implementing it. Some core properties of the system are then established, before a link back to the original model is made.

4.1 Motivation

In order to provide a high level representation of the behaviour of the satellite, it was desirable to think of the system as operating in one of several modes, namely: ON, SCIENCE, COMMUNICATION, and SAFE. In each mode, all four subsystems - power (EPS), communications (COM), payload operations (PLD) and attitude determination and control (ADCS) - act in a given way.

This results in an autonomous system, upon which we will verify the following properties in section 4.4. Those properties give us assurances regarding power consumption management and operations scheduling.

Property 1. *COM is never operating at the same time as PLD and ADCS.*

Property 2. *I2C_mode_c always eventually returns to one of its base states: ACTIVE and IDLE.*

Property 3. *If an error occurs and triggers a switch to SAFE mode, the system can always eventually come out of it and resume normal operations.*

(Note: refer to section 4.3 for explanations on I2C_mode_c.)

With those properties established on the high-level model, we will outline a mapping to the original mode, ensuring those properties are valid for it too.

4.2 Design of the Mode Control Model

Subsystems Separation

First of all, the components in the original model were separated into subsystems, according to the diagram in Figure 4.1 upon which relies the original design in [1]. In the absence of a formal specification to rely on, this ensures the abstraction is as close as possible to the original mission statement, and provides better assurance that the original model is a refinement of the abstraction.

Essentially, we can distinguish:

1. **The COM subsystem**, which comprises the CCSDS, the tcReceiver, and all services. Additionally, the abstraction needs a way of receiving mode switch orders from ground control - this is a functionality assumed by COM.

2. **The PLD subsystem** which directly maps the Payload component.

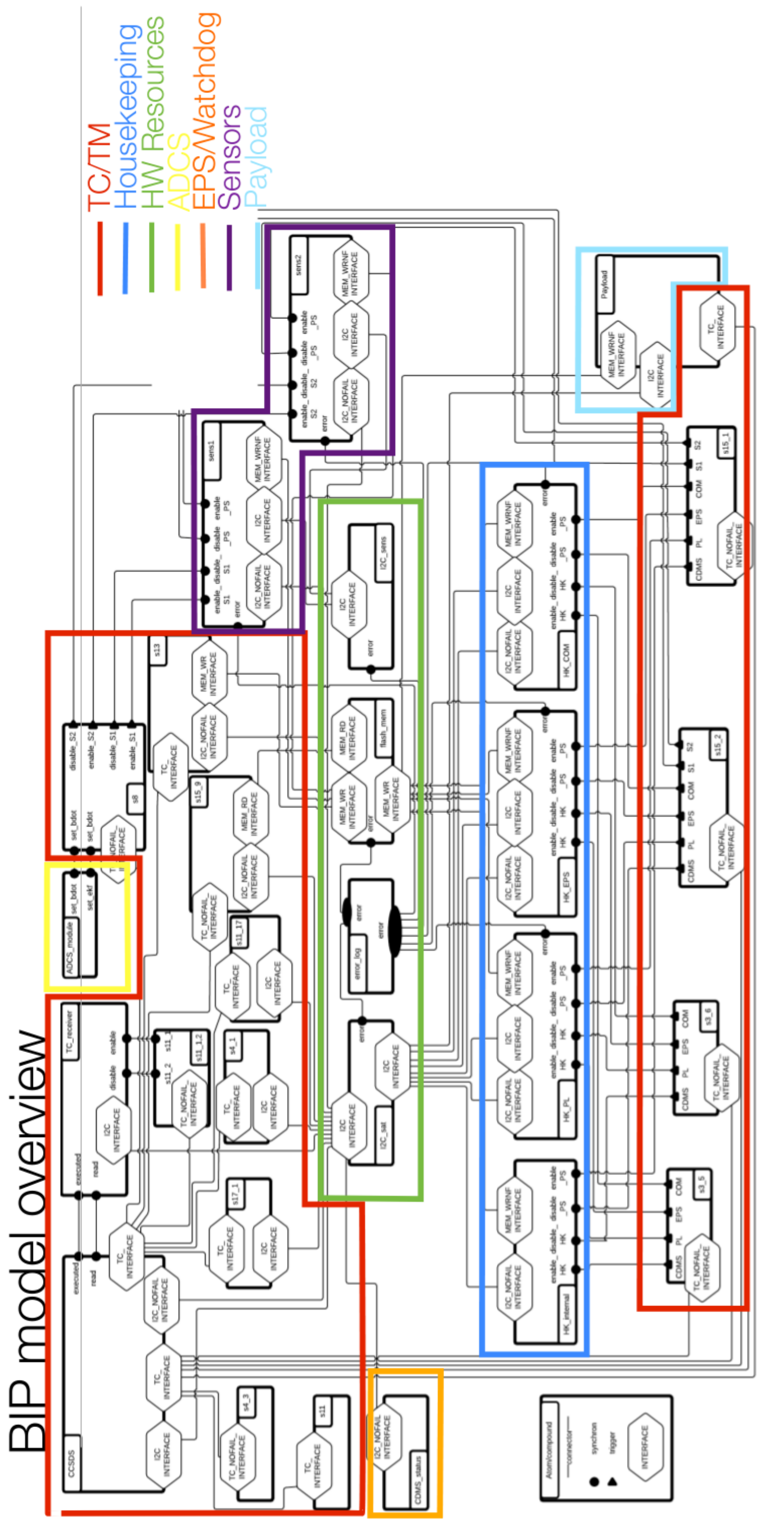


Figure 4.1: A diagram representing the separation of components from the original model into the different subsystems.

3. **The ADCS subsystem** which directly maps the ADCS component.
4. **The EPS subsystem** which interacts with the CDMS watchdog (a heartbeat signal allowing EPS to detect failures of the CDMS subsystem, and perform power cycles if necessary.)
5. **The CDMS subsystem** which is further broken down into *I2C_sat* bus, non-volatile *memory*, *error log*, and *housekeeping* activities. Additionally, an abstraction named *I2C_mode_c* controls the modes of operation of each of the above subsystems, and coordinates transitions with orders received through COM.

Sensors have been omitted from the initial model for simplicity. However, adding them should be straightforward.

Modes of Operation

The various modes of operation are defined as follows:

1. **ON:** All components are activated, but prevented from operating (i.e. firing transitions).
2. **SCIENCE:** Payload and ADCS are operating. Only essential communications are allowed - e.g. mode switches and reset orders.
3. **COMMUNICATION:** All communications are allowed. Payload and ADCS are disabled.
4. **SAFE:** Payload and ADCS are disabled, and only essential communications are allowed.

Additionally, EPS is always active.
In order to achieve that behaviour, subsystems independently switch amongst several main states:

1. **PLD:** Switched off (OFF), activated (ON), operating (OPER)
2. **ADCS:** Switched off (OFF), activated (ON), operating (OPER)
3. **COM:** Limited communications (LISTENING), operating (OPER)
4. **EPS:** Activated (ON)

Table 4.1 combines these information, and describes, for each mode of operation, the corresponding state of each subsystem.

In other words, EPS must always remain active, and COM is always at least alert to vital communications (low power state). PLD and ADCS act in parallel, operating only in Science mode and switching off in Communication mode.

Subsystem	ON	SCIENCE	COMMUNICATION	SAFE
PLD	ON	OPER	OFF	OFF
ADCS	ON	OPER	OFF	OFF
COM	LISTENING	LISTENING	OPER	LISTENING
EPS	ON	ON	ON	ON

Table 4.1: States of subsystems in each mode.

4.3 Implementation

Diagrams of all the components are listed in figure 4.3, and synchronisations between components are represented in figure 4.2.

Some explanations are given here, while more details as to the way the model behaves are given in the next sections.

- In the COM, PLD and ADCS state machines, transitions named *On*, *Saf*, *Sci* and *Com* represent transitions to modes ON, SAFE, SCIENCE and COMMUNICATION respectively.
- Transitions with several names separated by commas represent several transitions in the actual BIP code. This is true for both the individual state machines and the interaction diagram.
- Transitions in I2C_mode_c and HK containing an X in their name represent several transitions, with X $\in \{ON, SAFE, SCI, COM\}$ if not otherwise specified. This is again true for state machines and the interaction diagram.
- *generic* transitions are further explained in section 4.6, where they are referred to as *generic_x_y*. They represent interactions between components in the original model.

I2C Mode Controller

As mentioned in section 4.2, the model uses an additional component named I2C_mode_c, which is in charge of coordinating mode changes of the various subsystems. It has two basic states, from which it handles its different tasks: IDLE when the components are in SAFE mode, and ACTIVE otherwise. From both of these states, a mode switch process can be started. I2C_mode_c will then ask EPS to standby (to avoid conflicts with concurrent resets), while the subsystems are notified to switch modes.

All reset orders from ground control go from COM to I2C_mode_c, which can allow them if it is ACTIVE (through a synchronised transition *reset_order*). Reset orders are disabled when the satellite is in SAFE mode - mode switches to ON replace them.

New Functionality

The mode control abstraction adds new functionality through the handling of the various modes of operation.

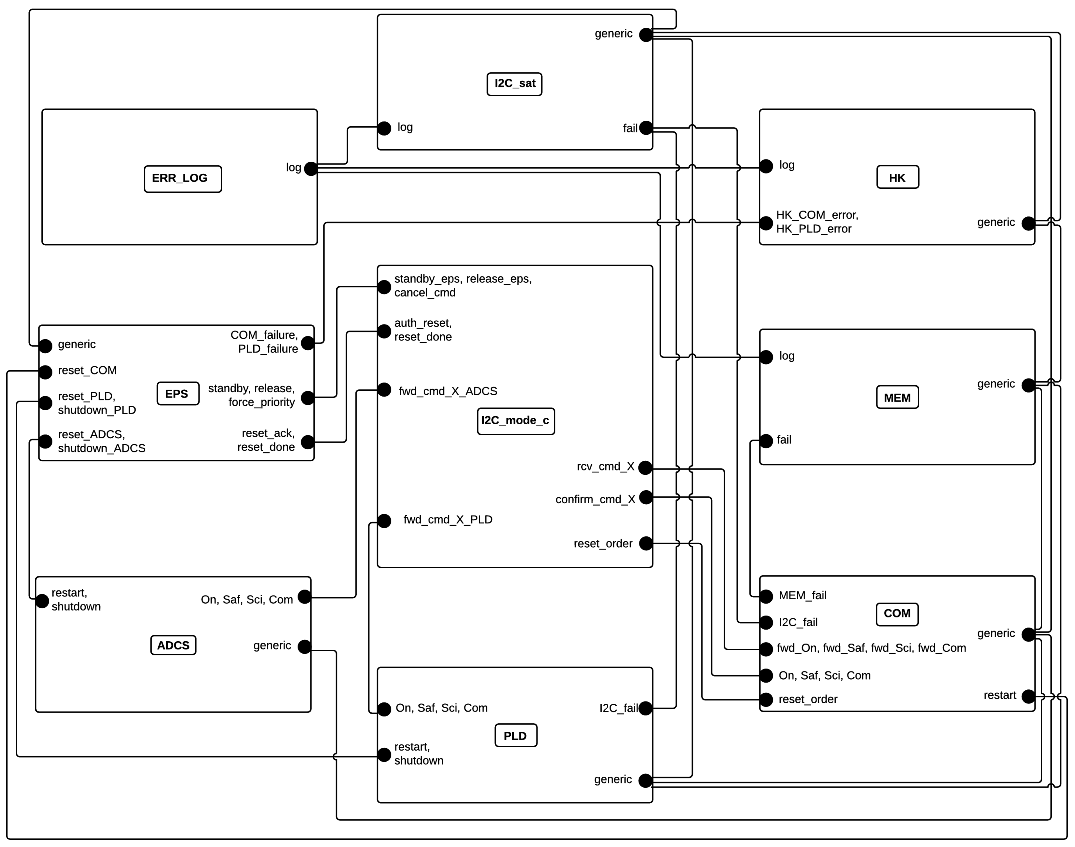
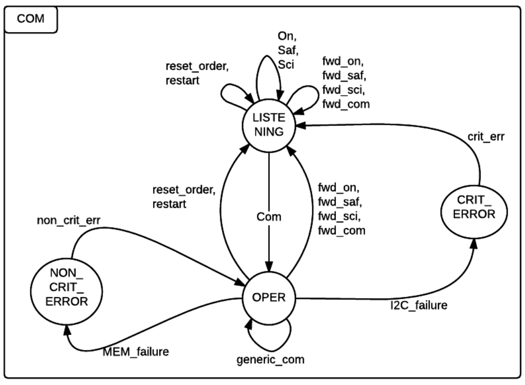


Figure 4.2: Diagram representing BIP synchronisations between the different components.

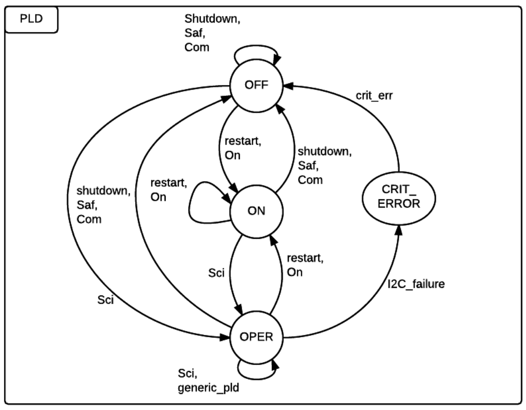
Mode changes occur in four ways - described below - and involve the I2C_mode_c component.

1. **An order to switch modes is received by COM from ground control.** In such a case, COM will forward the order to the I2C_mode_c component on the CDMS, which will transmit the order to PLD and ADCS, before notifying COM back that the order has been applied; COM may then switch its mode as well. Additionally, throughout these events, EPS is put in a standby mode, to avoid overlapping resets.
   This process is shown in purple on figure 4.4.

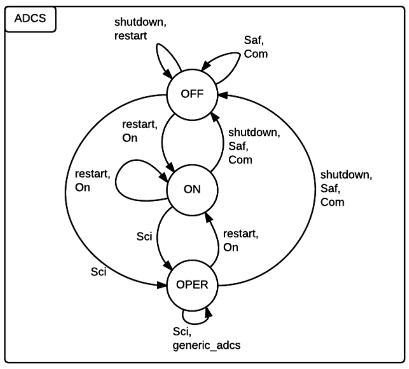
2. **A reset order on PLD and ADCS is received by COM from ground control.** COM will once again forward the order to I2C_mode_c, which will notify EPS of the reset order. EPS will then restart PLD and ADCS, before notifying I2C_mode_c. Additionally, throughout these events, I2C_mode_c is put in a waiting state, making any overlapping mode switch impossible.
   This process is shown in green on figure 4.4.
   Note that a reset order on COM cannot be sent by ground control (how would it be



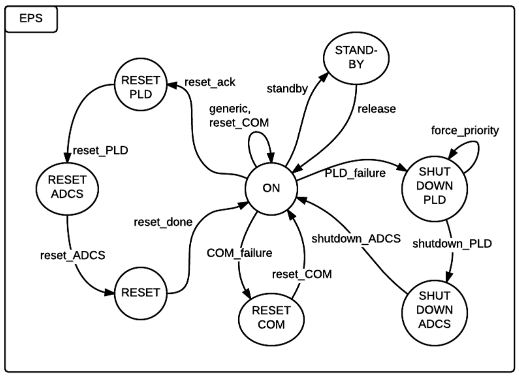
(a) COM



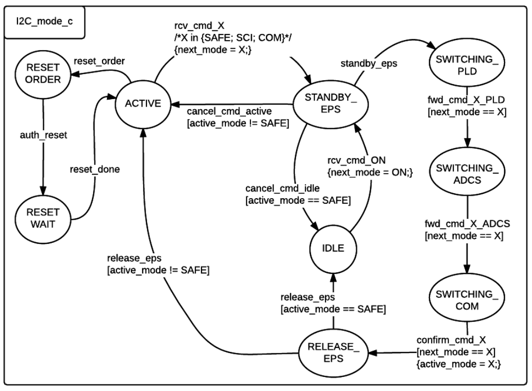
(b) PLD



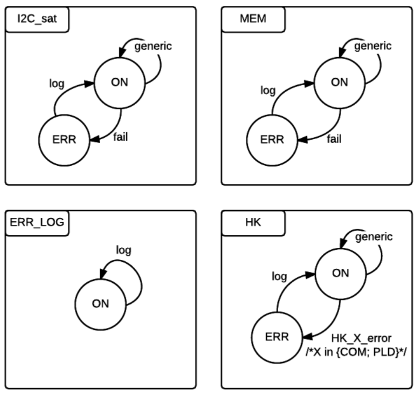
(c) ADCS



(d) EPS


I2C_mode_c
rcv_cmd_X /*X in {SAFE; SCI; COM}*/ {next_mode = X;}
RESET ORDER
reset_order
ACTIVE
standby_eps
SWITCHING_ PLD
STANDBY_ EPS
cancel_cmd_active [active_mode != SAFE]
fwd_cmd_X_PLD [next_mode == X]
auth_reset
reset_done
rcv_cmd_ON {next_mode = ON;}
SWITCHING_ ADCS
cancel_cmd_idle [active_mode == SAFE]
RESET WAIT
IDLE
fwd_cmd_X_ADCS [next_mode == X]
release_eps [active_mode != SAFE]
SWITCHING_ COM
release_eps [active_mode == SAFE]
confirm_cmd_X [next_mode == X] {active_mode = X;}
RELEASE_ EPS

(e) I2C_mode_c



(f) Other CDMS components

Figure 4.3: Model of the Mode Control Abstraction

received?). Instead, EPS autonomously performs a power cycle on COM when errors are detected.

3. **An I2C communication error occurs.** This case is described in detail in section 4.4 under "I2C Communication Error". The general idea is that the system detecting the buggy I2C bus will autonomously move to SAFE mode.

4. **A subsystem failure occurs.** This case is described in detail in section 4.4 under "Non-Responsive Subsystem". The general idea is that the housekeeping component dedicated to the subsystem in question will detect the failure, and ask EPS to shut it down.

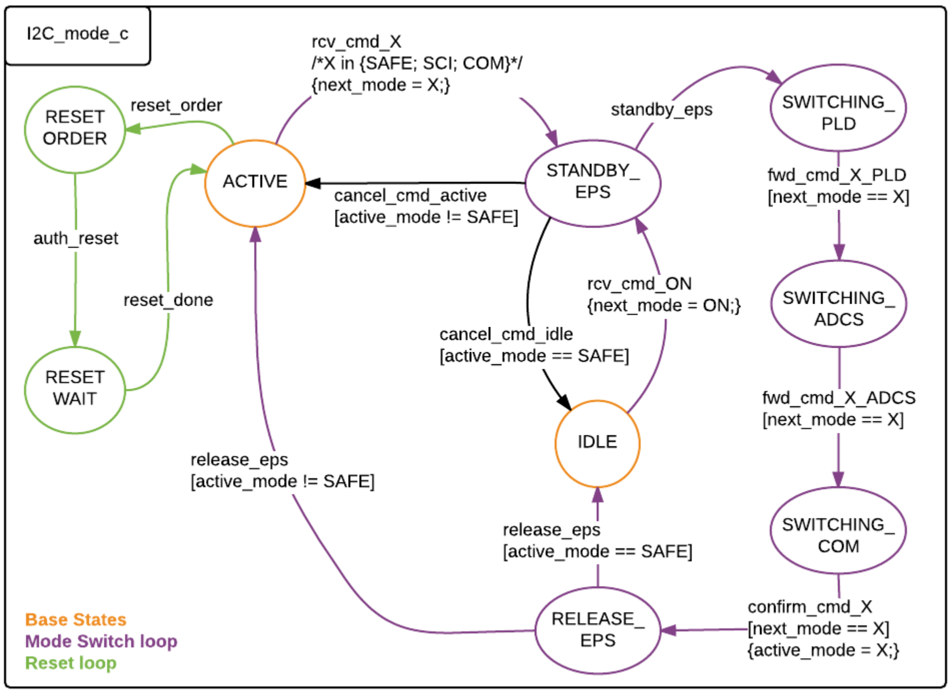

I2C_mode_c
RESET ORDER
reset_order
ACTIVE
rcv_cmd_X /*X in {SAFE; SCI; COM}*/ {next_mode = X;}
standby_eps
SWITCHING_ PLD
STANDBY_ EPS
cancel_cmd_active [active_mode != SAFE]
auth_reset
reset_done
fwd_cmd_X_PLD [next_mode == X]
rcv_cmd_ON {next_mode = ON;}
SWITCHING_ ADCS
cancel_cmd_idle [active_mode == SAFE]
RESET WAIT
fwd_cmd_X_ADCS [next_mode == X]
IDLE
release_eps [active_mode != SAFE]
SWITCHING_ COM
release_eps [active_mode == SAFE]
confirm_cmd_X [next_mode == X] {active_mode = X;}
RELEASE_ EPS
Base States
Mode Switch loop
Reset loop

Figure 4.4: I2C_mode_c component with reset and mode switch loops indicated.

4.4 Handling Errors

We define some expected errors in the table below, and show both how they are detected by the model, and that the system responds correctly to each of them.

Error	Type	Reaction
I2C Communication Error	critical	Involved subsystem goes to SAFE mode.
Non-responsive Subsystem	non-critical	Command subsystem to OFF position.
Memory Overflow	non-critical	Allow information to be overwritten.

I2C Communication Errors

I2C communication errors consist of all communications between a subsystem and the CDMS (therefore going through the I2C_sat bus) that result in a failure. Those are represented in the model by the *X_I2C_sat_failure* transitions (where *X* is the subsystem interacting with the I2C_sat bus), and are synchronised with I2C_sat's *fail* transition - that synchronisation being an event which both sides of the communication realise and react to. I2C_sat will react by transitioning into the *ERROR* state, from which it can only leave by calling the *log* transition itself, which is synchronised with the ERROR_LOG and will report the error.The other subsystem involved will transition into a *CRIT_ERROR* state, from which it will only be able to call an internal transition *crit_err* that transitions it to SAFE mode.

Non-responsive Subsystem

Non-responsive subsystems are detected by their dedicated housekeeping component on the CDMS - i.e. HK_X for subsystem X. In such cases, the housekeeping component will fire a *HK_X_error* transition, synchronised with an *X_failure* transition in EPS (only available when EPS is in *ON* state.) As a result, HK_X will enter its ERROR state, from which it must fire its *log* transition synchronised to the ERROR_LOG to resume operations. In parallel, EPS will enter a state from which it must command subsystem X to the OFF position (via the transition *shutdown_X*), before returning to its ON state.
The subsystem that is put into OFF state can resume operations through a reset. The reset is sent from ground control for PLD and ADCS. In the case of COM, EPS can autonomously restart it.

Memory Overflow

This case is the one discussed in section 3.2.1, for which a custom failure transition was added to the *flash_mem* component in the original model. In the mode control model, that failure transition is summarised within MEM's *fail* transition, which synchronises to the ERROR_LOG component, ensuring memory overflows are always reported.

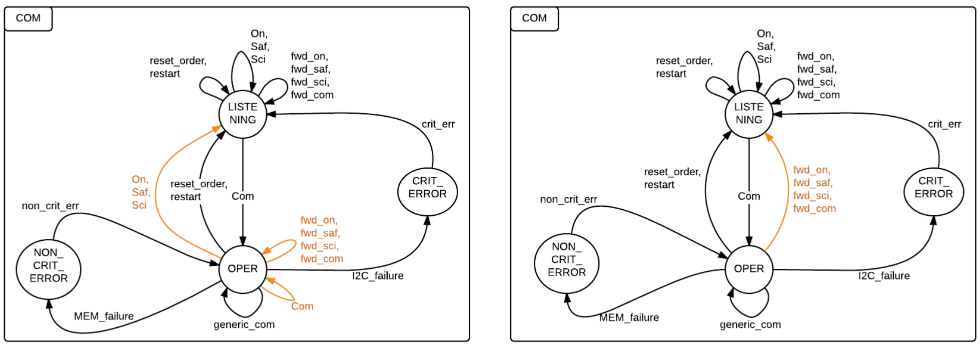
4.5 Properties of the Mode Control Model

As mentioned at the beginning of section 4, we wish to verify the following properties:

> ***Property 1.*** *COM is never operating (i.e. in OPER state) at the same time as PLD and ADCS.*
>
> ***Property 2.*** *I2C_mode_c always eventually returns to one of its base states: ACTIVE and IDLE.*
>
> ***Property 3.*** *If an error occurs and triggers a switch to SAFE mode, the system can always eventually come out of it and resume normal operations.*

Property 1 is essential to ensure power consumption remains under control. In order to prove it, the model was run through bipchecker, a tool which verifies whether certain combinations of states are reachable. In this case, the state COM.OPER cannot be active simultaneously to either of or both PLD.OPER and ADCS.OPER. Running bipchecker on all three combinations proves indeed that the model is correct, with a complete search of the state space yielding no contradictory execution path.

A previous version of the model failed to pass the bipchecker test. In cause was the fact that COM would only transition to its new mode after PLD and ADCS had done so. Therefore, when going from COMMUNICATION mode to SCIENCE mode, PLD and ADCS would enter their OPER mode before COM left its own. The behaviour of COM was thus changed to the current one, where it always transitions to LISTENING mode after having forwarded a mode switch order. This is described by the diagrams in Figure 4.5.



(a) Old model of COM (b) Current model of COM.

Figure 4.5: A comparison of COM and on older version. Solid orange transitions represent modifications.

Property 2 is useful to prove property 3. We demonstrate it by construction: I2C_mode_c is composed of two loops - a reset loop, which can trivially be seen to always loop back to the ACTIVE state - and a mode switch loop, which also loops back to the base states, with no possible loop in between. Hence, if all transitions are assured to terminate, both execution paths will eventually reach back to the base states.
Because all of I2C_mode_c's transitions represent communications over the I2C bus, a non-terminating transition could result from permanent I2C failure, or a synchronisation in the BIP model that never fires.
In the first case, the watchdogs would detect the failure, prompting EPS to reset the system, effectively returning I2C_mode_c to its IDLE state.
On the other hand, the second case occurs only if the other synchronised component (COM, PLD, ADCS or EPS) never reaches the appropriate state, i.e. it is blocked. It is trivial to see that there are no deadlocks in the state machines, so this could only happen if the component is waiting for the I2C bus to be freed. In such a case however, EPS would again not receive the heartbeats, and consequently would trigger a power cycle.
Taking this all into account, property 2 must be true.

Property 3 is crucial to the functionality of the system - without it, the first error triggering the SAFE mode would render the satellite useless. It can be proved true by construction: in SAFE mode, COM is in LISTENING state, while PLD and ADCS are in OFF state. The former means COM can trigger mode switches, if I2C_mode_c is either in ACTIVE or IDLE state. By property 2, we know that must always happen eventually. Therefore, it will always be eventually possible to trigger a mode switch, and thus leave SAFE mode. Property 3 is valid.

4.6 Relation to the Original Model

With properties of the abstract model established, we now want to make the link back to the original model. Indeed, if we can prove that the abstract model can simulate the original model, those properties would be valid in both cases.

The abstraction simulates all subsystem interactions in the original model through two types of connections - successful and failing interactions. Those can only be executed when the involved subsystems are both in their operational modes (or a mode authorised to execute the underlying actions, such as critical services when COM is in LISTENING mode).

Successful interactions between two subsystems x and y are summarised in two synchronised transitions: *generic_x_y* for subsystem X, and *generic_y_x* for subsystem Y. Such interactions can occur between:

1. **COM and I2C_sat, MEM, PLD or ADCS.** Those include the various services under the control of the COM subsystem, that interact with the whole system.
2. **PLD and I2C_sat, MEM or COM.**
3. **ADCS and MEM or COM.**
4. **HK and I2C_sat or MEM.**
5. **EPS and I2C_sat.** This corresponds to heartbeats sent to the EPS which allow it to monitor critical system failures and reset the CDMS.

While communications between subsystems always go through the CDMS, the original model includes some instances of direct communications between COM and PLD, and COM and ADCS. Those are kept for the sake of staying as close as possible to the original model.

Similarly to the successful ones, failing interactions are summarised by transitions *x_y_failure* and *y_x_failure*, which are also synchronised. We have three possible failing interactions:

1. **Between PLD and I2C_sat.** Those are always critical.
2. **Between COM and I2C_sat.** Those are always critical.
3. **Between COM and MEM.** Those are never critical.

The failures involving I2C_sat mirror only failures synchronised to the *fail* transition of the original component. The *error1, error2*, and *error3* transitions are not immediately critical (though they may lead to the *fail* transition), and therefore are grouped under the "successful" transitions instead.

Similarly, failures involving MEM correspond only to failures synchronised with the *fail* transition of the original *flash_mem* component. The *failure* and *bad_crc* transitions are grouped under the successful transitions for the same reasons.

It should be noted that, due to the grouping of all services into the COM subsystem, and given this scheme of modelling interactions, only one service at a time can interact with the I2C_sat. This is coherent with the original model in which "once the request transition is executed, no other user will be able to access the resource until it has returned in its IDLE state; the request transition is not enabled anymore." ([1], p.46)
The consequence of this, however, is that errors resulting from a buggy I2C_sat allowing simultaneous requests would not be detected by this abstraction.

5 Future Work

Future steps related to this project can be taken in different directions. The mode control model now provides an interesting framework to define more properties and validate them. Adapting the original model to support the same mode-based behaviour would likely make a lot of sense.
Modifications to be made to the abstract model would include the support of multiple services (as mentionned in section 4.6), and most importantly of the sensors and the I2C_sens bus.
Referring back to the translation attempts in section 3.1, the development of a translator tool from new to old BIP syntax would be extremely helpful, and allow for simpler verification of complex properties through nuSMV. Applying other verification tools such as bipchecker to the (simplified version of the) original model might also yield interesting results.

6 Conclusion

Different validation strategies were considered at the beginning of the project, with each yielding different results. Application of verification tools proved too complex on the very detailed satellite model. One reason was that software supporting BIP code does not always support the current BIP2 syntax, and the translation process can be made very difficult by some particular issues such as the need to remove C code. Another reason was simply the model's high complexity, though simplifications articulated around specific properties could help solve this.
While still working with the original model, but by altering it in a precise way, we were also able to demonstrate a useful property by construction.
Finally, much of the complexity was abstracted away by designing a new model, in which a high-level control of the satellite was provided through the mode-based behaviour; that allowed us to verify, using the bipchecker tool, more general results based on those modes, and provides a framework to prove additional properties. Those results were then linked back to the original model by establishing a relation between both models.

References

[1] M. Pagnamenta. Rigorous Software Design for Nano and Micro-Satellites Using BIP Framework. 2014.

[2] New BIP tools, . URL `http://www-verimag.imag.fr/New-BIP-tools.html`. Visited in December 2015.

[3] BIP2 (RC5) documentation, April 2015. URL `http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=72ADCD0BEEEED5FA94953FE24CB51007?doi=10.1.1.368.7593&rep=rep1&type=pdf`.

[4] Swisscube project homepage. URL `http://swisscube.epfl.ch/`. Visited in January 2016.

[5] Bipchecker download page, . URL `http://risd.epfl.ch/bipchecker`. Visited in January 2016.

[6] nuSMV homepage. URL `http://nusmv.fbk.eu/`. Visited in January 2016.