Rigorous software design for nano and micro satellites using BIP framework

Master Project

Prepared by:

MARCO PAGNAMENTA

Checked by:

SIMON BLIUDZE
LOUIS MASSON

Approved by:

ANTON IVANOV

●

Space Center EPFL
Lausanne
Switzerland

●

September 14, 2014

●

Record of Revisions

Issue	**Revision**	**Date**	**Modifications**	**Created/modified by**
1	0	14.08.2014	First complete version	Marco Pagnamenta
Issue	**Revision**	**Date**	**Modifications**	**Created/modified by**
1	1	28.08.2014	Corrections	Marco Pagnamenta

Contents

List of Tables

List of Figures

1 Introduction

CubETH is a 1U cubesat class satellite developed in a collaboration between the EPFL, ETHZ and other swiss universities.
The scientific goal of CubETH is to precisely determinate the position and attitude of a satellite in space. To accomplish this, the payload of the mission is composed of several GPS receivers built by u-blox. This payload is placed on a face of the satellite that shall be oriented toward GPS satellites.
CubETH is composed of the following subsystems; EPS (electrical power subsystem), CDMS (command and data management subsystem), COM (telecommunication subsystem), ADCS (attitude determination and control subsystem), PL (payload) and the mechanical structure including the antenna deployment subsystem.
During the development of the previous mission managed by the EPFL; Swisscube, it has been noticed that the software design was not approached in the right way. Standards exists for hardware components and hardware designs and they can be found off the shelf. On the contrary the software has to be adapted to the hardware architecture selected for the satellite. There is not a rigorous and robust way to design software for cubesats yet. Some projects simply structure their code in C/C++ and then extensively test it, maybe using some analysis tools such as "Lint" [5]. This is an helpful tool to find somer design errors, but it does not guarantee that the software behavior is the desired one. Others use SysML to describe the system as a whole [4] and then check some properties such as energy consumption. SysML can be a valid tool for system engineering as a whole, but it is not rigorous enough to allow automatic software behavior verification and validation.
In this project the BIP framework will be used to design the software running in the CDMS of CubETH. The hardware of the CDMS and its software requirements are described in section 2.
The BIP framework is a component-based language which can be used to model and program complex systems, it is introduced in section 3. It has been developed by the Verimag laboratory in Grenoble university and is currently used in the EPFL by the "Rigorous System Design Laboratory" (RISD).
BIP can be used to formally model complex systems and supports a toolset for its verification and validation [9].
The framework is young, therefore there are not many practical designs with it yet. The most resemblant to this implementation is the DALA robot [8] [9], altough this time the software design is made only with BIP. For this reason, the design is not based on existing projects. To structure the model some guidelines, detailed in section 4, have been followed.
The design of the CDMS software model shall be modular so it can be adapted for other satellites developed at the "Swiss Space Center" in the future. In order to accomplish modularity, common patterns in components and structures are identified in section 5. Those can be used as recommendations for future satellite system software design.
The verification of the correctness of the model is made by using a set of common rules in the assembly of atoms and compounds. Those rules are specified in section 6.
All the component designed following those rules and definitions are shown and explained in section 7. Some errors that have been identified enforcing the rules are shown in section 8.

Some advices on the implementation of external functions in the BIP model are presented in section 9. Unfortunately it was not possible to implement the BIP framework on the real BIP hardware because the engine is closed source. Nonetheless section 10 contains informations on how to compile the BIP model for the CDMS hardware.
Finally, the proposed design flow using the BIP framework is proposed in section 11. A final thought on the use of the BIP framework for this kind of application is presented in section 12 along with possible improvements for the framework itself.

2 CubETH CDMS

2.1 Hardware

The CDMS microcontroller is a EFM32GG880 ARM Cortex-M3 core with 1MB of program memory and 128kB of RAM running at 48MHz [1].
It is connected to all the other subsystems (EPS, PL, COM) through an I2C bus (called I2C_SAT) as a master, except for the ADCS, which is connected by SPI.
External non volatile memories are connected to it by an EBI bus. Those are a 128kB MRAM and a 64MB NOR flash devices.
Additional sensors; a magnetometer and a gyroscope, are connected to it on another I2C bus (called I2C_SENS).
All the informations on the CubETH project are available in the ICD document [2].
The hardware is represented in figure 1.

2.2 Software applications requirements

- The CDMS shall provide telecommand execution and telemetry generation according to the selected [2] services defined in the ECSS standards [3] (see figure 2).
- The CDMS shall supervise the correct execution of the software functions on the other subsystems. If a sensor or subsystem is misbehaving, the CDMS can ask to the EPS for a reset of such malfunctioning hardware.
- The CDMS shall be able to save its status in order to resume correct operations following an unexpected reset. Otherwise it shall detect that the variables stored in the non volatile memories are not valid at start up and wait for a telecommand or dump them and start as a "fresh" system. Those variables are the list of telecommands as described in the ECSS standards for service 11 and the list of parameters described in appendix D.
- The CDMS shall manage the data generated from the payload and housekeeping routines in a non volatile memory.
- The CDMS shall run the principal ADCS algorithm and feed results to the ADCS subsystem.
- The CDMS shall periodically contact the EPS subsystem with a "heartbeat" and shall periodically reset both the internal and external watchdogs.

- The CDMS shall respect the timing requirements from TABLE 1.

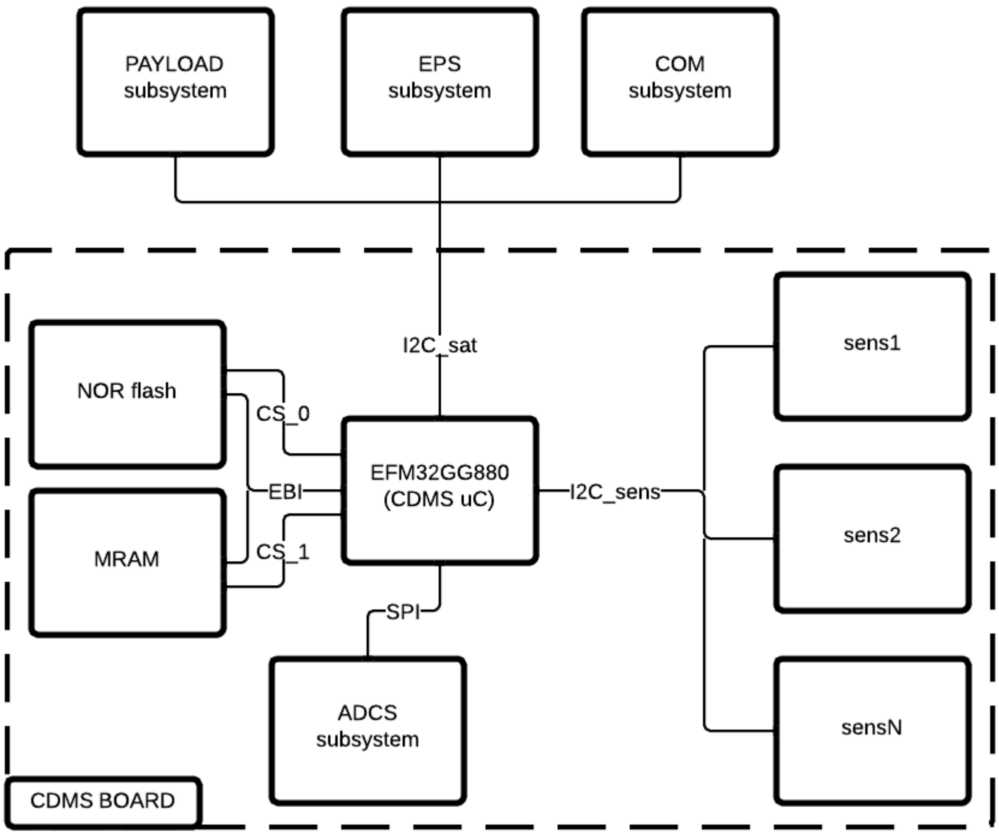
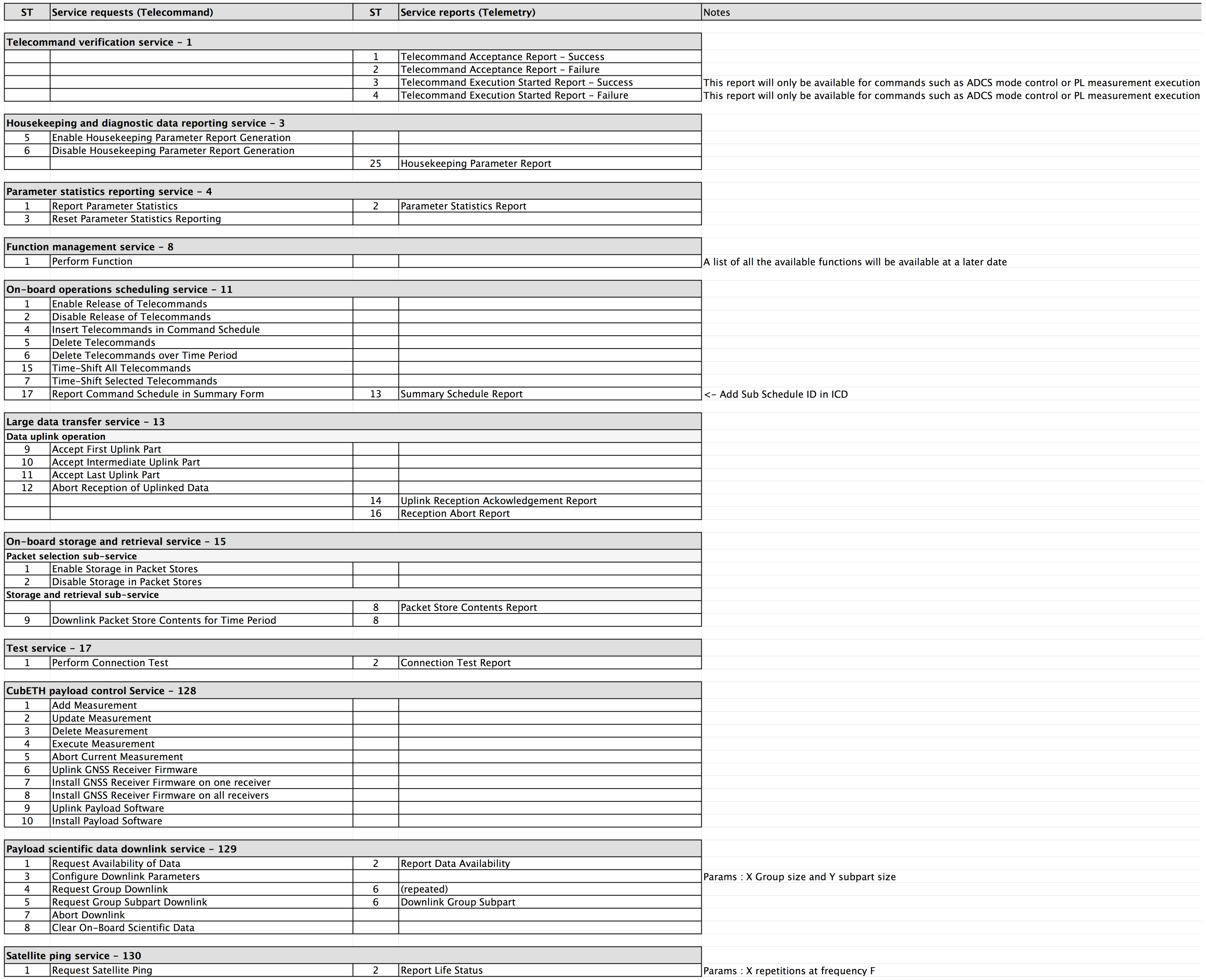


Figure 1: *Simplified CDMS hardware and connections. CS means "Chip select"*

Table 1: CDMS time requirements for task execution

Task	Period
Command scheduling	500 ms
Sensor data handling	500 ms
Scientific data handling	1 s
Housekeeping data handling	1-5 s
ADCS algorithm	3 s



ST	Service requests (Telecommand)	ST	Service reports (Telemetry)	Notes
Telecommand verification service – 1				
		1	Telecommand Acceptance Report – Success	
		2	Telecommand Acceptance Report – Failure	
		3	Telecommand Execution Started Report – Success	This report will only be available for commands such as ADCS mode control or PL measurement execution
		4	Telecommand Execution Started Report – Failure	This report will only be available for commands such as ADCS mode control or PL measurement execution
Housekeeping and diagnostic data reporting service – 3				
5	Enable Housekeeping Parameter Report Generation			
6	Disable Housekeeping Parameter Report Generation			
		25	Housekeeping Parameter Report	
Parameter statistics reporting service – 4				
1	Report Parameter Statistics	2	Parameter Statistics Report	
3	Reset Parameter Statistics Reporting			
Function management service – 8				
1	Perform Function			A list of all the available functions will be available at a later date
On–board operations scheduling service – 11				
1	Enable Release of Telecommands			
2	Disable Release of Telecommands			
4	Insert Telecommands in Command Schedule			
5	Delete Telecommands			
6	Delete Telecommands over Time Period			
15	Time–Shift All Telecommands			
7	Time–Shift Selected Telecommands			
17	Report Command Schedule in Summary Form	13	Summary Schedule Report	<– Add Sub Schedule ID in ICD
Large data transfer service – 13				
Data uplink operation				
9	Accept First Uplink Part			
10	Accept Intermediate Uplink Part			
11	Accept Last Uplink Part			
12	Abort Reception of Uplinked Data			
		14	Uplink Reception Ackowledgement Report	
		16	Reception Abort Report	
On–board storage and retrieval service – 15				
Packet selection sub–service				
1	Enable Storage in Packet Stores			
2	Disable Storage in Packet Stores			
Storage and retrieval sub–service				
		8	Packet Store Contents Report	
9	Downlink Packet Store Contents for Time Period	8		
Test service – 17				
1	Perform Connection Test	2	Connection Test Report	
CubETH payload control Service – 128				
1	Add Measurement			
2	Update Measurement			
3	Delete Measurement			
4	Execute Measurement			
5	Abort Current Measurement			
6	Uplink GNSS Receiver Firmware			
7	Install GNSS Receiver Firmware on one receiver			
8	Install GNSS Receiver Firmware on all receivers			
9	Uplink Payload Software			
10	Install Payload Software			
Payload scientific data downlink service – 129				
1	Request Availability of Data	2	Report Data Availability	
3	Configure Downlink Parameters			Params : X Group size and Y subpart size
4	Request Group Downlink	6	(repeated)	
5	Request Group Subpart Downlink	6	Downlink Group Subpart	
7	Abort Downlink			
8	Clear On–Board Scientific Data			
Satellite ping service – 130				
1	Request Satellite Ping	2	Report Life Status	Params : X repetitions at frequency F

Figure 2: *Selected telecommands as from the CubETH ICD*

3 BIP framework

BIP stands for "Behavior, Interaction, Priority". It is a component-based language used to model and program complex systems [6] [9].
A BIP component is modeled in three layers. A set of atomic components modeled by finite state automata, with a certain behavior described by a set of transitions between the different states of each atomic component. A set of connectors describing the interactions possibles among different atomic components. A set of priorities used to schedule the interactions between those atomic components.
A set of interconnected atoms is called a "compound". Compounds can then be further assembled using the same procedure.
It is possible to verify automatically certain features of the BIP model (e.g. deadlock freedom) using a dedicated tool; the "D-finder". However in this project this is proven enforcing a set of rules (both at atoms and connectors levels), which are defined in section 6.6. That way it is possible to build bottom up a "correct-by-construction" model.
Execution of the BIP model is driven by the BIP Engine, which implements the BIP operational semantics. It is provided as a precompiled library, linked with the generated C++ code of the model.
In this section the main features of BIP are explained. This is not meant as a detailed tutorial, which is available at [7].
One must be cautious when learning BIP because trying to associate its concepts to those of existing programming languages, such as C, can be misleading. For example, the use of a connector is in no way related to a function call. A connector is used to enforce some properties, e.g. ensure that some operations happens only if they are possible at the same time. The same applies to transitions; during a transition it is possible to call external functions, but the transition only represent the state progress.

3.1 Atomic component

An atom is the basic component of BIP. It consists of a set of states, transitions, ports, variables, guards, priorities and external code. An example is given in figure 3. Refer to the guide in figure 4.

3.1.1 State

A **state** (or **place**) is where the atom waits for an interaction to be enabled. If there are no enabled transitions, the atom does nothing. In the case where there is no enabled interaction for every atom in the model, it is a **deadlock**. In figure 3 "WAITING", "PROCESS" and "READY" are states.

3.1.2 Transition

A **transition** represents the progression of the atom from one state to another (or to itself). During this transition, external code can be executed. A transition can be guarded by a boolean condition, as explained in section 3.1.4.
There are two kinds of transitions: they can be internal, which means that they are not visible

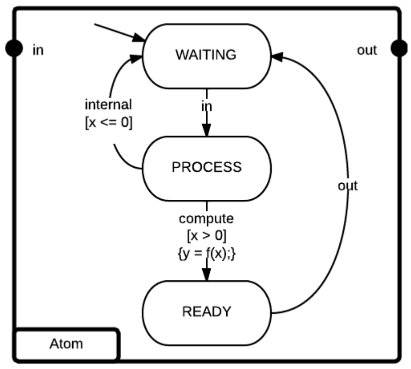


Figure 3: *BIP atom. The starting place is "WAITING". As soon as the "in" port is enabled, the state progress to "PROCESS" following the transition labeled by the "in" port itself. Then, the guards (x>0 or x<=0) for the "internal" and "compute" transitions are evaluated. The place progresses following the transition whose guard evaluates to "true". Assuming that "x" was bigger than 0, then the place is "READY" and an "out" transition may be enabled, leading the atom back to the "WAITING" place*

from the exterior or they can be labeled by a port (section 3.1.3).
Moreover, a transition whose port is exported as described in section 3.1.3 can be **synchronized** with transitions of other components through connectors. Connectors are shown in section 3.2.1.
In figure 3 "in", "internal", "compute" and "out" are transitions.

3.1.3 Port

A **port** is used to to synchronize components. Variables can be associated to ports; that way those variables are visible to other components. Ports label transitions as explained in section 3.1.2.
A port is **enabled** if a transition that it labels is possible, i.e. if the atom is in a state where such a transition is enabled, its guard (if present) is true and it has maximum priority over other possible transitions.
In order to interact with other components of the model, a port can be **exported**. An exported port is accessible by a connector.
In figure 3 "in" and "out" are ports.

3.1.4 Guard

A **guard** is a boolean condition on a component variable that determines if a transition can be executed or not. When the boolean condition evaluates to true, then the transition may be enabled. Guards apply to priorities, too.
In figure 3 the "internal" and "compute" transition have guards.

3.1.5 Priority

In an atom, a **priority** is used to set a specific behavior when more than one transition is possible from the same state.

3.2 Compound

A compound is composed of a set of sub-components (atoms and/or compounds), connectors and priorities. A compound can export ports defined in connectors in order to interact with other components in a larger compound.

3.2.1 Connector

A **connector** is used to synchronize transitions labeled by external ports of different atoms and compounds. Connectors connect ports exported by the sub-components of the compound. Ports must be typed either as **synchrons** or **triggers**.
A connector enforces **strong synchronizations** when all the involved ports are defined as synchrons. That means all the transitions labeled by the ports involved in the connector must be enabled and executed at the same time in order to enable the execution of the connector.
When at least one port is defined as a trigger, the connector authorizes any interaction, involving some of its attached ports, as long as it contains at least one trigger.
Connectors may export a single port. This port can be used to synchronize interactions provided by this connector with interactions provided by other connectors.
In figure 3 connectors can be attached to the two ports "in" and "out".

3.2.2 Priority

In a compound, a **priority** is used to set a specific behavior when more than one interaction is enabled at the same execution step.

4 Guidelines for the BIP model design

The primary goal of this project is to provide a BIP model for the CubETH CDMS. Furthermore, this would allow to validate the applicability of the BIP framework for other projects of the Swiss Space. The design must be modular, in order to allow parts of the model to be used again. The functions of a satellite in the cubesat class remains more or less the same. The biggest differences being in the ADCS and payload subsystems.
This section explains which assumptions have guided the design of the final CDMS model.
In section 4.1 the ways to interact with the hardware are shown; including the microcontroller, external memory, busses between the microcontroller and sensors, actuators and other subsystems.
Section 4.2 shows how the BIP components are designed in order to provide maximum modularity.
In the case that a different BIP engine type is used (BIP engine implementations exist in single or multi-threaded and real-time versions) some changes in the code may be necessary, as explained in section 4.3.
Finally, since the software is designed for a space application, it shall incorporate a proper reaction to hardware failure. Those aspects are shown in section 4.4.

4.1 Variations in hardware components

BIP already provides the tools for hardware abstraction: during transitions, external C/C++ functions can be used. Those can be modified to drive hardware peripherals. In the same way, they can be modified to support a different microcontroller.
The BIP components closer to the hardware may need some adaptation. For example, some buses are single master, other can be multi master. Some memories can overwrite previously written locations, other first need to erase all the content. Changes in the procedure whereby resources are accessed require a change on the behavior of the BIP component.

4.2 Modularity

To ensure modularity, hardware dependent procedures are separated from the applications processes. That means, a hardware resource and its protocol are modeled in BIP in the form of an atom. That way, the rest of the design does not change if a different resource is used. For example, changing the protocol to access the I2C bus will only change the I2C_bus atom itself.
The user-resource concept is used in the BIP modeling:
A **resource** can be a memory region, an hardware component such as a sensor or an external memory, an actuator or a bus. A resource can not be used in parallel by different users.
A **user** is an application process computing algorithms using one or more of the previously listed resources. In principle, all the users can run and progress in parallel, as long as they do not require a resource or the result of another user. Synchronization through connectors is used to enforce those relationships.
Another way to support modularity is to define common structures that can be used to assemble new modules; this is shown in sections 5.2 and 5.3.

4.3 Different BIP engines

The single thread engine allows the execution of only one interaction at a time. The multi thread version, which requires primitives for thread management from an underlying OS, can execute multiple transitions at the same time, as long as they respect the BIP semantics.
The real time version of BIP, which at this time exist only in single thread version, requires an hardware specific function giving a real time clock. This engine version is based on an older BIP syntax and on a real-time dedicated engine. Thus, moderate change in the BIP code is required. Assuming the same syntax and engine version, one shall still add guards on the timing (i.e. time constraints) of the transitions [10] [11].
To avoid possible conflicts, external code operating on a given resource should be used only in one atomic component. This would guarantee that it is never called several times in parallel, leading to thread-safety by construction.

4.4 Dealing with hardware failure

Space is a really harsh environment. The challenges on the hardware are extreme because of radiation and temperature ranges. Therefore hardware failures shall be expected. Some failures can be tolerated, but the software has to be aware of them and react accordingly.

To answer those challenges redundancy is commonly used by the space industry. That is for buses, processors, memories and other electrical components. It is clear that the BIP model should be able to manage redundant hardware and also to react, as far as possible autonomously, to hardware failures.
In order to detect the failures, external functions who verify the performance of the resource shall be provided. Those can be functions actively reading the state of the peripheral or functions to measure time.
Therefore, a failure in an hardware resource should not result in a deadlock, in the limit of feasibility. When an hardware failure is detected, measures to correct it shall be undertaken. In a more advanced fashion, the BIP model should be able to cut away resources which are irretrievably broken. An example is given in appendix F.
Another challenge comes from the fact that the hardware where the BIP model is running could shut down (e.g. following a SEL). In order to save operational time, it is practical if the BIP model could restart from the point when it was shut down. Of course, this requires non volatile memory, which is available on the CDMS. Some thoughts on the issue are illustrated in appendix E.

5 Design of BIP modules for CubETH CDMS

BIP is a component based framework; components can be assembled in order to create more complex systems. It is possible to define classes of components that have the same role, even if they are part of different applications. A top down description of those roles is made in sections 5.2 (high level roles) and 5.3 ("atomic" roles).
Finally, in section 5.4 the way to connect those components is described. In order to make the model easier to check and comprehend, common ways to connect components are defined as "interfaces".
In section 5.1 the conventions used in the graphical model are explained.

5.1 Conventions

The textual BIP model can be directly derived from the graphical BIP model. That means there is only one way to go from the graphical BIP model representation to the BIP code. It could be possible to generate the BIP code directly from its graphical representation. Unfortunately such automatic procedure does not exist yet, so this project uses an ad hoc convention, which is presented in figure 4, and the translation is made by hand.

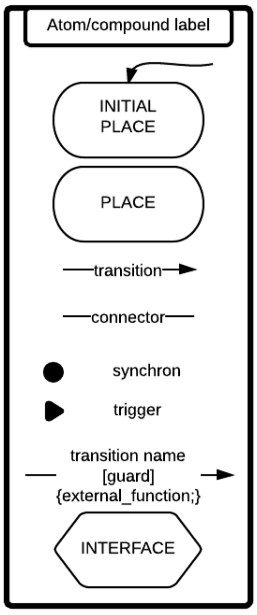


Figure 4: *Symbols used in the graphical BIP model*

5.2 Roles at compound level

The top level of the model represents the whole system, i.e. all the CDMS software applications. It is composed of a set of activities (section 5.2.1), resources (section 5.2.2) and triggered activities (section 5.2.3) connected one to each other with connectors (usually regrouped in interfaces, as in section 5.4). Those are the roles assigned to compounds (and

atoms, if the component is simple enough).
Every compound is built with a set of atoms whose roles are specified in section 5.3.

5.2.1 Activity

The BIP compounds directly satisfying the requirements from section 2.2 are called **activities**. Activities are periodic tasks that the CDMS has to execute. They may use shared resources, in this case they also are users. From the list of components in appendix B, the ones that represent activities are:

- TC_receiver
- CDMS_status
- ADCS_module
- Payload
- HK_internal
- HK_subsystems
- I2C_sensor

5.2.2 Resource

A **resource** is used to model an hardware peripheral that can be used only once at a time. Before starting a new operation on that peripheral, the previous operation shall finish. Another type of resource can be a memory region in the RAM.
In CubETH, the best example for this category is the I2C bus. Its protocol, defined in [2] requires for an I2C communication to be over before starting a new one.
From the list of components in appendix B, the ones that represent resources are:

- I2C_sat
- I2C_sens
- flash_memory
- error_log

5.2.3 Triggered activities

A **triggered activity** is an activity that is not executed periodically, but only in response to a specific event in the environment. All the components treating telecommands belongs to this category, that is the ECSS services.

5.3 Roles at atom level

At the atom level, i.e. the atomic components forming the compounds, other macro types can be distinguished. Those are the **software application** (5.3.1), the **switch** (5.3.2), the **status** (5.3.3), the **mode** (5.3.4) and the **mixed status/mode** (5.3.5).
If a compound is simple enough, it may be composed of a single atom. In that case, it will be either a software application or a switch. This is the case for most of the services and resources.

5.3.1 Software application

A **software application** is an atom where the externals C or C++ functions are executed. Theoretically BIP can support more languages, but here C and C++ are used. The code is used to compute algorithms or to access specific hardware peripherals.
A software application represents a single sequence of execution; compounds with software applications can run in parallel to each other, as long as they do not have to wait for an interaction between themselves or resources.
A generic example is given in figure 5. The atom starts in WAIT state, here it waits for an event to start its execution. In the case of an activity (like here) it waits for a period to elapse. In the case of a triggered activity, it would wait for a "start" strong synchronization coming from another activity. The execution continues and functions are performed during transitions. Depending on synchronizations (here "case1" and "case2") the software application can perform differently (i.e. execution of function2a instead of function2b and vice versa). A final transition leads to the starting place. In the case of a triggered activity, this transition strongly synchronizes back the triggered activity with the activity.
Some examples of software applications from the list of appendix B are:

- I2C_sat
- TC_fetch
- CCSDS_state_machine
- HK_process
- ADCS_routine
- CDMS_status

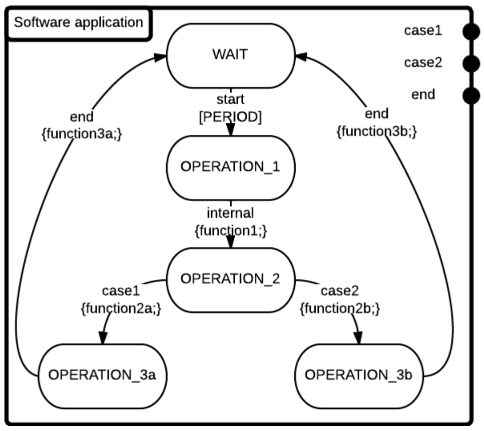


Figure 5: *Example of a "software application" atom*

5.3.2 Switch

A **switch** is a simple atom. A first transition synchronize with the switch which then synchronize with another atom depending on a (or several) specific variable. Usually that variable depends on the environment.
A generic example is given in figure 6. The atom starts in IDLE, here a function (getVal;) getting a specific value is executed. Then, from START to END, a specific transition depending on that variable triggers some other atom (with an actual trigger or with a synchrony typed port). In the end, a "finish" transition offers the possibility to synchronize back (but it is not always necessary).
Some examples of switches from the list of appendix B are:

- s3_5, s3_6, s15_1, s15_2
- apid_distr
- service_distr

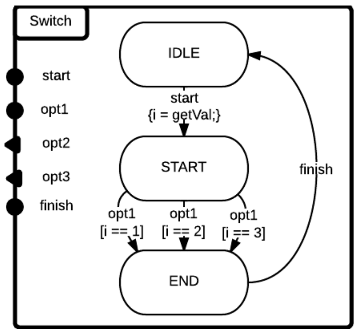


Figure 6: *Example of a "switch" atom*

5.3.3 Status

A **status** is an atom tracking the state of one or several software applications. That way, the status is not mixed with the software application itself and it is easier to model the behavior of the whole compound. A specific software application for every state can then be realized. Moreover if another compound has to synchronize with this one, it can simply interact with the atom keeping track of the status and not with the one performing the application. This leads to a cleaner model.
A generic example is given in figure 7. The atom starts in STATUS_1, and may progress observing some specific transitions ("observationN") of some software applications it is monitoring. Possibly, when reaching a specific place (here STATUS_4) it may enable the action of another software application (usually, a triggered activity) through the "action_start" and "action_end" strong synchronizations.
Some examples of status atoms from the list of appendix B are:

- status (in the various Housekeeping modules)
- TC_buffer

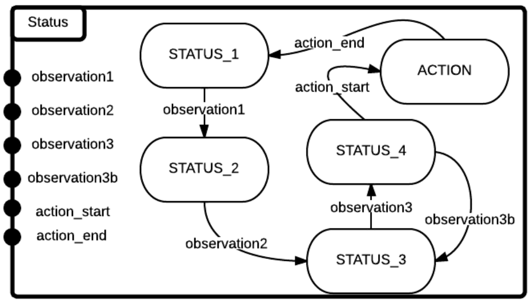


Figure 7: *Example of a "status" atom*

5.3.4 Mode

The **mode** atom changes the behavior of the software application. The main difference with the status atom is that instead of changing according to the software application itself, it is changed by the environment. In other words, the status is changed by a software application belonging to the same compound, the mode instead is changed by others and the software application has to adapt to that change.
A generic example is given in figure 8. An "event1" or "event2", coming from the environment, change the place of this atom. The software application connected to this "mode atom" has then an inhibited transition ("action2" if the place is MODE_1) and an enabled one ("action2").
Typical cases, as from the list in appendix B are:

- HK_switch
- Packet_store_switch

- TC_list_switch
- ADCS_mode

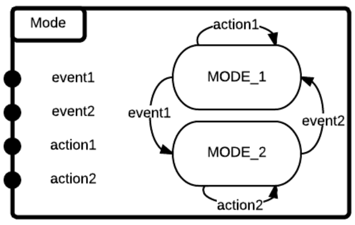


Figure 8: *Example of a "mode" atom*

5.3.5 Mixed mode/status

The **mixed mode/status** atom combines the particularities of the "mode" and "status" types. This kind of atom is connected to various "software application" atoms. They are dependent on the place of the "mixed" type atom but they may also change it.
A generic example is given in figure 9. The various "software applications" connected to this atom can influence it by changing its place (when the "observationN" transition are synchronized), but may also be inhibited. E.g. if the place is STATUS_2, then only the "action2" transition may be enabled.
A practical example is the payload compound (section 7.3).

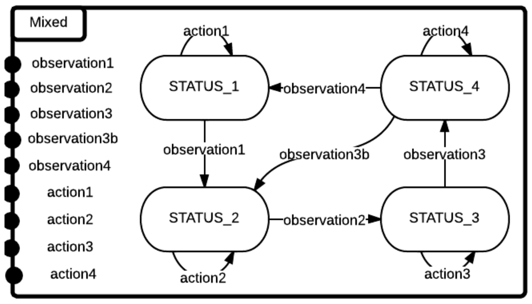


Figure 9: *Example of a "mixed" atom*

5.4 Connectors and interfaces

An **interface** is a set of ports and connectors between two compounds which is repeated many times. A graphical representation of those interfaces is given is figure 10. Those are:

I2C Every time an user uses the I2C resource the pattern shown in section 6.2 must be used. The user must synchronize with I2C atom request port. It is then in a place where it wait either for a "I2C_res" or "I2C_fail transitions". The user "I2C_ask", "I2C_res" and "I2C_fail" ports are connected by one-to-one connectors enforcing strong synchronization respectively to the "request", "return" and "fail" ports.

Memory Every time an user uses the memory resource the pattern shown in section 6.2 must be used. The user must synchronize with write_request or read_request memory atom port. It is then in a place where it wait either for a "mem_res" or "mem_fail transitions". The user "mem_ask", "mem_res" and "mem_fail" ports are connected by one-to-one connectors enforcing strong synchronization respectively to the "write/read request", "return" and "fail" ports.

TC success/failure Every module/compound processing a telecommand must have an unique port labeling a transition from one or several places to start the software application. This port is strongly synchronized with the CCSDS compound. The process shall terminate either with a success or fail synchronization back to the CCSDS compound in a place where the "start" transition may be enabled again.

Others Other interfaces are not spread across the rest of the system, they are unique to their compound. Those are explained case by case in section 7.

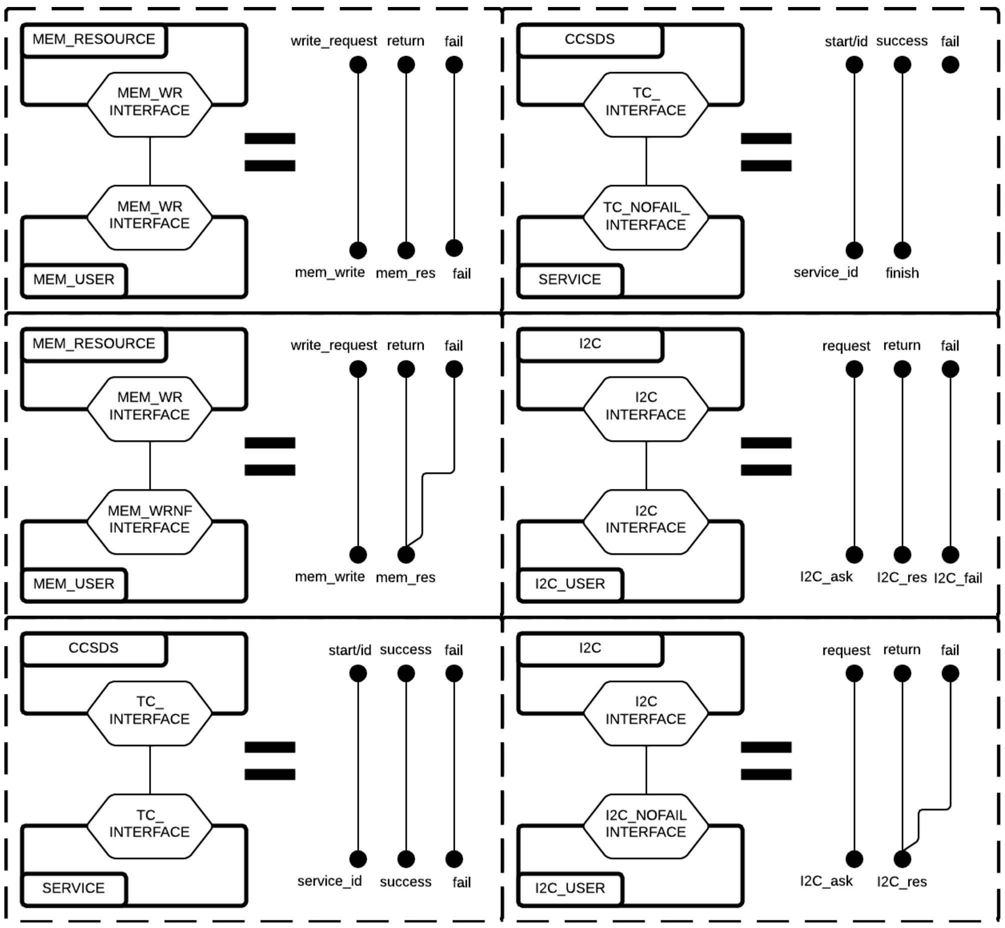


Figure 10: *Interfaces used in the BIP model*

6 Rules used for a "correct by construction" BIP model

The correctness of the model can be assured by verifying first the correct behavior of every atom, then the correct connections between atoms and compounds.
In this section, common features used to determine the correctness of the model are explained. To simplify the verification procedure those common features are used in every atom and connector of the model. That means if every atom and every connection between atoms (or compounds) respect those rules, then the model is correct.
In order to better understand the rules, the structure of the model is recalled in section 6.1.
Finally, a summary of the rule set is given in section 6.6.

6.1 Model structure

Activities (section 5.2.1) are at the **upper level** of the model. Those are periodic activities that shall always be executed. For the model to be correct, none of those activity shall ever come to a local deadlock. This is completely true only for the **TC_receiver** and **CDMS_status** activities. The others can be stopped, but only in a particular state and only when it is a desired behavior. Those are highlighted case by case in section 7.
Activities can use resources (section 5.2.2). In that case, they become **users** of that resource. They also can trigger some "triggered activities"(section 5.2.3). Triggered activities can trigger more triggered activities themselves, or use resources.

6.2 User access to a hardware resource

Each user must acquire the resource before using it and release it after the use. In the BIP model, this property is enforced structurally to avoid simultaneous resource access. This approach may generate deadlocks, they may be avoided implementing the solutions proposed in section 6.4. Moreover it is possible to set priorities to decide which one should go first, in the case where many users request the resource at the same engine step. Otherwise, one will be randomly selected.
The structure enforcing the correct resource usage is illustrated by a generic example as shown in figure 11. An "user type" atom can implement this pattern as many times as it is needed.
Refer to section 5.1 (figure 4) for the conventions used in the graphical BIP model.

The initial place is called WAIT_FOR_RESOURCE. A transition labeled by an exported port; **resource_ask**, leads to the next place. This external port is connected to the resource external port **request** with a connector enforcing strong synchronization. As soon as the resource is in a state where the request transition is enabled, the user has access to the resource and can perform the **resource_ask** transition.
During this transition, the user sets the variables that the resource uses to perform its function (this concept is developed in section 9.1 and is not strictly necessary here) with the *setParameters;* external C function.
Once the user is in the RESOURCE state it waits for a **resource_res** or **failure** interactions. Those are, again, strong synchronized with the transition happening in the "resource" atom respectively through the **return** and **failure** ports.

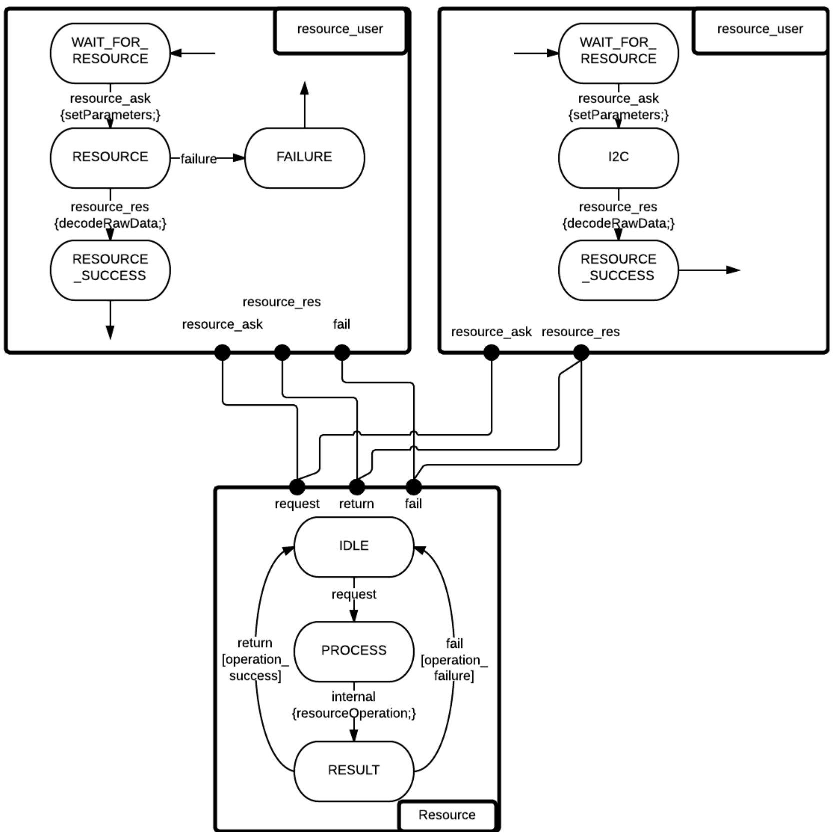


Figure 11: *BIP model of the relationship between a generic hardware resource and two kinds of generic users. One user has a different behavior if the hardware peripheral fails, the other one does not care about the success or failure of the operation*

If a **failure** transition occurs, then the user goes to the FAILURE state, if **resource_res** is issued it goes to RESOURCE_SUCCESS place and uses the data produced by the RESOURCE with the *decodeRawData;* function.
If the user does not care for its following operations about the fate of the resource operation, then the failure transition is eliminated. Both return and fail transitions of the resource atom are in this case strong synchronized to the resource_res transition, still using two separate connectors.
On the "resource" atom side, it has to be noted that the internal procedure between the "request" and "return/fail" transitions may be as complex as necessary.

6.3 User access to shared RAM region

It is possible that many users necessitate writing or reading access to the same variable or array in the RAM. In order to avoid mismatches, this type of resource can be represented by a token as in figure 12. That means, each transition manipulating said RAM region should "take it" (be strong synchronized with the token transitions) to avoid simultaneous access.

This atom has an associated C library implementing the array read and write functions. The array will be directly visible only in that atom.
It is possible to use a token in a different way, as in figure 13.

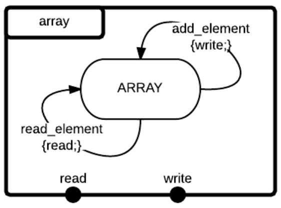


Figure 12: *BIP model of an atom representing a shared memory region. The memory is accessible only during the transition of the resource atom*

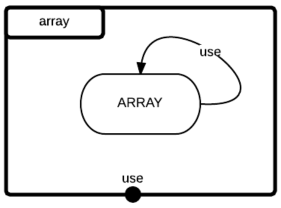


Figure 13: *BIP model of an atom representing a shared memory region. One must take care in using the token every time that a transition uses this memory region*

This approach has advantages and drawbacks: it is more direct to use, because the function using the memory can be directly called in the user transition, without adding more steps and without passing variables through the connectors. The array is in fact declared as a global variable.
Besides, one has to pay attention and manually add a strong synchronization with the token every time that this region of memory is used.
In other words, in the first case (figure 12), forgetting the synchronization means that the variable is not accessible. In the other (figure 13) the memory is still accessible, but there is no more control on its modification; that means an atom could be modifying the array while another is reading it. Worse is that the BIP model would be unaware of that.
In this design, the **second option is chosen**. It is simpler from a BIP point of view (no variables passing through connectors). This choice can be justified by the fact that one should know which shared variables exists and who uses them, so it can directly synchronize all those transitions with the atom of figure 13.

6.4 Simultaneous use of several resources

If more than one resource has to be used at the same time, for example during a DMA transfer involving an external memory and the I2C bus, then the two resources needs to be strongly synchronized with the user at the same time. This is accomplished using a connector connected to the transition implementing this kind of application, the I2C resource and the memory resource. This case is not present in this model, therefore it is not developed. An example is given in figure 14 in case of future developments.
Note the triggers on the fail ports of the resources. The implication is that for the "success" connector to be enabled both resources must have performed correctly; on the other hand, one fail transition is enough to "kill" the process.
The resources are acquired at the same time to avoid deadlocks (i.e. a first user acquire a

resource, a second user acquire another resource and both users, to continue, necessitate the resource that the other has already reserved).

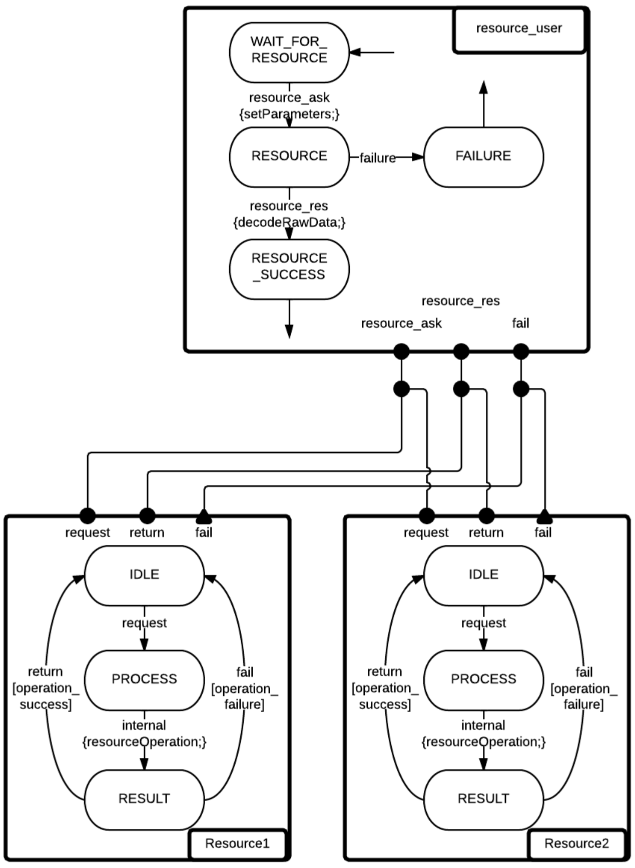


Figure 14: *BIP model of an user necessitating two resources at the same time*

6.5 Synchronization with triggered activities

A triggered activity shall have a synchronization at the start and at the end of its execution sequence, without any possible local deadlock in between.
The difference with resources is that those represents hardware peripherals, which are well defined and do not require additional synchronization steps other that the ones with the user. On the other hand, triggered activities depends on software specification and may interact with various components at the same time.

6.6 Rules imposed on the BIP model

For the BIP model to be correct, all the rules resumed in this section must be respected.

- **Rule 1** An activity (from section 5.2.1) **shall not** encounter a local deadlock. If specified in the requirements the activity can be stopped, but in a localized place and the action must be reversible.
- **Rule 2** A triggered activity (from section 5.2.3) **shall** always synchronize twice with the triggering activity (simple or triggered). At the beginning of its sequence of execution and at the end. A triggered activity **shall not** cause a local deadlock between those two transitions. The first and last places of the sequence of execution do not have to be always the same, but they **shall not** progress until the next strong synchronization.
- **Rule 3** A resource (from section 5.2.2) **shall** always synchronize twice with the corresponding user activity. At the beginning of its sequence of execution and at the end. A resource **shall not** cause a local deadlock. In particular, a resource **shall** expect a hardware failure. The latter **shall not** cause a resource deadlock and its appearance **shall** be available to the rest of the system.
- **Rule 4** A transition labeled by the port strongly synchronized with the beginning of an execution sequence of a triggered activity or resource **shall not** be available anymore before the transition labeled with the port strongly synchronized with the end of the execution sequence.
  Similarly a transition labeled by the port strongly synchronized with the end of an execution sequence of a triggered activity or resource **shall not** be available anymore before the transition labeled with the port strongly synchronized with the beginning of the execution sequence
- **Rule 5** The interfaces used to connect compounds (and atoms) described in section 5.4 **shall** be respected.
- **Rule 6** There **shall not** be cycles in the triggering graph.

The first three rules ensure that a compound can not generate a local deadlock. The combination of the other three rules ensure that if the ports are synchronized with the right connectors, then the fact of "constraining" the compounds with synchronizations can not cause a deadlock.
Consequence If every component satisfies those rules, then the system is deadlock (local and global) free.
Additional assumptions and observations are:

- **Assumption 1** Every C function is assumed correct.
- **Assumption 2** If an external event derails the execution of the BIP engine (e.g. RAM corruption), then the microcontroller is resetted by a watchdog (external or internal).
- **Observation** The satisfaction of all the rules does not exclude temporal deadlocks (in real-time BIP). A temporal deadlock happens if an important deadline is missed [10] [11].

7 BIP modules description and verification

In this section, every BIP compound is described. First, a generic explanation of its role and procedure is given, then, using the rules from section 6 (resumed in section 6.6) its correctness is proved. The global triggering graph is shown in figure 16. Since this graph is a tree, i.e. there are no cycles, rule 6 is respected. The correctness of the compounds and of how they are interconnected implies that the system is deadlock-free.
The convention used in the graphical model is presented in section 5.1.
A generic view on the system, restricted to the "interaction" layer of BIP (ports and connectors), is shown in figure 17 (a more complete vision on the system is available separately in the delivered CD).
Examples of design errors are presented in section 8.

7.1 CDMS_status

The **CDMS_status** component is an activity. It is composed of a single "software application".

7.1.1 Description

This component is in charge of the reset of both the internal and external watchdogs. Moreover it sends a periodic message to the EPS subsystem called "heartbeat". It is an additional security system; if the EPS can not hear the CDMS for some time, it performs a power cycle on it. In figure 15 there is its graphical representation.

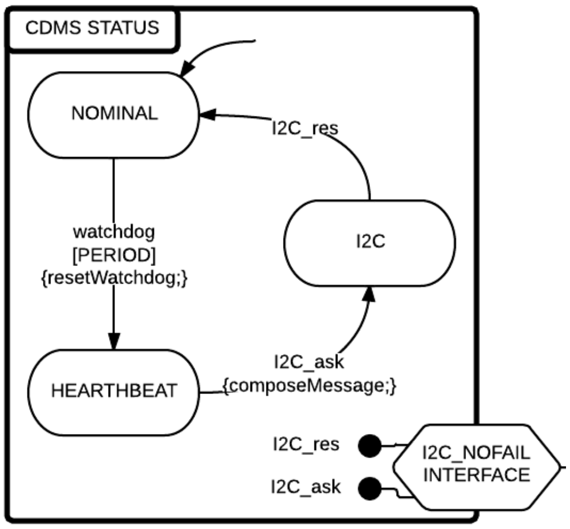


Figure 15: *CDMS status atom*

7.1.2 Discussion on correctness

The I2C interface is correctly implemented and there is no possible local deadlock; this atom is correct.

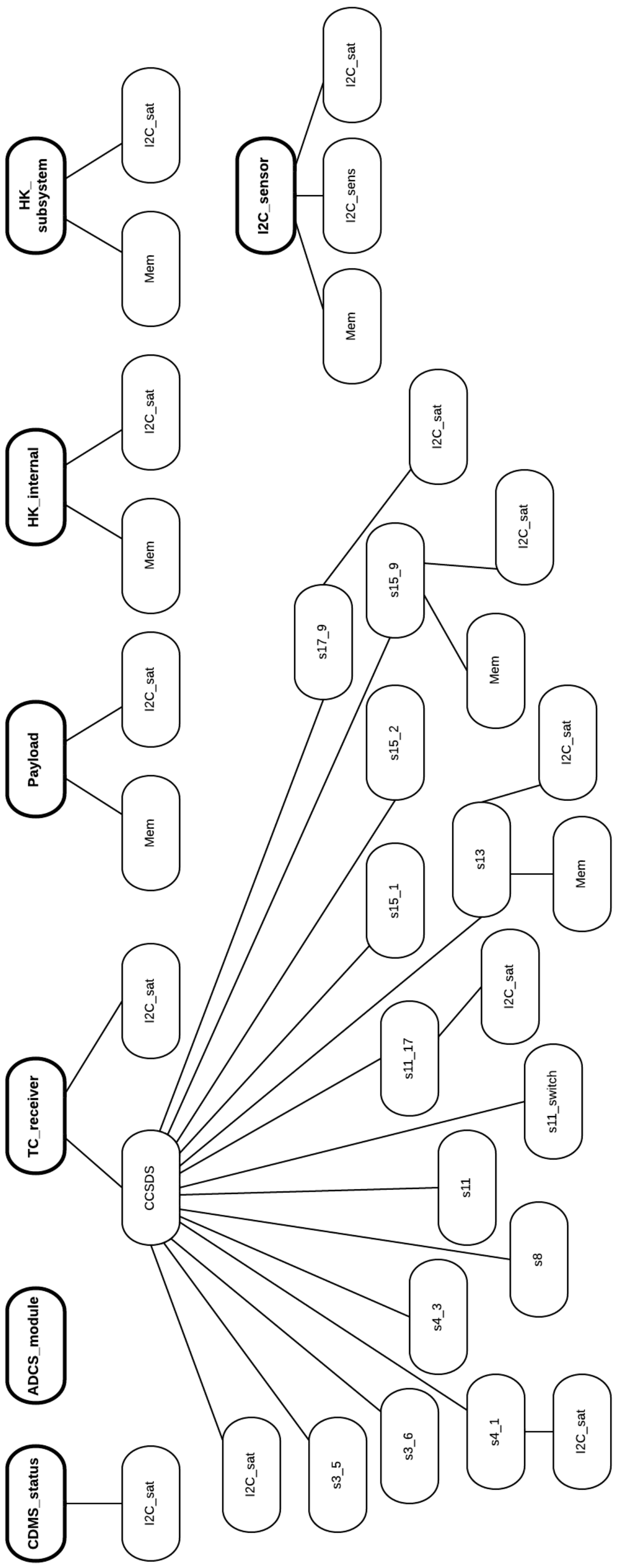


Figure 16: *Global triggering graph, the activities are represented in bold. HK_subsystem and I2C_sensor are represented only once since their structure is always the same*

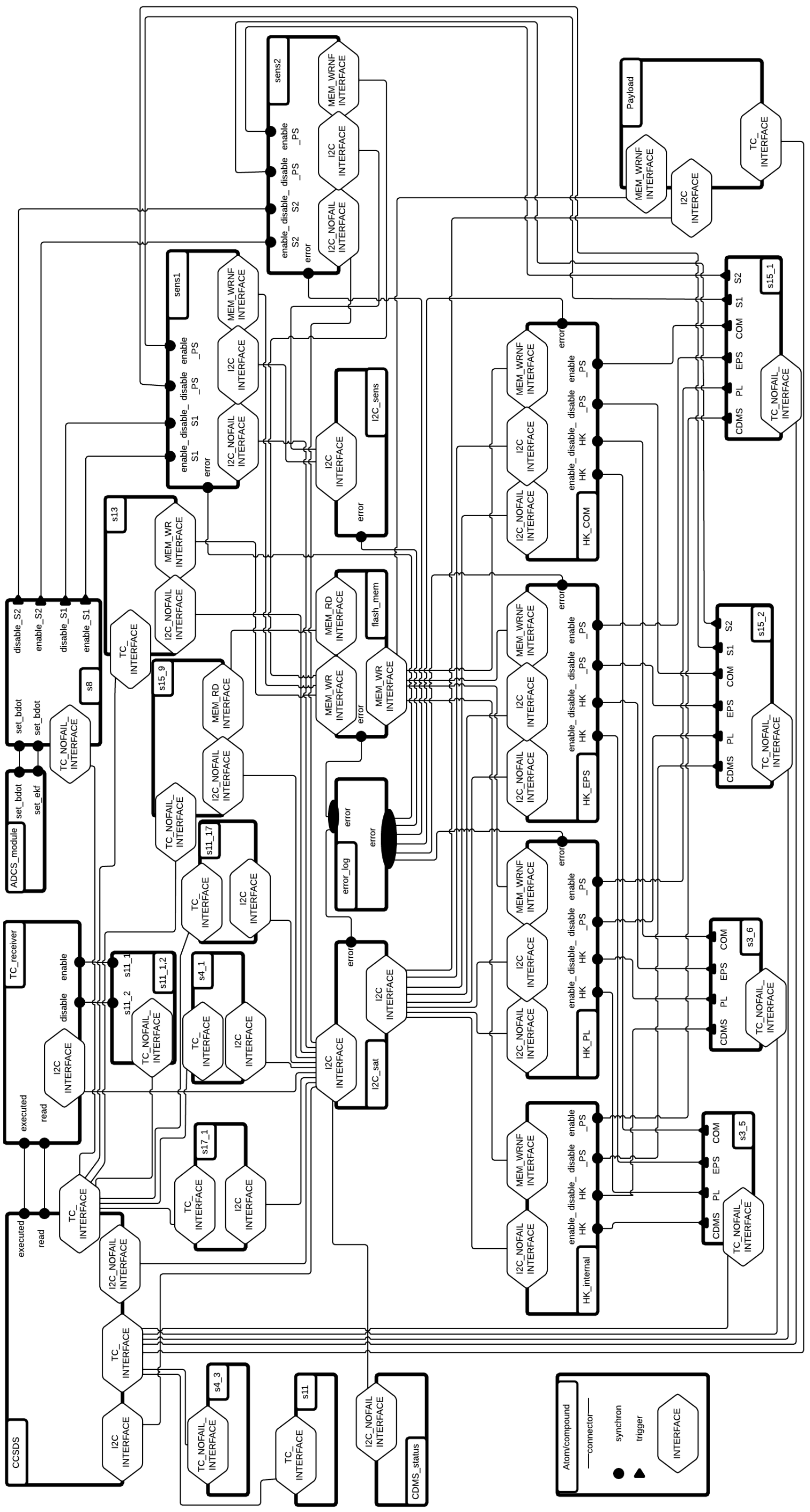


Figure 17: *"interaction" layer of the BIP CDMS model*

7.2 ADCS_module

The **ADCS_module** component is an activity. It is composed of a "software application" (ADCS_routine) and a "mode" (ADCS_mode). The events of the mode atom are "set_bdot" and "set_ekf". They depends on ground telecommands. They may enable the "ekf" or "bdot" transitions.

7.2.1 Description

The ADCS component (figure 18) is in charge of the satellite stabilization. There are two types of stabilization algorithms; one is called "EKF", the other "Bdot". The "EKF", a quite computationally expensive algorithm, is used to precisely control the attitude of the satellite. It is in fact required for the satellite to be in a determinate orientation during the payload data acquisition phase.
The "Bdot" is a simpler algorithm, which runs directly on the ADCS microcontroller.
The CDMS computes the "EKF" control values and sends them to the ADCS microcontroller, otherwise it communicates to the ADCS microcontroller to compute "Bdot" by itself.
The mode of operation is defined by the EKF and BDOT places.
There is no need to synchronize with the sensors; in fact, they are supposed to be sampled at a higher rate than the frequency of the "EKF" algorithm.

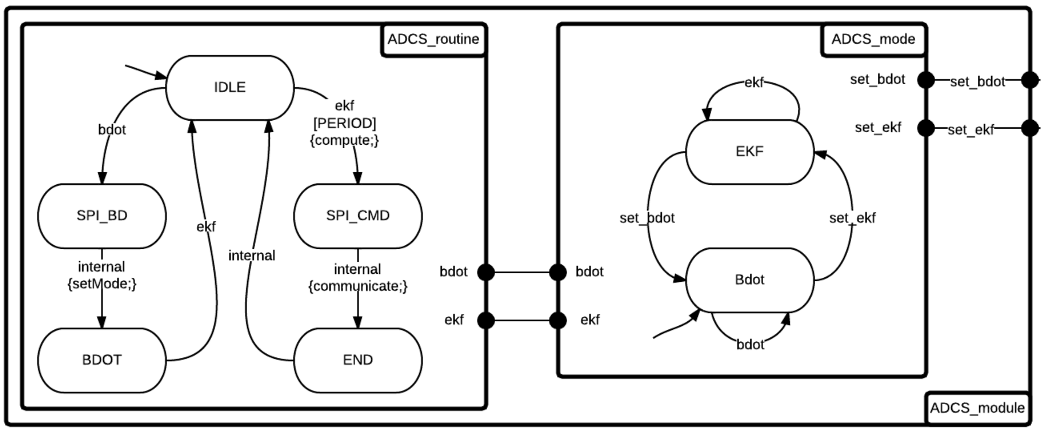


Figure 18: *ADCS_module, composed of a software application: ADCS_routine and a mode atom: ADCS_mode*

7.2.2 Discussion on correctness

The mode is selected by one to one connectors coming from the service8 atom. The latter has triggers on its connector end, so no local deadlock can be caused by this atom.
The software application, from the IDLE state can synchronize with both the modes, so there is no deadlock.
Following the bdot side, the ADCS_routine goes to the BDOT state. Here the state progression may be blocked depending on the mode. Nevertheless this is a desired behavior and can be reversed by changing the mode, thus rule 1 is satisfied.
Following the ekf side, no deadlock is possible anymore.

7.3 Payload

The **Payload** component is an activity. It is composed of:
five "software applications": s128_1, that is adding a scenario to the payload board, s128_4, that is execute scenario telecommand, s128_5, that is abort any operation on the payload, data_transfer, to transfer data from the payload to the non volatile memory and status_verification which checks the advancement of the payload board internals algorithms.
A mixed mode/status atom which coordinates everything: "status" tracks the upload, execution and result retrieval of a scenario and enable one of the software applications according to this.

7.3.1 Description

This component takes care of payload operations. Those are not completely defined yet, therefore only a minimal set of applications is implemented. It is represented in figure 19.
The payload has four macro states; IDLE, SCENARIO_READY, STARTED and RESULT_READY.
In IDLE state, the payload is not operating. In SCENARIO_READY a scenario is loaded on the payload board, thus the measurement can now start anytime. In STARTED the payload data acquisition begins. The "status_verification" module polls the payload board to know if its memory is full. Once the memory is full, the state changes to RESULT_READY. From this state, the data is transferred to the CDMS non volatile memory, with the "data_transfer" atom. Then, if the retrieval of data from the payload board is not finished, the status goes back to STARTED and the same procedure is repeated until the "complete" transition is issued.
The payload operation can be aborted any time with a telecommand (s128_5 atom).

7.3.2 Discussion on correctness

Atoms internal local deadlock freedom

The three services are very similar. The TC interface is correctly implemented, in fact, s128_5 will always loop back to the IDLE state finishing with a fail or success strong synchronization. The port labeling the "aborted" transition does not cause any deadlock because it is typed as a trigger. Moreover, the I2C_sat interface is correctly implemented as shown in section 6.2.
In the other two sub services there is an additional transition going from START to FAIL, which is internal_wrong_state. This transition happens if the corresponding "exec" or "load" transitions, which have higher priority, are not possible. This causes a fail strong synchronization, so those modules are also correct.
The data_transfer atom can progress only if the status atom is in the RESULT_READY place. In that case, it ends up with a done transition, synchronizing back with the status which goes to the STARTED place. The memory and I2C are correctly interfaced.
A similar reasoning applies to the status_verification atom. It can execute only when the status is in the STARTED place. It syncs back either with full or complete transition.

Local interfaces

In order to prove the correctness of this compound, the local interfaces between the five

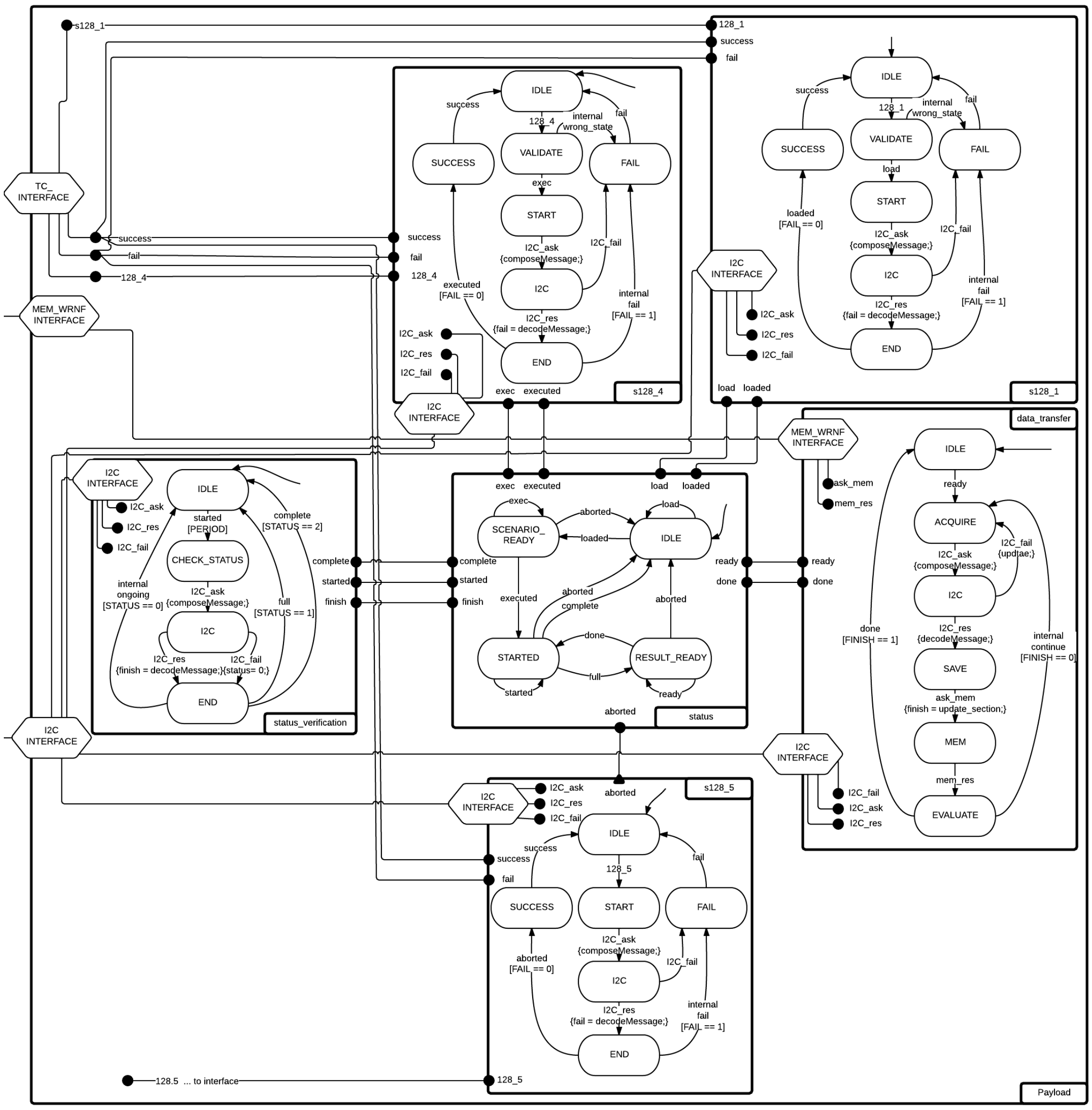


Figure 19: *Payload BIP model, composed of several software applications: s128_1, s128_4, s128_5, data_transfer, status_verification and a status atom: status*

software applications and the status must also be correct.
The s128_1 software application can only execute in IDLE otherwise it causes a fail synchronization. If successful, it changes the status to the SCENARIO_READY place with a "loaded" synchronization, thus respecting the load/loaded interface and without causing any local deadlock.
Software application s128_4 can only execute in SCENARIO_READY otherwise it cause a fail synchronization. If successful, it changes the status to the STARTED place with an "executed" synchronization, thus respecting the exec/executed interface and without causing any local deadlock.
The status_verification software application only executes in STARTED. Depending on the payload board status, it may issue a "complete" or "full" transition, the first one leading to

the RESUL_READY place, the other to IDLE. Thus the started/full or started/complete interfaces, which are well distinguished by an internal guard, are respected. Therefore this atom does not cause any local deadlock.
The data_transfer software application can only execute in RESULT_READY place. It eventually issues a transition leading back to the STARTED status. Thus the ready/done interface is respected. Therefore this atom does not cause any local deadlock.
Software application s128_5 causes an aborted transition only when the latter was successfully executed, always bringing back the status place to IDLE. As shown before, no local deadlock is caused by this atom.
Therefore, the payload compound is correct and does not cause any local deadlock. In the IDLE and SCENARIO_READY places nothing happens without a TC, but this is an intended design feature.

7.4 HK_subsystem

The **HK_subsystem** component is an activity. It is composed of:
a "software application" which communicates with the subsystem board.
A "status" that observes the correct functioning and anomalies of the monitored subsystem and starts a "reset" action when in the CRITICAL_FAILURE state.
Two "mode": HK mode which may inhibit the HK process and Packet store mode which changes the destination of the housekeeping packets (COM subsystem or non volatile memory).

7.4.1 Description

This compound is used to recover engineering data from the various subsystems of the satellite. In this system, there are three of them. The only difference is that they address a different subsystem, thus their internal C/C++ functions are different. The graphical BIP model is in figure 20.
The compound has three macro states; NOMINAL, ANOMALY and CRITICAL_FAILURE. When in NOMINAL state, the subsystem is performing correctly. If it can not be reached ("failure" transition), or if the engineering data are not correct ("error" transition), at the automata progress to the ANOMALY state. After some time in ANOMALY, the state changes to CRITICAL_FAILURE. At this point, the EPS is contacted and a restart of the malfunctioning subsystem is demanded.
During nominal operation, the subsystem is contacted to retrieve engineering data. Those data is then sent to the non volatile memory or directly to the COM subsystem, depending on the packet store atom mode.
The housekeeping can be stopped by changing the mode of the HK_mode atom.

7.4.2 Discussion on correctness

Depending on the state of HK_mode, the process can be blocked in the WAIT place. However, this is a desired feature and can be reversed by externally changing the HK_mode status.
The I2C_sat is respected, therfore there are no problems until the SEND_HK_REPORT or FAILURE states.

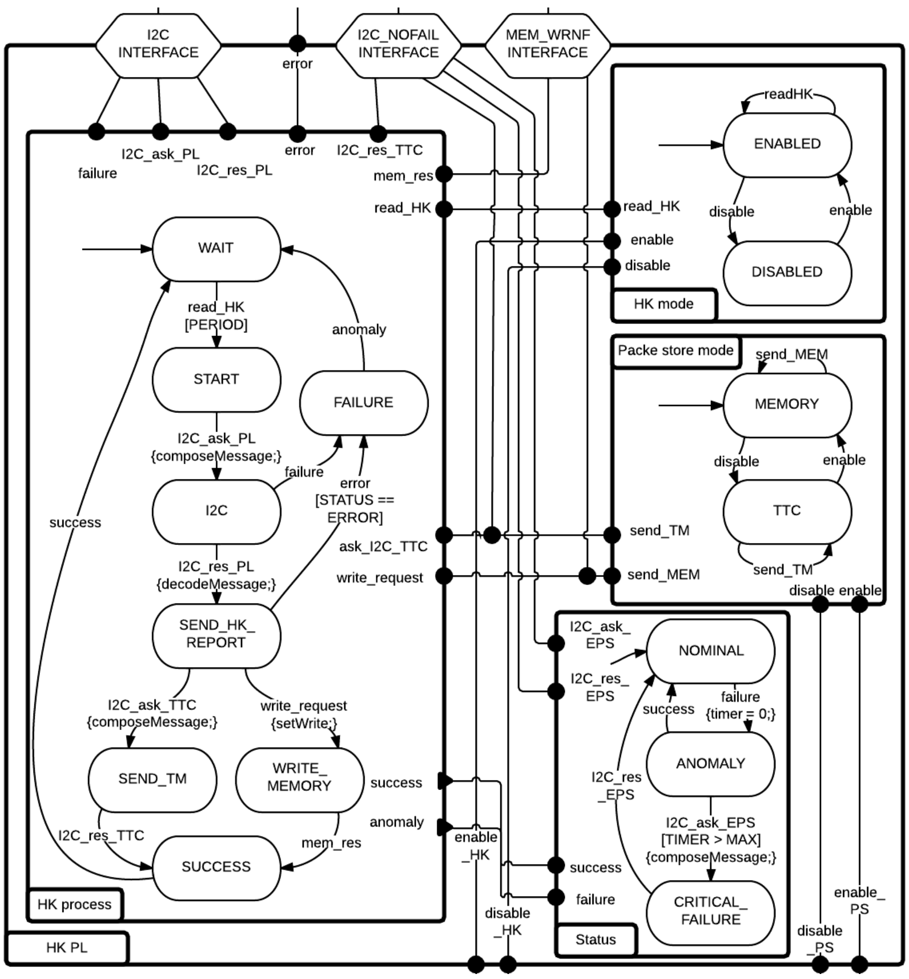


Figure 20: *HK_subsystem, composed of a software application: HK_process, a status: status and two mode atoms: Packet_store_mode and HK_mode*

Assuming the state is FAILURE, then the anomaly transition can always happen, since there is a trigger on this side of the connector.
From SEND_HK_REPORT one of the two transitions is always available (synchronized with the Packet store mode atom), the interfaces are correctly implemented, thus a transition to the SUCCESS state always occurs.
The "success" transition is always possible, for the same reason as the "anomaly".
This compound is then correct.

7.5 HK_internal

This is an activity comparable to HK_subsystem.

7.5.1 Description

The concept is the same as HK_subsystem. The difference is that the HK data is not retrieved through the I2C bus but through internal processes (e.g. GPIO and state registers). The status is also removed, in fact, the EPS subsystem is directly responsible for it, so the status of the CDMS is sent through the heartbeat shown in section 7.1.
The HK_internal compound can clearly be seen in figure 21.

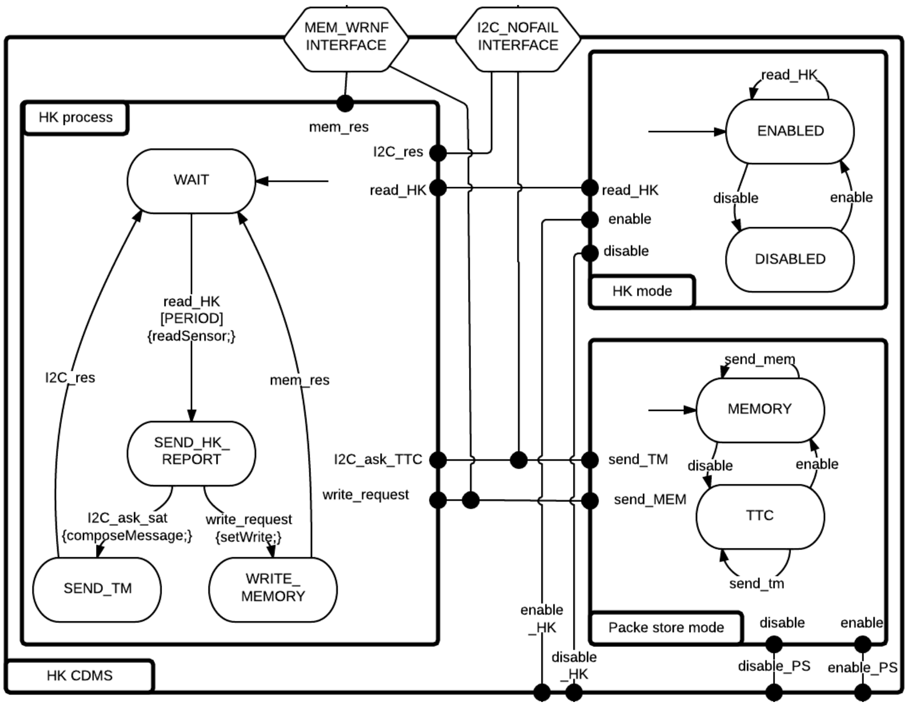


Figure 21: *HK_internal, composed of a software application: HK_process and two mode atoms: Packet_store_mode and HK_mode*

7.5.2 Discussion on correctness

The same reasoning as 7.4 applies.

7.6 I2C_sensor - Sensor connected on the I2C_sens bus

This is an activity comparable to HK_subsystem.

7.6.1 Description

The concept is the same as HK_subsystem. The difference is in the use of a different I2C bus, see figure 22.

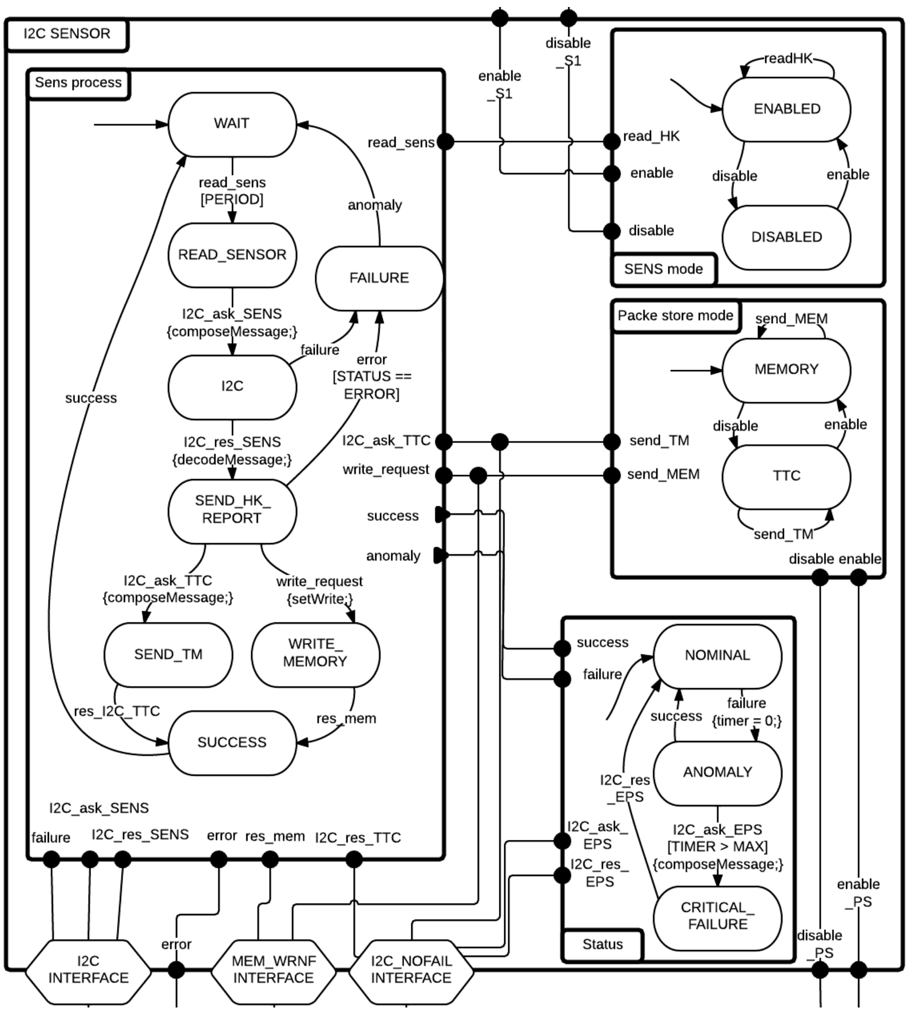


Figure 22: *I2C_sensor, composed of a software application: sens_process, a status: status and two mode atoms: Packet_store_mode and SENS_mode*

7.6.2 Discussion on correctness

The same reasoning as 7.4 applies.

7.7 TC_receiver

The **TC_receiver** component is an activity. It is composed of:
a "software application" which periodically polls the communication board to find an available TC or searches for it on the on-board schedule.
A "status" (TC_buffer) that observes the search of a telecommand and enable its treatment when it is found (from the LOADED place) as an action.
A "mode" (TC list mode) which enables or disables the execution of telecommands form the on-board schedule.

7.7.1 Description

This component is in charge of the retrieval of telecommands. There are two sources of telecommands. The first one is the COM subsystem, the other one is the list of scheduled telecommands. A direct telecommand has priority on a list telecommand. The BIP graphical model of this compound is presented in figure 23.
The TC_receiver has four macro states: EMPTY, FETCH, LOADED and EXEC. It has two modes: (TC list) ENABLED or (TC list) DISABLED.
When the state is empty, then the component looks at the COM subsystem (every defined period of time). If a telecommand is found, a synchronization progressing the state from FETCH to LOADED is executed. If not, a valid telecommand is searched from the list. If one is found, then a synchronization progressing the state from FETCH to LOADED is executed, else a synchronization progressing the state from FETCH to EMPTY.
If the list has been deactivated, then the list is not checked and a synchronization progressing the state from FETCH to EMPTY is executed.
When the state is LOADED, then the search of a telecommand is suspended until the last one has been processed.
This compound is strongly synchronized with CCSDS. They are strongly synchronized through the read and executed ports.
It has to be noted that a new telecommand can not be fetched until the current one is executed.

7.7.2 Discussion on correctness

There is a local interface (as described in section 5.4) with the CCSDS compound. It is composed of the executed and read transitions. Those are strongly synchronized with compatible ports on the CCSDS compound 7.8 through two one to one strong synchronization connectors.
The fetch synchronization, synchronizes the transition of WAIT to CHECK_TTC and EMPTY to FETCH.
The I2C interface is correctly implemented. Following a "fail" synchronization it goes back to the starting place in both the application and status atom, thus no local deadlock is generated. After a "success", from the CHECK_LIST place, provided that a TC was correctly fetched, a "write" strong synchronization happens between status and application atoms.
Since the CCSDS atom is correct (section 7.8) both atoms progress to the initial states and no local deadlock is generated.
If no TC has been fetched, and the TC_list mode is in DISABLED state, both atoms are

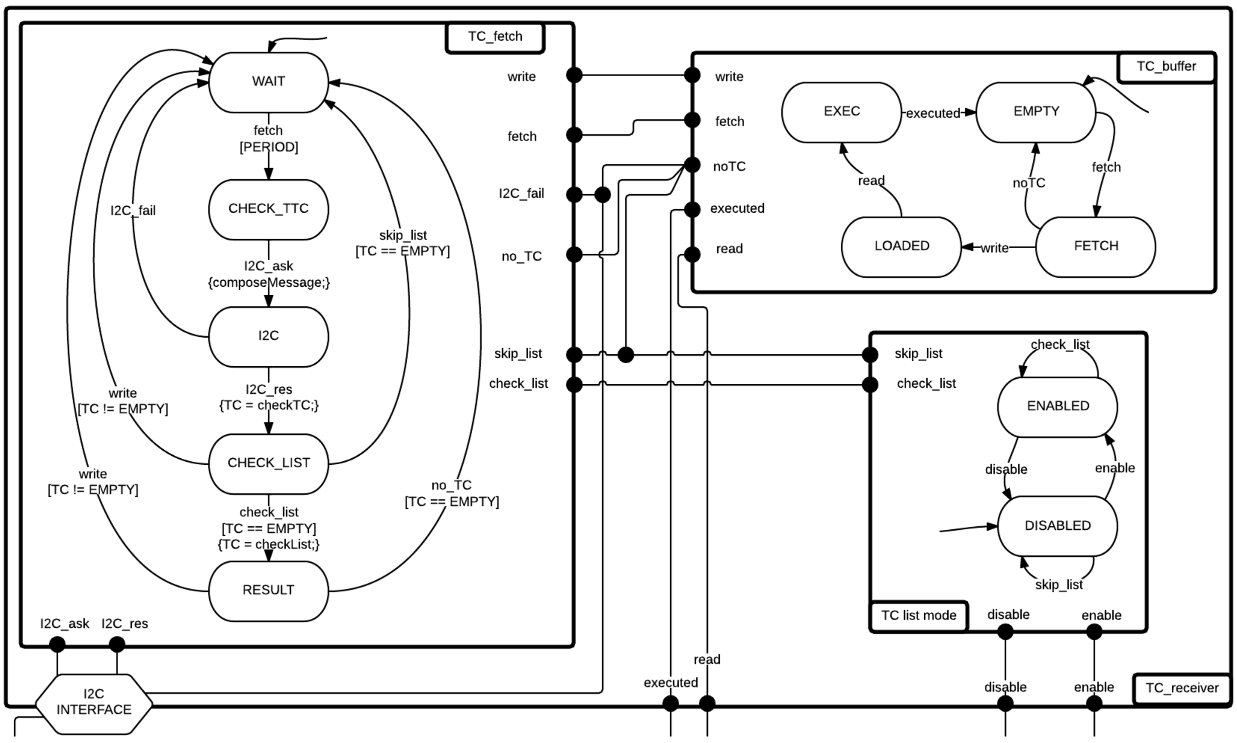


Figure 23: *TC_receiver, composed of a software application: TC_fetch, a status: TC_buffer and a mode atom: TC_list_mode*

again synchronized back to the starting place, thus no local deadlock is generated.
If the mode is ENABLED, then an internal transition changes the place of the TC_fetch atom to RESULT. Independently of the result, a sequence of synchronizations (identical to the one after CHECK_LIST) is issued.
This compound is then correct, as it does not generate any local deadlocks.

7.8 CCSDS

The **CCSDS** component is a triggered activity, it is activated when a telecommand is found through the "read" transition and terminates with the "executed" one. It is composed of:
two "software applications": the "CCSDS state machine" which process the telecommand according to the ECSS standards and "other_apid" which is used to relay TC to the other subsystems.
Two "switches": "apid_distr" which distribute the telecommand either to other subsystems or to the CDMS itself and "service_distr" which enables the atom perform the action according to the telecommand service and subservice types.

7.8.1 Description

This module (figure 24) implements the CCSDS standard state machine. After a telecommand has been fetched, it has to be checked and executed. It is also possible to have real time feedback to the ground on its execution state if required. This feedback is given by the telemetry services s1_1, s1_2, s1_7 and s1_8 only if the acknowledge variables of the TC are equal to 1. Otherwise those services are skipped.
It is important to note that **only one telecommand shall be executed at a time**.
A CRC verification is performed to assure that the telecommand is not corrupted. Then others fields of the telecommand are checked, such as service type and subtype, to verify if they are valid or not.
The third step enables the execution of the atom that implements the apid, service and subservice of the telecommand. At the end of execution, the service will synchronize a transition showing the status of the execution (success or failure). At this point, a new telecommand can be fetched.
This compound is strongly synchronized with TC_receiver. They are strongly synchronized through the read and executed ports.

7.8.2 Discussion on correctness

The process start in the CCSDS_state_machine atom, from its IDLE place, when the read transition is strongly synchronized with the TC_receiver compound.
The first thing to notice is that the local interface between the CCSDS (triggered activity) and TC_receiver (activity) are respected.
Internally, the software application of the CCSDS_state_machine atom does not cause any local deadlock. It is the case because all the interfaces with the I2C_sat resource are respected (no s1, s1_ret sequence can cause local deadlocks).
Between the READY and RESULT states there is the interface with all the telecommands ("start" - "success"/"fail"), which is also respected.
The two switches, service_distr and apid_distr can not cause a local deadlock.
The other_apid software application is correctly interfaced with the I2C_sat resource and the CCSDS_state_machine atom.
Therefore, the CCSDS compound is correct.

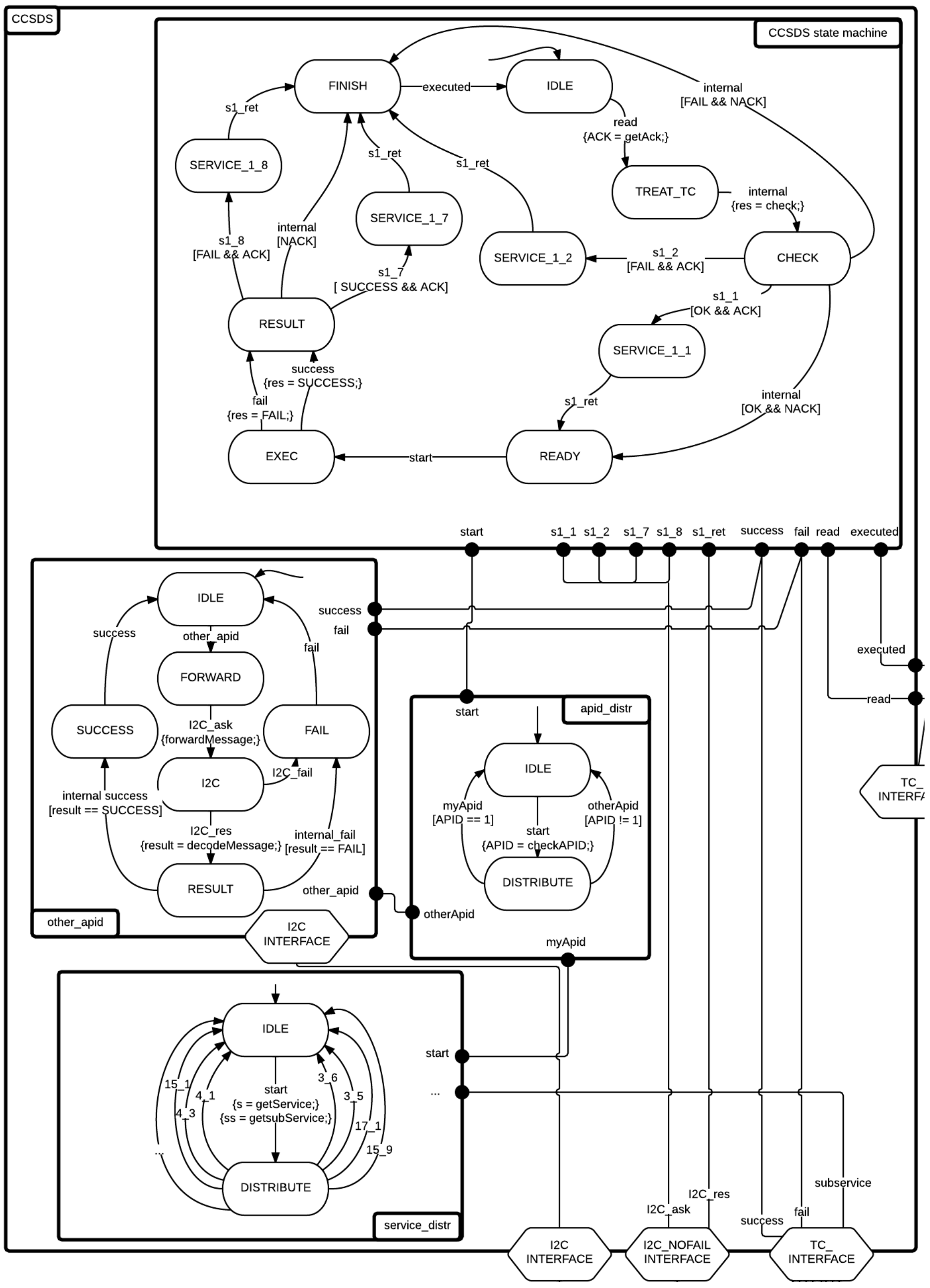


Figure 24: *BIP model of the CCSDS compound, composed of two software applications: CCSDS_state_machine and other_apid and two switches: apid_distr and service_distr*

7.9 I2C_sat

The **I2C_sat** component is a resource implementing the I2C_sat bus protocol and I2C bus 1 resource. It is composed of a single "software application".

7.9.1 Description

The protocol used to communicate through the I2C bus of the satellite, described in the CubETH ICD [2], is directly implemented through BIP as in figure 25. This is an atom representing a **resource**.
The request transition is enabled as soon as an user wants to send a message through the I2C bus. The transition is strongly synchronized. Mutual exclusion is assured by the fact that this atom is the only one implementing the use of the I2C peripheral. Therefore, 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.
It is possible to put priorities on the connectors of the different users to decide who should access the resource if two (or more) of them require it at the same engine step. Otherwise an interaction will be randomly selected.
The send transition executes the function that send a message to the selected slave on the line.
In the MASTER_READ state a feedback is fetched from the slave every designed period of time with the poll transition. After a maximum number of tentatives defined by the MAX_POLL parameter the atom produces an error and is ready for a fail transition with the user that has sent the message. The same can happen after several wrong write (MAX_WRITE).
The poll transition enables four different transitions: "no message" if the slave has not elaborated the message yet, "error" (1 or 2) if the slave answer or if the user message were corrupted and "return" if the answer was correctly received.

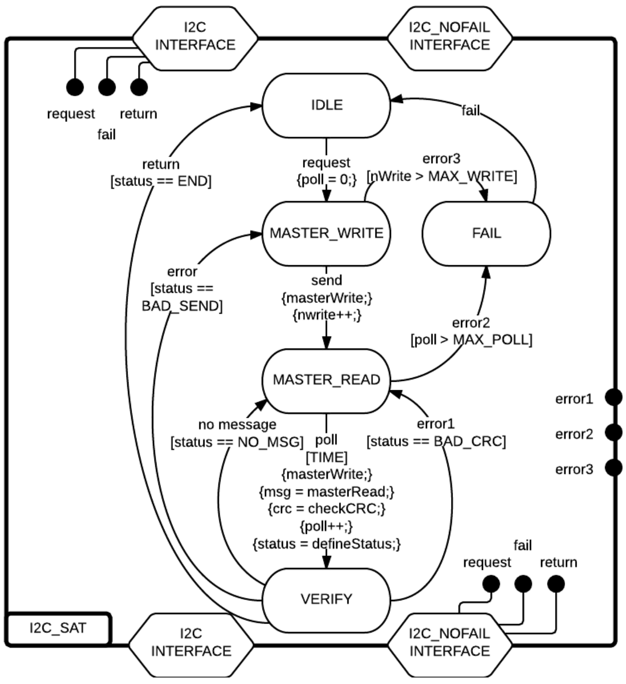

I2C INTERFACE
I2C_NOFAIL INTERFACE
request
return
fail
IDLE
fail
return [status == END]
request {poll = 0;}
error3 [nWrite > MAX_WRITE]
MASTER_WRITE
FAIL
error [status == BAD_SEND]
send {masterWrite;} {nwrite++;}
error2 [poll > MAX_POLL]
MASTER_READ
error1
error2
error3
no message [status == NO_MSG]
poll [TIME] {masterWrite;} {msg = masterRead;} {crc = checkCRC;} {poll++;} {status = defineStatus;}
error1 [status == BAD_CRC]
fail
request
return
VERIFY
I2C_SAT
I2C INTERFACE
I2C_NOFAIL INTERFACE

Figure 25: *BIP model of the atom implementing the I2C satellite high level communication protocol*

7.9.2 Discussion on correctness

This resource is structured as specified by rule 3. From the IDLE state a strongly synchronized (with the user) request transition leads to the MASTER_WRITE state. If the operation is successful, a return synchronization closes the loop. If an hardware failure occurs, no internal deadlock is created because eventually the error transition going from VERIFY to FAIL will occur, which will then lead to a fail transition.
The various error transitions can not cause a local deadlock because they are synchronized with a single state atom, as described in section 6.3.
As long as the interface from section 6.2 is respected among all components (rule 5), then no local deadlock can be created by this component.

7.10 I2C_sens

The **I2C_sens** component is a resource implementing the I2C_sens bus protocol and I2C bus 2 resource. It is composed of a single "software application".

7.10.1 Description

The sensors on the I2C_sens bus are accessed with a "I2C master write" (to select the sensor output) and a "I2C master read" as in figure 26. The result is then evaluated to determine the correctness of the output.
It has a simpler structure compared to to I2C_sat, but the basic is the same. This is a good example on how a BIP atom representing a resource may change depending on the used protocol.

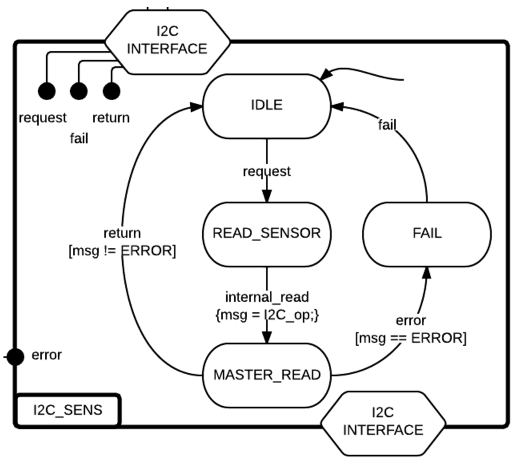


Figure 26: *BIP model of the atom implementing the I2C sensor high level communication protocol*

7.10.2 Discussion on correctness

This resource is structured as specified by rule 3 according to section 6.2. From the IDLE state a strongly synchronized (with the user) request transition leads to the READ_SENSOR state. If the operation is successful, a return synchronization closes the loop. If an hardware failure occurs, the fail transition is triggered. The internal_read transition can not cause any local deadlock. Therefore, this atom is correct.

7.11 Flash memory management

The **Flash_memory_management** component is a resource implementing the NOR flash device and its write/read protocol. It is composed of a single "software application".

7.11.1 Description

The reading and writing procedures to the external non volatile NOR flash memory are repres ented by the atom in figure 27. This is a **resource** atom.
The concept is similar to the I2C_sat atom, only the procedure is different.
The memory device can be read and written but not at the same time, hence both **write_request** and **read_request** transitions are strongly synchronized with the user that want to access the memory. If those transitions are not enabled, the user will be blocked until the resource is free again.

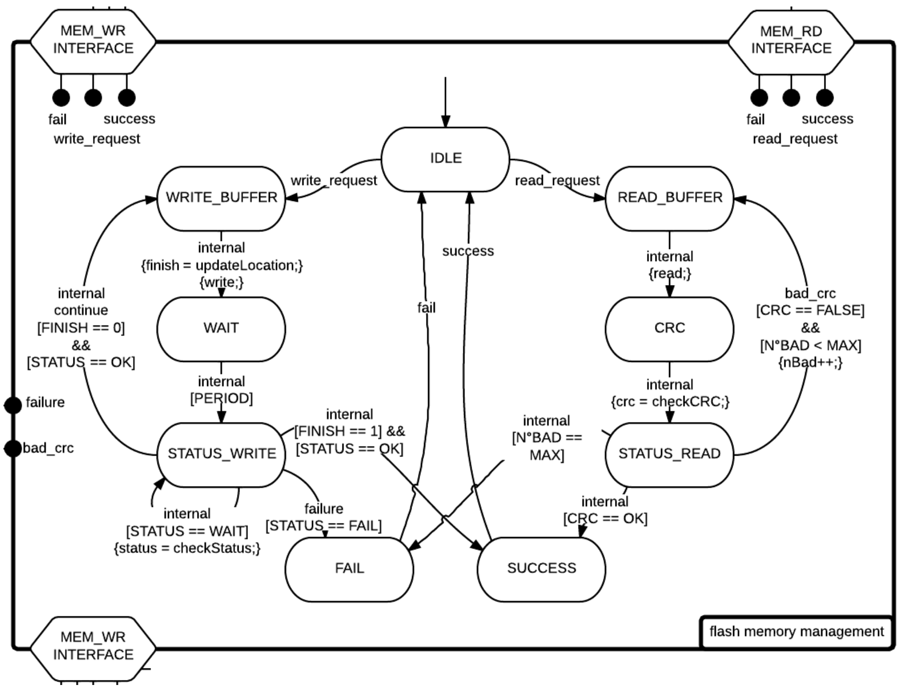


Figure 27: *BIP model of the atom implementing the high level read and write procedures to an external flash memory*

Writing procedure

After a write request transition, the atom takes the buffer prepared by the user during the previous transition and start writing it in the memory with an internal transition. Following

(from WAIT to STATUS_WRITE) there is another internal transition which waits for a certain period to elapse. This is the minimum time to wait for the memory device to effectively write the buffer internally. Then, an internal transition checks the status of the writing procedure.
If the buffer is bigger than the write sector (provided finish is equal to zero), then this procedures continues. If there is no more data to write and the procedure was successful, then an internal transition leads to the SUCCESS state.
If, for some reason, i.e. internal timeout, internal error, the memory does not perform as expected, the writing procedure is aborted, then an internal transition leads to the FAIL state with the failure transition. In fact, there is no way to react to a write error other than copy all the good data in another sector, erase the one that misbehaved and then copy them back.
It as to be noted that at this point, in the CubETH project, how to organize the external memory is not defined yet.

Reading procedure
After a read request transition, the atom reads the memory and puts data in the buffer indicated by the user during the previous transition. After the region is read, a CRC is performed during an internal transition. If the result is bad, then the memory region is read again for a MAX number of times, then a failure transition leading to FAIL state is issued. If the CRC is successful, the state progresses to SUCCESS.

7.11.2 Discussion on correctness

This resource is structured as specified by rule 3. From the IDLE state a strongly synchronized (with the user) request transition leads to the WRITE_BUFFER or READ_BUFFER state. If the operation is successful, a success synchronization closes the loop.
If an hardware failure occurs on the "write" side, no internal deadlock is created because as soon as the write procedure is not correct or times out (detected by the checkStatus; function) the failure transition synchronizes to the FAIL state.
If an hardware failure occurs on the "read" side, no internal deadlock is created because eventually the error transition going from STATUS_READ to FAIL will occur which will then lead to a fail transition.
The failure and bad_crc transitions can not cause a local deadlock because they synchronize with a single state atom, as described in section 6.3.
As long as the interface from section 6.2 is respected among all components (rule 5), then no local deadlock can be created by this component.

7.12 error_log

The **error_log** component is a resource used to protect a specific RAM region to simultaneous write access. It is composed of a single "software application".

7.12.1 Description

This simple atom (figure 28)s, with only one transition and state represent a resource. Specifically, one that implements a memory region in the RAM that is accessible by many users.

Every time a hardware error is produced, it is stored in an array managed by this atom. The goal is to avoid that two users try to write an error at the same time.

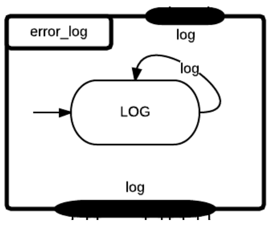


Figure 28: *BIP model of the error_log atom*

7.12.2 Discussion on correctness

This simple single atom single transition resource can not cause a local deadlock in any way.

7.13 Housekeeping report enable, housekeeping report disable, packet store storage enable and packet store storage disable - s3_5, s3_6, s15_1, s15_2

s3_5, **s3_6**, **s15_1** and **s15_2** are, as all the others telecommands, "triggered activities". They are enabled by a "start/id" (enable_HK, ...) synchronization and terminate with "finish" according to the TCNOFAIL interface presented in section 5.4.
Every component is composed of a single "switch".

7.13.1 Description

The first two subservices are respectively in charge of the activation or deactivation of the housekeeping activities. The last two modify the destination of the housekeeping data, respectively; non volatile memory or COM subsystem. Their representation is in figure 29.
As long as the telecommands packet contains transition to be triggered, the finish transition going from START to IDLE is not executed.

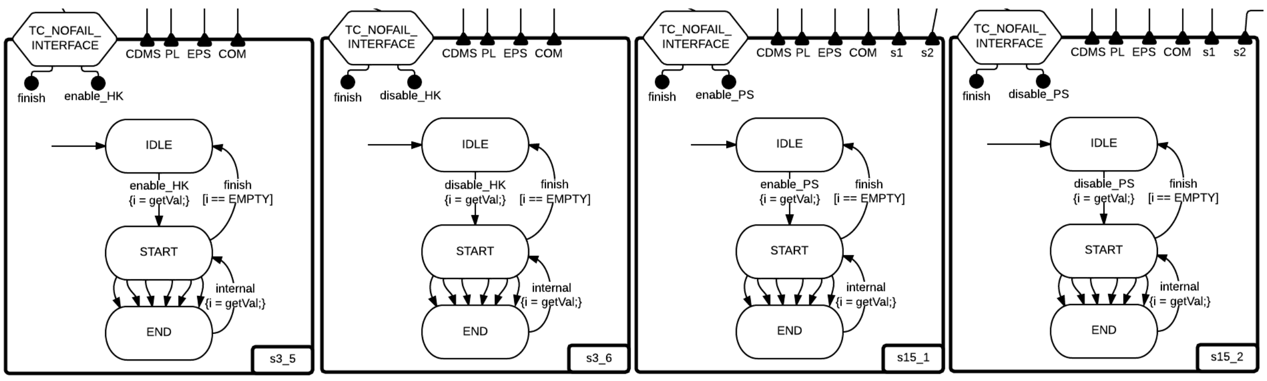


Figure 29: *BIP models of the s3_5, s3_6, s15_1 and s15_2 subservices*

7.13.2 Discussion on correctness

Those services are synchronized to the CCSDS compound through the common interface of section 5.4. All the transitions going from START to END can not cause deadlocks, since they synchronize as triggers on this side of the connector. Meanwhile, the interface is respected (rule 5). Therefore those kind of atoms can not cause any local deadlock.

7.14 Report parameter statistics - s4_1

The **s4_1** component is, as all the others telecommands, a "triggered activity". It is enabled by a "start/id" (report) synchronization and terminate with "fail" or "success" according to the TC interface presented in section 5.4.
It is composed of a single "software application".

7.14.1 Description

This component (figure 30) provides TC subservice 4_1; computation of report variables and TM subservice 4_2; downlink of report variables.

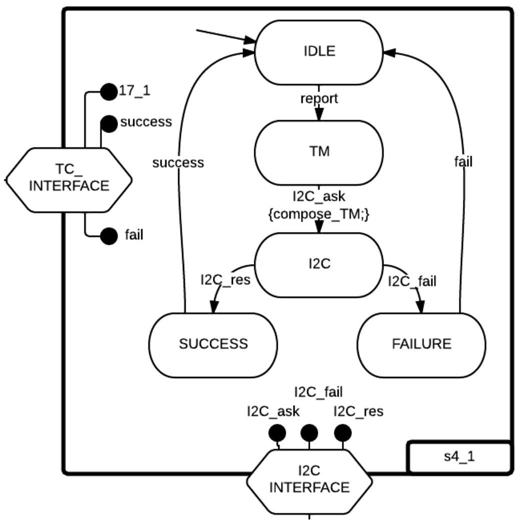


Figure 30: *BIP model of the s4_1 and s4_2 subservices*

7.14.2 Discussion on correctness

This service is synchronized to the CCSDS compound through the common interface of section 5.4, satisfying rule 5 and 2. Report is the transition going from IDLE to TM, strongly synchronized with the CCSDS compound, both success and fail close the loop. The interface with the I2C_sat resource is respected (see section 6.2).
Therefore those kinds of atoms can not cause any local deadlock.

7.15 Reset parameter statistics - s4_3

The **s4_3** component is, as all the others telecommands, a "triggered activity". It is enabled by a "start/id" (clear) synchronization and terminate with "finish" according to the TCNO-FAIL interface presented in section 5.4.
It is composed of a single "software application".

7.15.1 Description

This component (figure 31) provides TC subservice 4_3; the reset of report variables.

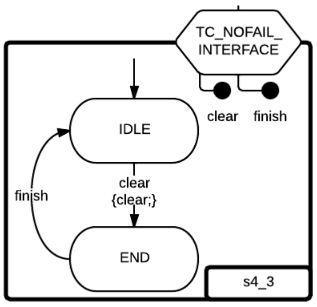


Figure 31: *BIP model of the s4_3 subservice*

7.15.2 Discussion on correctness

This atom correctly implements the TC interface, thus it is correct.

7.16 Function management service - s8

The **s8** component is, as all the others telecommands, a "triggered activity". It is enabled by a "start/id" (enable) synchronization and terminate with "finish" according to the TC interface presented in section 5.4.
It is composed of a single "switch".
This component may be expanded to a "success"/"fail" interface if some specific functions are added and they may produce errors. This service is, in this stage of the project, not completely defined.

7.16.1 Description

This is the implementation of service 8 (figure 32), depending both on the ECSS standard and on the CubETH ICD.

7.16.2 Discussion on correctness

See section 7.13.

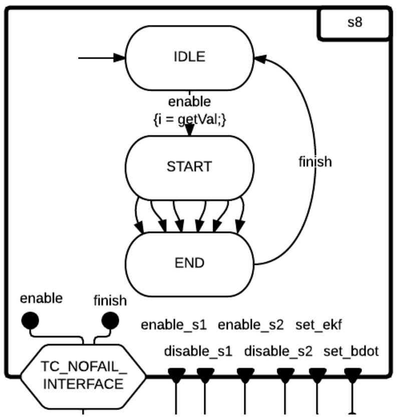


Figure 32: *BIP model of service 8*

7.17 On-board operations scheduling service - s11

The **s11** component is, as all the others telecommands, a "triggered activity". It is enabled by a "start/id" synchronization. It terminates with "fail" or "success" according to the TC interface presented in section 5.4.

7.17.1 Description

This atom (figure 33) implements sub services 11_4 (add telecommand to schedule), 11_5 (delete telecommand from schedule), 11_6 (delete telecommand from schedule over a time period), 11_7 (time shift selected telecommands) and 11_15 (time shift all telecommands).
It is composed of a single "software application".
It is important to notice that this is and TC_receiver are the only components supervising the TC on-board schedule. The schedule can not be corrupted because the TC_receiver has to wait the end of a telecommand execution before manipulating it again. Therefore the schedule will never be manipulated simultaneously by two different atoms.

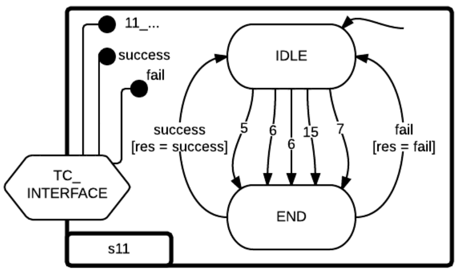


Figure 33: *BIP model of some s11 subservices*

7.17.2 Discussion on correctness

All the transitions going from IDLE to END are algorithms that can not cause deadlocks (i.e. they are only synchronized with the CCSDS compound). The TC interface is respected. Rules 2 and 5 are respected and the atom can not cause any local deadlock.

7.18 Enable/disable the release of telecommands - s11_switch

The **s11_switch** component is, as all the others telecommands, a "triggered activity". It is enabled by a "start/id" synchronization. It terminates with "finish" according to the TCNOFAIL interface presented in section 5.4.

7.18.1 Description

This is the implementation (figure 34) of TC subservice 11_1 and 11_2. That is the activation/deactivation of the release of telecommands from the schedule.

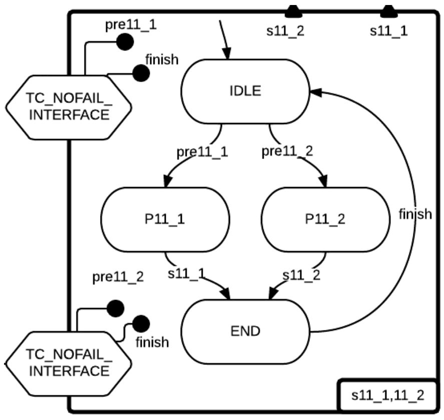


Figure 34: *BIP model of s11 subservices 1 and 2*

7.18.2 Discussion on correctness

The CCSDS interface is respected; both s11_1 and s11_2 synchronize with pre11_1 and pre11_2. The following transition, synchronized with the TC_receiver compound has a trigger on this side of the connector, so no local deadlock is possible.
In the end, a finish transition synchronize back. The rules are respected and therefore the atom is correct.

7.19 Report command schedule in summary form - s11_17

The **s11_17** component is, as all the others telecommands, a "triggered activity". It is enabled by a "start/id" synchronization (report). It terminates with "fail" or "success" according to the TC interface presented in section 5.4.
It is composed of a single "software application".

7.19.1 Description

This is the implementation (figure 35) of TC subservice 11_17 and TM 11_13. That is the downlink of the current on board TC schedule.

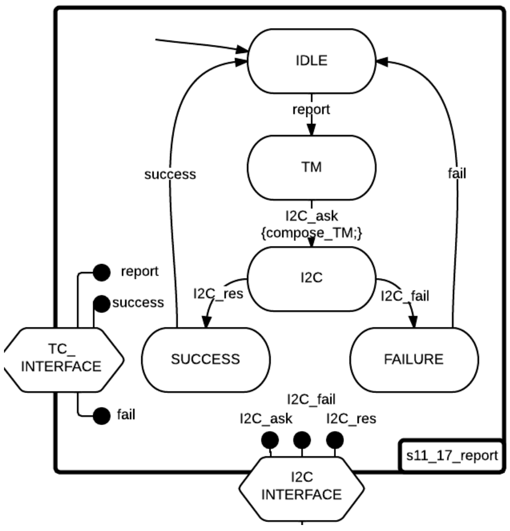


Figure 35: *BIP model of the s11_17 and s11_13 subservices*

7.19.2 Discussion on correctness

See section 7.14.

7.20 Large data transfer service - s13

The **s13** component is, as all the others telecommands, a "triggered activity". It is enabled by a "start/id" synchronization depending on the subservice (s13_9, s13_10, s13_11, s13_12) type from the IDLE or NEXT place. It terminates (either to IDLE or NEXT place) with "fail" or "success" according to the TC interface presented in section 5.4.
It is composed of a single "software application".

7.20.1 Description

Service 13 is the compound (figure 36) implementing the large uplink service. A large uplink is initiated by a 13_9 TC, progressed by a 13_10 TC and terminated with a 13_11 TC. A 13_12 TC can be issued anytime to abort the uplink. Two TM subservices are in place for ground feedback; 13_14 (correct reception) and 13_16 (abort confirmation).

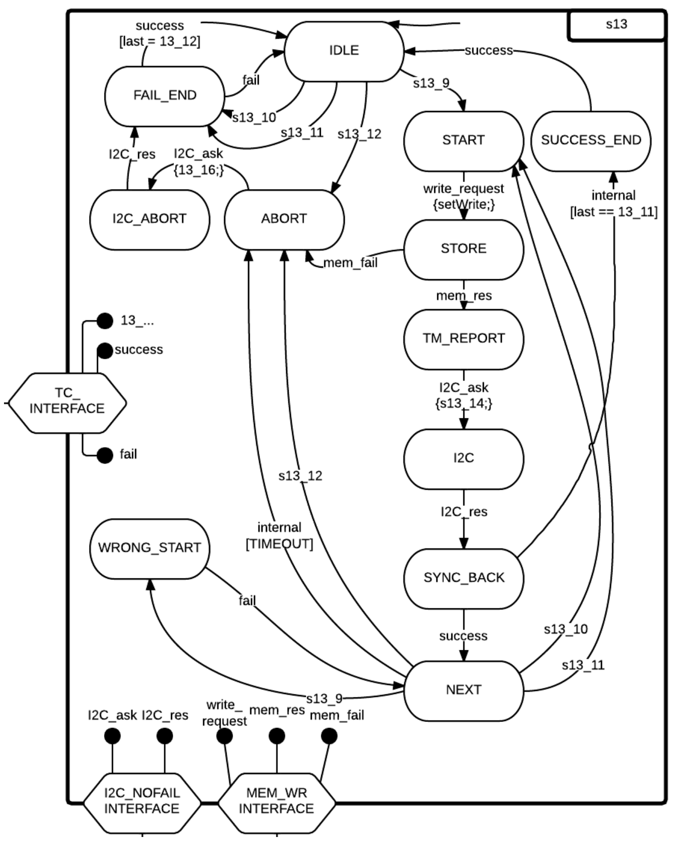


Figure 36: *BIP model of the s13 subservices related to the large data uplink*

7.20.2 Discussion on correctness

The telecommand (success, fail) interface is respected. In fact there are two starting points; one is IDLE, the other is NEXT. Both can synchronize with the four sub services. Both ultimately lead to a success or fail transitions.
From IDLE, if a "wrong" telecommand is issued (not s13_9), then the state change to

FAIL_END. From here, a fail transition closes the loop. A success transition also closes the loop if the last telecommands was the s13_12. This is because the goal of that TC is to abort the process.
If the state switches to START, then there is a correctly implemented memory interface. If the memory operation is successful, the atom goes to the TM_report state. Once again the I2C_sat no fail interface is respected. This leads to SYNC_BACK status, which then satisfies the TC interface, either by issuing directly a success or going to the place SUCCES_END if s13_11 was issued.
It is also important to notice that the s13 can not locally deadlock in the NEXT state because there is an internal timeout (required by the ECSS standard) which would ultimately lead to a fail transition. If a s13_9 TC is issued form the NEXT place, then a fail synchronization follows.
This module is therefore correct.

7.21 Downlink packet store contents over time period - s15_9

The **s15_9** component is, as all the others telecommands, a "triggered activity". It is enabled by a "start/id" synchronization (15_9). It terminates with "finish" according to the TCNOFAIL interface presented in section 5.4.
It is composed of a single "software application".

7.21.1 Description

This is the implementation of the subservice used to downlink data present in the non volatile memory of the CDMS (figure 37). A packet store and a time period are issued with a 15_9 TC and the valid packets are downlinked using the 15_8 TM subservice.

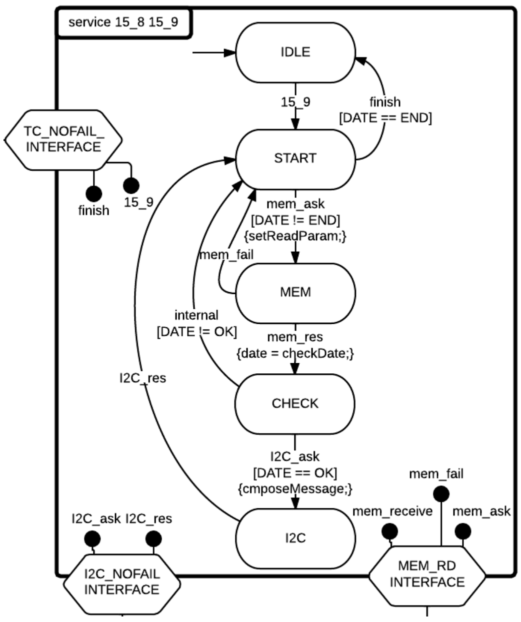


Figure 37: *BIP model of the s15 subservices related to the packet store downlink*

7.21.2 Discussion on correctness

This service is synchronized to the CCSDS compound through the common interface of section 5.4, satisfying rule 5 and 2. 15_9 is the transition going from IDLE to TM, strongly synchronized with the CCSDS compound. The loop is only closed by a finish transition, which happens when all the packets have been checked (a packet which is corrupted will be skipped).
The interface with the memory resource is respected (see section 6.2).
Therefore this atom can not cause any local deadlock.

7.22 Test service - s17_1

The **s17_1** component is, as all the others telecommands, a "triggered activity". It is enabled by a "start/id" synchronization (17_1). It terminates with "fail" or "success" according to the TC interface presented in section 5.4.
It is composed of a single "software application".

7.22.1 Description

This atom implements service 17 (figure 38), with TC 17_1 and TM 17_2. It is a simple connection test between the ground and the satellite.

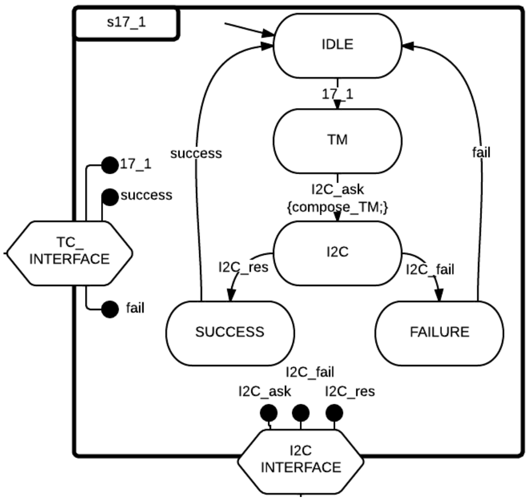


Figure 38: *BIP model of the s17 service*

7.22.2 Discussion on correctness

See section 7.14.

8 Examples of rule violations

After the definition of the rules and the analysis of the model enforcing them, some errors were found. They are presented here along with the rules they violate.
In the first example, in figure 39 (compare to figure 24) the rule 5 is broken. The interfaces are somewhat respected in the sense that, after a telecommand is started there always are a success or fail strong synchronization in the end bringing the CCSDS_state_machine atom to the RESULT place. The problem here is that the other_apid transition instead of syncing with a service, syncs directly with the I2C_sat resource request transition. The same is for success (result) and fail (fail). In other words, a TC_INTERFACE is connected to a I2C_INTERFACE which is not a valid connection as it is not represented in section 5.4.

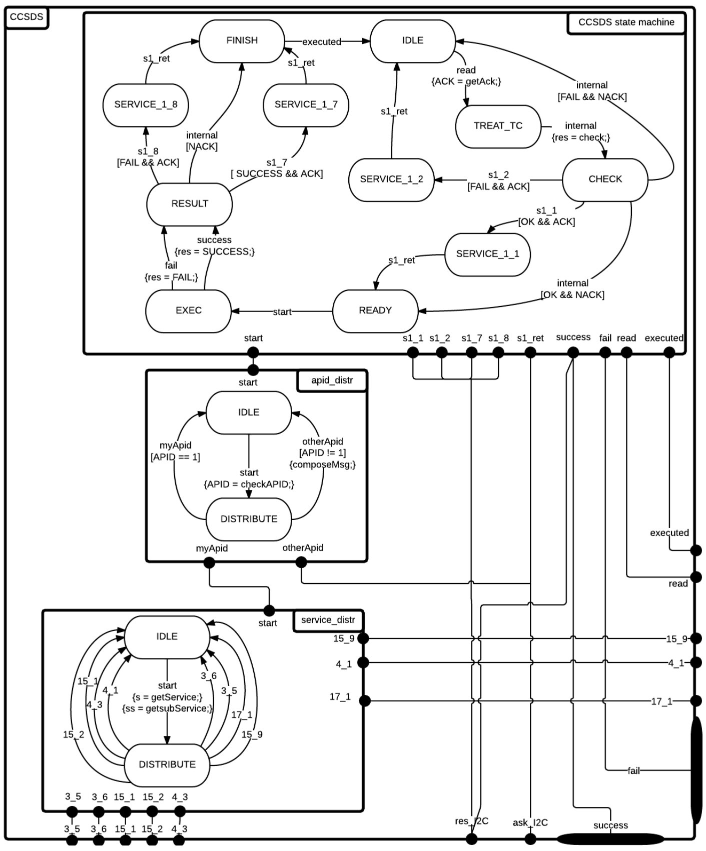


Figure 39: *Errors in a previous version of the CCSDS compound*

Here is what can go wrong:
CCSDS_state_machine is in place EXEC, apid_distr just synchronized through the start port and is in DISTRIBUTE PLACE. Here, the I2C resource has not been "reserved" yet (otherApid has not been performed), but the res_I2C port on the CCSDS compound (which is internally connected to CCSDS_state_machine "success" port) can be enabled. The reason is: since the resource is not reserved, maybe someone else is already using it. Therefore the I2C_sat atom "return" port may synchronize either with the original user, either with the CCSDS_state_machine "success". This would have lead the state of the CCSDS_state_machine atom to RESULT without having performed the telecommand, but leaving the original re-

source user "unfinished".
In other words, an I2C_fail/res synchronization could have been performed without a previous I2C_ask, this, along with the mixed interface would have caused unpredictable behavior. Another error in this design is the internal transition form the CHECK to the IDLE place, which skip the executed transition, thus breaking rule 5. Suppose that from the CHECK place the internal transition leading to IDLE is executed. Then, in the TC_receiver compound, the TC_buffer atom place will stay in EXEC, never going back to EMPTY, therefore no telecommand will be fetched nor executed again. This is an example of the problem that may arise when connecting components: in the CCSDS compound going back to IDLE state after an error seems reasonable, but in fact causes a local deadlock.

Another error is presented in figure 40 (compare with figure 36). In the state NEXT the "success" transition can always be executed, thus invalidating rule 2. A success transition could happen even without the proper service issuing it.
Imagine that a s13_9 TC is issued, then the atom place progresses to NEXT, here a success synchronization with the CCSDS compound happens (and it is necessary, but for the application purpose it does not leads back to the initial state). Then, the atom waits in the NEXT place. Suppose that a new TC, not part of the s13 family, is issued (e.g. s17_1). As soon as the CCSDS start synchronization is executed, the success synchronization through the connector connecting the CCSDS compound and the s13 atom is possible again.

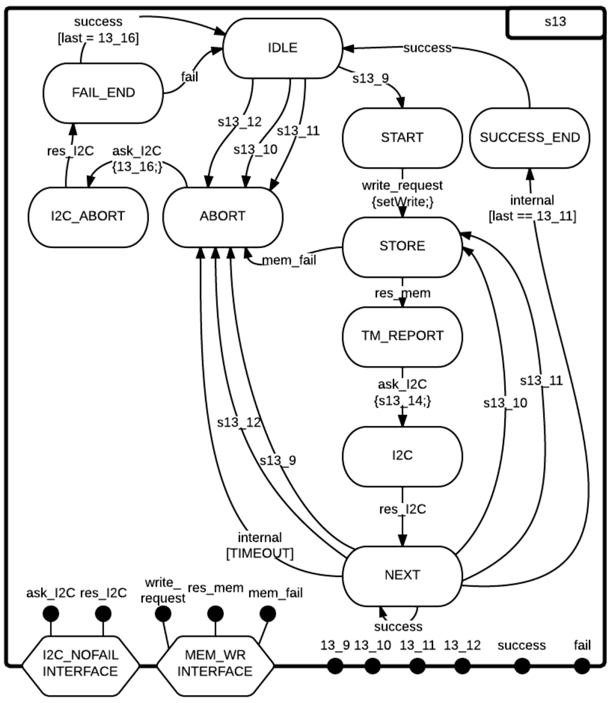


Figure 40: *Error in a previous version of the s13 atom*

9 How to integrate C/C++ functions in BIP

This section proposes some conventions to integrate C/C++ functions in the BIP model.
Complexity in the C/C++ integration arises when a function requires information from other atoms in order to be performed.
A practical example is the transfer of a telecommand from the atom receiving it (tcReceiver, section 7.7) to the service that uses its content (for example, s128_1 in the payload compound, section 7.3).
In such a case, the C function implementing the TC 128_1 requires data coming from the TC packet, which size can be up to 512 bytes. If this information is transferred using the BIP framework, its path would be: I2C_sat –> I2C-tcReceiver-connector –> tcReceiver –> tcReceiver-CCSDS-connector –> CCSDS –> CCSDS-s128_1-connector –> s128_1. Hence, the data would be copied up to 7 times, for a total of 3.5 kB permanently occupied in the RAM (the connectors and atoms variables are always allocated).
It is obvious that both for speed and memory performance such a procedure is really inefficient. Moreover, there are more than 10 atoms managing TC, so the RAM footprint increases further.
To solve the issue, the approach in section 9.1 is proposed. The cases in the CDMS software where this can be applied are shown in section 9.2 on TABLE 2.

9.1 Internal memory (RAM) management

Global variables must be defined, for example in a global_variables.c and global_variables.h files. These must be included in the .bip files implementing the C/C++ functions using those variables.
In order to avoid conflicts, the transitions in BIP where the variable is used must be clearly defined. For example, from the case of the previous tcReceiver example the TC structure is written only when a TC is received. Since only one telecommand can be treated at any time it is clear that this variable will not be written again until it has been used. Therefore, there will not be conflicts over this variable.
Another example is the I2C message. The maximum size of an I2C message is not defined yet, but it will not be smaller than 512 bytes. Nonetheless, for every I2C bus, only one message can be sent at any time. Therefore the I2C message will be somehow associated with the I2C_sat resource.
In this model, the I2C_send message structure will be written during the user I2C_ask transition (as from section 6.2, figure 11) and read by the resource internal_send transition (from figure 25). This procedure, if it is always correctly applied, avoids that the I2C_send structure is modified before it is used. In fact, no other user can write it until it can issue an I2C_ask transition, which is possible only when the I2C_resource is not used.
The same reasoning applies to the I2C_receive message structure. It is modified by the internal_poll transition of the resource (figure 25) and is used by the I2C_res transition from the user (figure 11).

9.2 CDMS global variables

Many of those global variables (TABLE 2) are in reality structures. A draft of their composition can be found on appendix C.

Table 2: Global variables used in the model, associated with the transitions where the variable is read or written. The format is "atom.transition_label"

Variable	Read transitions	Write transitions	Description
I2C_request	I2C_sat.internal_send	user.I2C_ask	Structure containing I2C message and address (sat)
I2C_report	user.I2C_res	I2C_sat.internal_poll	Structure containing I2C slave message (sat)
I2C_request	I2C_sat.internal_send	user.I2C_ask	Structure containing I2C message and address (sens)
I2C_report	user.I2C_res	I2C_sat.internal_poll	Structure containing I2C slave message (sens)
hw_error	log.error_log	atom.error	Table containing all the hw generated errors
TC_active	service.internal CCSDS.read	TC_fetch.I2C_res	Structure containing the current used TC
write_buffer	flash.internal_write	user.mem_ask	Structure for the packet going in ext. memory
read_buffer	user.mem_ask	flash.internal_read	Structure for the packet read from ext. memory
TC_schedule	TC_fetch.check_list	s11_4,5,6,7,15	Table of stored TC schedule
HK_COM_report	s4_1.compose_TM	HK_COM.decode_message	COM subsystem report parameters structure
HK_EPS_report	s4_1.compose_TM	HK_EPS.decode_message	EPS subsystem report parameters structure
HK_PL_report	s4_1.compose_TM	HK_PL.decode_message	PL subsystem report parameters structure
HK_internal_r	s4_1.compose_TM	HK_int.decode_message	internal report parameters structuret
sens1	ADCS_routine.ekf	sens1.I2C_res_sens	Sensor1 structure output
sens2	ADCS_routine.ekf	sens2.I2C_res_sens	Sensor2 structure output

9.3 Definition of C/C++ functions to implement

Thanks to the BIP model it is possible to see which functions needs to be developed. According to the definitions in section 5.1 all the functions are already identifiables in the components BIP graphical models (names between {function;}). Nonetheless a more comprehensive list of functions with variables used (according to section 9.2) is presented in appendix C.

10 Implementing the BIP model on the satellite hardware

The BIP parser (brief procedure explanation in section 10.1) produces a C++ output. Therefore a C++ compiler for the target platform (ARM Cortex-M3) is required. Moreover, BIP needs STL C++ libraries, according to the C++11 standard. Therefore the C++ compiler shall be C++11 compliant.
The BIP engine, which is closed source at the moment, is precompiled for an x86 architecture and is linked to the BIP model as a static library (bipengine.a). Thus, it has to be recompiled for the target platform.
The choice of the toolchain used to cross compile the BIP model is presented in section 10.2. In order to execute the BIP model on this embedded platform while satisfying space safety guidelines, some changes to the engine may be required, as explained in section 10.3.
Additional information and description on the cmake files and toolchain is given in appendix A.

10.1 BIP parser

The BIP model is described in one (or several) .bip files. Additional C/C++ libraries can be used by including the sources directly in the .bip files.
The BIP parser produces a directory containing the various C++ source files (for every atom, connector, port and compound) and the cmake files (various CMakeLists.txt) used to create the makefile. The cmake files also contain all the informations for the linker to link against the BIP engine static library and the others external C/C++ code sources.
To compile the executable, one shall call cmake to create the makefile and then "make".

10.2 Cross compilation

Several options to compile C and C++ code for the ARM Cortex-M3 exists.

IAR workbench
The most trivial one is to use the official tools provided by IAR (IAR workbench). The downside of this approach is that the IDE may not be very friendly with the outputs generated by the BIP parser (structure of the source code folder and cmake files). Moreover, said IDE is not free to use; a license must be purchased.

GNU Tools for ARM Embedded Processors
Another option is to manually create a toolchain using the GNU compilers from [12]. This is indeed a good solution because it can easily be integrated with the cmake files. In fact, one can create a toolchain file for cmake which assigns the good compilers. A downside is that the debug tools are not easy to use as the ones included in the official IDE.
Unfortunately, this solution does not work with C++ compilation. It seems that some specific ARM libraries to build C++ are either missing or incomplete, at least, for this specific chip. Another possibility is that the "retargetio.c" file provided by EnergyMicro/SiliconLabs does not support this specific toolchain.

Code sourcery tools

A solution to the previous problem is to change the compiler, using the ones provided by code sorcery. The compilers are free to use, but the various standard C/C++ functions of the free version are not optimized for embedded systems (they still works, but are heavier).

10.3 BIP engine adaptations

In those small microcontrollers, such as the EFM32GG880 the program and data memories are quite small, moreover the CPU works at a (relatively) slow frequency.
For the BIP model to run on this platform code size must be reduced. A way to do that would be to eliminate all the text feedbacks (which are either way useless, since there is no screen) and to eliminate some BIP features which may not be used in this kind of system. For example the capabilities of data transfer and function execution in connectors can be completely eliminated.
Another code reduction could be achieved by modifying the BIP parser to output plain C instead of C++. Nonetheless a serious study should be performed before endeavoring in this solution.
It is important, for space applications, to eliminate dynamic memory allocation. If BIP use this, it should be modified to simply use static allocation.

11 Proposed design flow for a BIP model integrated with the complete system development

A suggested procedure to design a system software with the BIP framework is presented in figure 41.

Once the hardware is chosen and the software requirements clearly defined one can start the development of the system software behavior with BIP. At this point, properties of the model can already be proven (e.g. deadlock freedom or absence of local deadlocks).
During (or after) the BIP model design one can identify all the C/C++ functions that are necessary along with the variables that they use.
It is suggested to develop those functions outside of the BIP model, since it will be easier to test them. Once they are correct (they also have to be verified outside BIP), they can be integrated in the BIP model and they can be tested and verified alongside the rest of the software.
The **BIP model design** phase procedure is proposed as follows:

- Identify the activities (according to section 5.2.1) needed to satisfy the software requirements
- Identify the resources (according to section 5.2.2) based on the hardware and define their interfaces (section 5.4)
- Identify the triggered activities (according to section 5.2.3), that is specifics environment events and how to process them

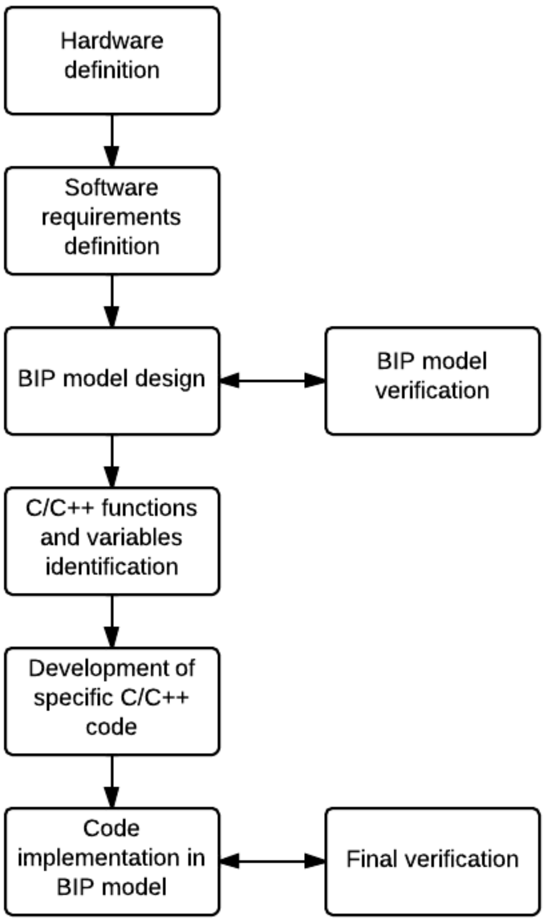

Hardware definition
Software requirements definition
BIP model design
BIP model verification
C/C++ functions and variables identification
Development of specific C/C++ code
Code implementation in BIP model
Final verification

Figure 41: *System software design with BIP framework*

- Identify the dependancies between activities (or triggered activities) and resources; define other possible interfaces
- Build a compound to support every activity, triggered activity and resource using the elements presented in section 5.3
- Connect all the interfaces and (if needed) synchronize the activities using connectors
- Iterate

12 Considerations on the BIP framework

An evaluation of system modeling with BIP framework is presented in section 12.1. In section 12.2 some features to improve the BIP framework are suggested.

12.1 BIP design considerations

The BIP framework is a promising that allows to model the system software behavior in a component based fashion. The components can then be assembled using simple rules via

connectors to express their interactions. Priorities can be added to choose between interactions available at the same time.
The graphical way of building the model is easy to comprehend and apply, but remains a powerful representation of the system. This, associated with a potential automatic code generation, can strongly simplify the design of complex systems. Moreover, it allows the creation of "correct-by-construction" components and opens the way to automatic ways for model checking. Therefore, not only the design phase is simplified thanks to its semantic, but the reliability of the model can be proved without extensive testing [9].
Nonetheless, the tools associated to the BIP framework are not yet ready for an industrial use. Even if the theoretical background is solid the tools for the practical use are not mature.

12.2 BIP development suggestions

Graphical interface and automatic BIP code generation. The possibility to design the BIP model using a GUI would largely simplify and accelerate the design process. Moreover, a source of error would be eliminated, that is, the transcription of the BIP code from the BIP graphical model. Contrariwise to C, where there are many ways to implement the same function (or diagram), the translation of BIP graphical model into the BIP code model has only one possible interpretation.

Better memory management. A problem with the BIP framework is memory management (see section 9.1). The global variables here are introduced in order to save on: unnecessary BIP model complication (data exchange through many atoms and connectors), unnecessary memory use and of course unnecessary processor time use.
The problem with this procedure is that now the BIP model is completely unaware of the existence of those variables. Therefore it is impossible to use automatic model checking procedures to determine the absence of memory conflicts. A solution for this issue must be found. The solutions presented in section 6.3 can be used to solve a part of the problem, by synchronizing the transition with a token representing a variable every time that the latter is used. However, this solution works only if the developer remembers to associate the use of the variable with a strong synchronization every time. The model is unaware of the fact that this requirement is effectively met or not.
A suggestion, which solves the problem only partially, is to create in place of a global variable a variable associated to an atom. This variable can be viewed and modified by other atoms only if they are having a strong synchronization with the atom to whom the variable belongs (without having to pass it through connectors). That way, it would also be possible to automatically check if the variables can be accessed or not.
Still, this does not solve the problem if the variable has to "travel" across multiple atoms (e.g. the TC_active variable from TABLE 2), but could work for example with I2C_request and report.

Upgrade of all the tools with the BIP2 syntax. Unfortunately, at the time this project is created, most of the tools are designed for a previous BIP syntax version. It is possible to rewrite the model in the first BIP version, but this is a tedious and error prone procedure, especially for large models. Moreover some BIP2 features (above all, multiple port export) are not supported.

Embedded system friendly BIP. An embedded system here is intended not only as an application specific hardware implementation. It is energy, memory (program and RAM) and speed constrained. A typical example is the microcontroller used in this project (ARM Cortex-M3).
If it is desired to push BIP toward this kind of applications it has to be optimized. Not only in size, but also to support microcontroller and safety specific characteristics. For example: no dynamically allocated memory (especially for space applications), support for hardware interruptions, support for microcontroller sleep time. That means, the BIP engine frequency of execution can be (dynamically) modified to save energy or to meet impending deadlines. Other considerations are presented in appendix E.

Multi threaded real time engine. This is in relation with the previous point. Having a multi threaded real time engine is the best for this kind of embedded applications; one can verify and enforce timing requirements directly thanks to the model and WCET. Moreover, computation expensive tasks (such as the ADCS algorithm here) would not block other lighter but more frequent tasks.

Support for macro in BIP. This is especially a problem with guards. Suppose that a guard (on a "status" variable) can be equal to 4 values, each of them representing something. I.e. 0 –> failure, 1 –> success, 2 –> error, 3 –> finish. Imagine that those are also used by the C/C++ function that returns the value of "status". In C/C++ it is possible to write:

```
#define FAILURE 0
#define SUCCESS 1
#define ERROR 2
#define FINISH 3
```

But those can not be reused in BIP, even if the C/C++ source that defines those macros is included. This is an additional source of error (especially if changes must be made later in the project).

13 Future work

The next step to be performed is to evaluate the performance of BIP on small embedded systems such as the one presented in this project. It is important to verify if the BIP engine will still work after cross compilation for a specific target.
Once this has been proven, the most interesting task is to characterize the BIP engine performance. That means, study how the binary size increases with the addition of more components, study how long does an engine step take depending on the size of the model (atoms and connectors) and what is the impact on the RAM.
The BIP framework and its associated tools needs changes to be more developer friendly. If

the use of the BIP framework will be taken into consideration for future studies and projects in the EPFL, then an explicit collaboration between the EPFL RISD laboratory, the Space Center and the Verimag laboratory in Grenoble is strongly encouraged.
The most urgent developments of the framework are: upgrade of the real-time engine version, cross compilation of the engine for other platforms (an open source engine would be a great step forward in this direction) and some tools to debug the model in system (using GDB should be possible).
After some initial developments to make the framework more desirable to developers, others EPFL laboratories can be introduced in the loop, for example those working on robotics.

Another project would be to develop the C functions simulating the behavior of external hardware and then design all the C/C++ functions that would be used in the satellite. Appendix C can be used as a starting point. Integrating those functions in BIP would help validate the design process.

14 Conclusion

A BIP model of the software running on the CDMS of the CubETH satellite has been designed.
This is (probably, since some may exist, but may not be public) one of the first designs for an hardware comparable to the one of CubETH entirely based on the BIP framework. Therefore it was not possible to extensively use preexisting procedures to design the system. For that reason, some "classes" of components have been introduced to be reused for future BIP projects. Along with that, the whole procedure to design a system software with BIP is introduced in section 11.
The BIP design is verified thanks to a reasoning on the model following a set of rules. Those rules, if respected, ensure that no global, nor local, deadlock can be reached by the system.
Special care has been taken to avoid possible hardware conflicts by enforcing mutual exclusion. This has been achieved by modeling all the hardware resources as atoms and by clearly defining how those resources are connected to their users. It is quite natural to achieve this with BIP and it is easy to highlight those characteristics in the model. The tools to automatically check some properties of the model looks really promising, but must be upgraded to support the newer release of BIP.
The use of the BIP framework is encouraged for other projects. It is a rigorous and well defined technique to model and describe such complex systems. Nonetheless, before doing that, the framework should be improved in the ways mentioned in this report. A clear collaboration with the engine developers must be initiated to adapt the BIP tools to the targeted microcontroller and to this kind of embedded systems in general.

Special thanks to Anton Ivanov and Louis Masson for supervising tho project and making it possible. To Jacques Combaz of the Verimag laboratory (Grenoble) for the discussion about the integration of the BIP engine on the CDMS hardware.
Extra special thanks to Simon Bliudze of the RISD laboratory for all the valuables discussions on the BIP framework and its fundamental concepts, for the advices on some design choices and for the suggestions on the report redaction.

A Toolchain for EFM32GG880 code cross compilation

The BIP framework compilation already uses CMake, therefore it is reasonable to use it even for cross compilation.
For this purpose, a toolchain file is necessary. It is **Toolchain-EFM32GG.cmake**.
In this file, the compilers used to compile C, C++ and assembly code are specified. Those are:
C: arm-none-eabi-gcc-4.8.3
C++: arm-none-eabi-g++
ASM: arm-none-eabi-gcc-4.8.3
The "FORCE" option is used to avoid that the compiler is changed to the default one (can happen on some CMake versions) and to avoid the test of the compiler. Otherwise an error may occur caused by the differences between the embedded and x86 libraries.
The structure of the **project folder** is: project/src, project/bin. The bin directory is used to build the code. An example is provided in the "hello" directory. The content of the src directory is:

- file.c/cpp
- CMakeLists.txt

CMakeLists.txt contains all the flags for the compilers in order to successfully cross compile for the EFM32GG880 and the linker directives.
It is important to notice that, in the case where only plain C is compiled, the G++/efm32gg.ld file must be replaced with GCC/efm32gg.ld and G++/startup_efm32gg.s with GCC/startup_efm32gg.S.
The CUSTOM_COMMAND at the end of the file is needed to generate the binary executable that can be loaded on the microcontroller using the EnergyMicro/SiliconLabs tools [13].
All the specific paths to the files provided by EnergyMicro/SiliconLabs for their microcontrollers in the CMakeLists file must be replaced.
To build the project, this command has to be executed from the project/bin directory:

```
$ cmake -DCMAKE_TOOLCHAIN_FILE=/home/marco/bin/EFM32/Toolchain-EFM32GG.cmake
../src
$ make
```

Where -DCMAKE_TOOLCHAIN_FILE sets the path where the toolchain file is located.

A.1 Know issue

Depending on the CMake version, an error similar to this one may occur:

```
  -D=EFM32GG990F1024 -Wall -Wextra -mcpu=cortex-m3 -mthumb -DDEBUG_EFM\
 -o CMakeFiles/cplusplus.dir/local/jcombaz/EFM32/EFM32GG/Source/G++/\
startup_efm32gg.s.obj /local/jcombaz/EFM32/EFM32GG/Source/G++/startup_efm32gg\
.s /local/jcombaz/MentorGraphics/Sourcery_CodeBench_Lite_for_ARM_EABI\
/bin/../lib/gcc/arm-none-eabi/4.8.3/../../../../arm-none-eabi/bin/ld\
: warning: cannot find entry symbol _start; defaulting to 0000800c \
/tmp/ccCBC42H.o:(.cs3.interrupt_vector+0x0): undefined reference to\
 `__cs3_stack'
```

In that case in the build.make file in the cplusplus.dir directory you may have:

```
   /local/jcombaz/MentorGraphics/Sourcery_CodeBench_Lite_for_ARM_EABI\
/bin/arm-none-eabi-gcc-4.8.3 $(ASM_FLAGS)-o CMakeFiles/cplusplus.dir\
/local/jcombaz/EFM32/EFM32GG/Source/G++/startup_efm32gg.s.obj /local\
/jcombaz/EFM32/EFM32GG/Source/G++/startup_efm32gg.s
```

instead of:

```
   /local/jcombaz/MentorGraphics/Sourcery_CodeBench_Lite_for_ARM_EABI\
/bin/arm-none-eabi-gcc-4.8.3 $(ASM_FLAGS)-o CMakeFiles/cplusplus.dir\
/local/jcombaz/EFM32/EFM32GG/Source/G++/startup_efm32gg.s.obj -c /local\
/jcombaz/EFM32/EFM32GG/Source/G++/startup_efm32gg.s
```

Note the missing **-c**. The cause of this is unknown, but it must be added manually (every time).

B List of components

- **CCSDS** This compound implements the CCSDS state machine from the ECSS standards which decodes, checks and distributes telecommands to the satellite subsystems. It is composed of the following atoms:
  - **CCSDS_state_machine**
  - **APID_distributor**
  - **service_distributor**
  - **other_apid**
- **TC_receiver** This compound is in charge of receiving telecommands directly from the telecommunication subsystem or from a list according to ECSS service 11. Then it feeds them to the CCSDS state machine. It is composed of the following atoms:
  - **TC_fetch**
  - **TC_buffer**
  - **TC_list_switch**
- **CDMS_status** This atom manages the CDMS "heartbeat" and the reset of watchdogs.
- **s3_5** This atom provides service 3, sub service 5 according to the ECSS standard
- **s3_6** This atom provides service 3, sub service 6 according to the ECSS standard

- **s4_1** This atom provides service 4, sub service 1 according to the ECSS standard and generates telemetry according to service 4, sub service 2
- **s4_3** This atom provides service 4, sub service 3 according to the ECSS standard
- **s8** This atom provides service 8 according to the ECSS standard
- **s11** This atom provides service 11 according to the ECSS. The possible sub services are the ones defined in CubETH ICD [2] not represented by other atoms.
- **s11_switch** This atom provides services 11_1 and 11_2 according to the ECSS.
- **s11_7** This atom specifically provides service 11, sub service 17 according to the ECSS standard and generates telemetry according to service 11, sub service 13
- **s13** This atom provides service 13 according to the ECSS. The possible sub services are the ones defined in CubETH ICD [2]
- **s15_1** This atom provides service 15, sub service 1 according to the ECSS standard
- **s15_2** This atom provides service 15, sub service 2 according to the ECSS standard
- **s15_9** This atom provides service 15, sub service 9 according to the ECSS standard and generates telemetry according to service 15, sub service 8
- **s17_1** This atom provides service 17, sub service 1 according to the ECSS standard and generates telemetry according to service 17, sub service 2
- **ADCS_module** This compound is responsible for the communication with the ADCS subsystem, depending on its state it may compute the ADCS algorithm. It is composed of the following atoms:
  - **ADCS_routine**
  - **ADCS_mode**
- **Payload** This compound is responsible for the payload management and the execution of payload specific TC and TM. It is composed of the following atoms:
  - **s128_1**
  - **s128_4**
  - **s128_5**
  - **status**
  - **status_verification**
  - **data_transfer**
- **HK_internal** This compound manages the housekeeping of the devices directly connected to the microcontroller running BIP and of the microcontroller itself. It is composed of the following atoms:

  - **HK_internal_process**
  - **HK_switch**
  - **PS_switch**

- **HK_subsystem** This compound manages the housekeeping of the subsystems connected by a bus to the main microcontroller. There are 3 of them (one for every monitored subsystem). It is composed of the following atoms:
  - **HK_process**
  - **HK_switch**
  - **PS_switch**
  - **status**
- **I2C_sensor** This compound reads (and possibly preprocesses) sensors on the I2C_sens bus. There is one for every sensor on the bus. It is composed of the following atoms:
  - **Sens_process**
  - **Sens_switch**
  - **PS_switch**
  - **status**
- **I2C_sat** This atom models the I2C_sat resource. That means, the hardware specific part, protocol (defined in CubETH ICD [2]) and error management
- **I2C_sens** This atom models the I2C_sens resource. That means, the hardware specific part, and error management
- **flash_memory** This atom models the external flash memory resource. The hardware specific part and error management
- **error_log** This atom models the error_log memory region

C C functions to be developed for CubETH CDMS

For every atom/compound in section 7 specifics C functions must be developed. A list of them (in the same order as section 7) is given here, with a brief description and the global variables they use (from section 9.2).
A draft on the global variables structures is proposed in appendix C.21.

C.1 CDMS_status

resetWatchdog() shall implement code that resets both the internal and external watchdogs.
composeMessage() shall load the I2C_request structure with the EPS address and the CDMS status as message.

C.2 ADCS_module

compute() shall implement the stabilization algorithm. It uses the values in the sens1 and sens2 structures.
communicate() shall send the computed stabilization values and the mode (ekf) to the ADCS board through SPI.
setMode() shall send the Bdot mode to the ADCS board through SPI.

C.3 Payload

C.3.1 s128_1

composeMessage() shall take the content of the TC packet from the TC_active structure and copy it in the I2C_request structure with the payload board address and "scenario load" command.
decodeMessage() checks the I2C_report structure for a positive acknowledge and sets the BIP fail variable. It is 1 if a software error in the transmission or execution (in the payload) is recognized. Otherwise, it is 0.

C.3.2 s128_4

composeMessage() shall set the I2C_request structure with the payload board address and the code of a "start scenario".
decodeMessage() checks the I2C_report structure for a positive acknowledge and sets the BIP fail variable. It is 1 if a software error in the transmission or execution (in the payload) is recognized. Otherwise, it is 0.

C.3.3 s128_5

composeMessage() shall set the I2C_request structure with the payload board address and the code of an "abort".
decodeMessage() checks the I2C_report structure for a positive acknowledge and sets the BIP fail variable. It is 1 if a software error in the transmission or execution (in the payload) is recognized. Otherwise, it is 0.

C.3.4 staus_transfer

composeMessage() shall set the I2C_request structure with the payload board address and the code for "status request".
decodeMessage() checks the I2C_report structure to obtain the status of the payload board.

C.3.5 data_verification

composeMessage() shall set the I2C_request structure with the payload board address, the code for "memory request" and the ID of the packet it wants to acquire.
decodeMessage() checks the I2C_report structure for a positive memory received acknowledge and updates the packet ID to acquire the next one. If it was the last, it sets the finish

variable to 1. The memory region shall be saved in an internal (to this BIP module) buffer. If the reception failed, it updates the request to ask for the same memory packet ID.
updateSection() sets the write_buffer structure to store the received message from the internal buffer in the non volatile memory.

C.4 HK_subsystem

Please note that the functions changes slightly depending on the subsystem they address. For example, the software errors that the subsystem produce may be different one from each other, thus those functions have to be tuned accordingly.

C.4.1 Process

composeMessage() shall set the I2C_request structure with the subsystem board address and the "request housekeeping" code.
decodeMessage() checks the I2C_report structure for a positive reception of HK and checks if the housekeeping shows signs of internal problems (those needs to be defined by the developers of the subsystems firmwares). The housekeeping is stored in an internal buffer. The HK_report structure shall also be updated.
write_request() sets the write_buffer structure to store the received housekeeping from the internal buffer in the non volatile memory.
composeMessage() (to TTC) shall set the I2C_request structure with the COM board address and the housekeeping values to be downlinked to the ground station.

C.4.2 Status

composeMessage() shall set the I2C_request structure with the EPS board address, the code to request a reset and an identifier of the component to be resetted.

C.5 HK_internal

readSensors() This functions fetches all the internal housekeeping variables and loads them in an internal buffer structure. The HK_internal_report structure shall also be updated.
composeMessage() (to TTC) shall set the I2C_request structure with the COM board address and the housekeeping internal structure to be downlinked to the ground station.
write_request() sets the write_buffer structure to store the internal housekeeping from the internal buffer structure in the non volatile memory.

C.6 I2C_sensor

Please note that the functions will differ depending on the sensor they address.

C.6.1 Process

composeMessage() shall set the I2C_request structure (the sens one) with the sensor address and the address of the memory region it wants to read.

decodeMessage() checks the I2C_report structure (the sens one) for a positive reception. The report is stored in sens1 (or sens2) structure. At this step it is also possible to preprocess the sensor data.
write_request() sets the write_buffer structure to store the received sensor data in the non volatile memory.
composeMessage() (to TTC) shall set the I2C_request structure with the COM board address and the sensor data to be downlinked to the ground station.

C.6.2 Status

composeMessage() shall set the I2C_request structure with the EPS board address, the code to request a reset and an identifier of the component to be resetted.

C.7 TC_receiver

composeMessage() shall set the I2C_request structure with the COM board address and the "next telecommand" code.
checkTC() checks the I2C_report structure for a positive reception of a telecommand. If a telecommand has been successfully received, it loads the TC_active structure with the content of the I2C_report structure (conveniently modified if necessary) and sets the TC internal variable to 1. Otherwise the TC_active structure is not modified and TC is set to 0.
checkList() checks the internal TC list (which resides in the non volatile RAM). If there is a telecommand that, according to service 11 [3], can be executed, it is loaded in the TC_active structure and the internal TC variable is set to 1. Otherwise the TC_active structure is not modified and TC is set to 0.

C.8 CCSDS

C.8.1 CCSDS_state_machine

getAck() reads (from the TC_active structure) the status of the acknowledge variables and stores them internally.
check() Performs a check on the TC_active structure to ensure the validity of the TC as expressed in the ECSS standard [3]
composeMessage() shall compose the s1_1, s1_2, s1_7 and s1_8 TM reports and put it in the I2C_request structure according to the ECSS standard.

C.8.2 apid_distr

checkAPID() Reads the apid variable from the TC_active structure and stores it on an internal variable.

C.8.3 service_distr

checkService() Reads the service and subservice variables from the TC_active structure and stores them on internal variables.

C.8.4 other_apid

forwardMessage() shall set the I2C_request structure with the board address to who the apid is assigned and the TC packet content from the TC_active structure.
decodeMessage() checks the content of I2C_report to verify the result of the execution of the telecommand previously dispatched.

C.9 I2C_sat

masterWrite() is a function that performs an I2C write operation according to the I2C_request structure.
masterWrite(). The second time, according to the protocol [2], this function shall perform a write operation to prepare the following read one.
masterRead() reads data on the I2C slave according to the protocol described in the ICD. The data is put in I2C_report.
checkCRC() checks the validity of the message received in the I2C_report structure.
error. During every error transitions, the error circular array should be updated with the error that just occurred.

C.10 I2C_sens

I2C_op performs a sequence of a write and read I2C operation on the I2C_sens bus according to I2C_request (the one associated to the sens bus). The result is placed on the I2C_report structure (the one associated to the sens bus). If the message was not received correctly, it sets "msg" to ERROR.

C.11 Flash memory management

updateLocation() shall determine where the data contained in the write_buffer structure will be written. This will depends on how the memory will be organized. The write_buffer structure shall have a variable determining the source of the packet to be written. Moreover, the location address must be changed after every page buffer of the memory that is written (here is 64 bytes at a time).
write() performs the hardware access and write process of maximum 64 bytes to the external flash memory.
checkStatus() contacts the memory device and return its status, that is: internal write, finish, error or timeout.
bad_crc(), failure(). During every error transitions, the error circular array should be updated with the error that just occurred. Operations to recover the memory should also be performed here (soft and hard resets, erase procedure, ..).
read() reads the memory regions specified in the read_buffer structure. The latter is filled with what is read.
checkCRC() verifies the validity of the CRC from the read packet.

C.12 s3_5,s3_6,s15_1,s15_2

getVal() extracts the value of the interaction to trigger from the TC_active data packet according to the ECSS standard [3]. Returns EMPTY when the packet is completely read.

C.13 s4_1

compose_TM() according to service 4, takes all the averages, minimum and maximum of the report parameters (e.g. from structure sens1) and puts them on I2C_request according to subservice 4_2 [3]. The address is the one of the COM subsystem.

C.14 s4_3

clear() shall erase average, minimum and maximum from the report parameters (e.g. from structure sense)

C.15 s8

getVal extracts the value of the interaction to trigger depending on the TC_active data packet according to the ECSS standard [3] and ICD [2]. N.B. Those are TBD in the ICD.

C.16 s11

There is a function associated with every subsystem transition (even if it is not explicitly marked in the graphical model). Those perform a function on the TC_schedule structure using the TC_active structure and according to the ECSS standard [3].

C.17 s11_17

compose_TM() according to service 11. The TC_schedule structure is put in I2C_request according to TM subservice 11_13. The address is the one of the COM subsystem.

C.18 s13

setWrite() shall take the content of the TC_active structure packet and place it in the write_buffer structure.
13_16() shall set the I2C_request structure with the COM board address and the content according to ECSS.
13_14() shall set the I2C_request structure with the COM board address and the content according to ECSS.

C.19 s15_9

setReadParam() shall set the parameters of read_buffer according to the packets in the packet store to be read. Note that the packet are supposed to be time ordered from older to newer.

checkDate() shall verify the date of the packet in read_buffer. If the date is within the selected boundaries it shall place the content of the read_buffer in an internal buffer. The date can be marked as OK if it is within the boundaries, NOT_OK if it is before the boundaries and END if it is after the boundaries.
composeMessage() shall place the content of the internal buffer in the I2C_report structure, according to TM subservice 15_8. The address is the one of the COM subsystem.

C.20 17_1

compose_TM() according to service 17, create an I2C_request for the COM subsystem according to subservice 17_2 [3].

C.21 Global variables

This is a possible composition of the structures of the global variables presented in section 9.2.

C.21.1 I2C_request and I2C_report

This description is valid both for the I2C_sat and I2C_sens busses.

I2C_request

- **uint8_t** I2C_request_message[I2C_SAT_MSG_MAX_SIZE];
- **uint16_t** I2C_requestSize;
- **uint8_t** I2C_address;

I2C_report

- **uint8_t** I2C_report_message[I2C_SAT_MSG_MAX_SIZE];
- **uint16_t** I2C_reportSize;
- **uint8_t** I2C_address;

It is also possible to directly use the structure defined for an I2C transfer by the EnergyMicro embedded libraries. See **I2C_TransferSeq_TypeDef**.

C.21.2 hw_error

Must be defined as a circular array. It shall contain the error type and a timestamp.

C.21.3 TC_active

The structure proposed in [14] can be reused.

C.21.4 write_buffer and read_buffer

Data location depends on how the data is organized. It may be an ID, if a file system is used. It can also be a packet store ID if the data is stored in predefined memory sections depending on its origin.

write_buffer

- **uint8_t** buffer[MAX_WRITE_BUFFER_SIZE];
- **uint16_t** buffer_size;
- **uint16_t** dataLocation;
- **uint16_t** crc;

read_buffer

- **uint8_t** buffer[MAX_READ_BUFFER_SIZE];
- **uint16_t** read_request_size;
- **uint16_t** dataLocation;
- **uint16_t** crc;

C.21.5 TC_schedule

The structure proposed in [14] can be reused.

C.21.6 HK_generic_report

Here, N_OF_PARAMETERS is how many housekeeping parameters a structure contains.

- **uint16_t** max[N_OF_PARAMETERS];
- **uint16_t** min[N_OF_PARAMETERS];
- **uint16_t** mean[N_OF_PARAMETERS];
- **uint16_t** n_samples[N_OF_PARAMETERS]; (uint32_t, depending on data rate)

C.21.7 sens1 and sens2

Here, N_OF_VALUES is how many different data the sensor produces (e.g. rotation on x, y and z axis are three separated values).

- **uint16_t** current_value[N_OF_VALUES];
- **uint16_t** max[N_OF_VALUES];
- **uint16_t** min[N_OF_VALUES];
- **uint16_t** mean[N_OF_VALUES];
- **uint16_t** n_samples[N_OF_VALUES]; (uint32_t, depending on data rate)

D Non volatile memory considerations

There are two kinds of non volatile memory on the CDMS. A NOR flash and a MRAM. The MRAM device can be used as an external RAM as it is completely transparent to the microcontroller. On the other hand, the NOR flash device must write a buffer of 64 bytes each time and the memory must be erased before a new write operation.
Therefore it is reasonable that, for speed considerations, all the data actively used by the satellite is stored in RAM. The NOR flash will only store packets to be later downlinked to the ground station (one write, one read).
The MRAM device should be more reliable than the NOR flash in the space environment. Nonetheless, the external MRAM is slower and power hungrier than the internal RAM, therefore it makes sense to store there only data which are important in case of an unexpected CDMS power down.
A list of those variable is given in section D.1.

D.1 Data in MRAM

In this external non volatile memory the following data must be stored to provide a successful start up (described in section E) after an unexpected reset:

- TC_active
- TC_schedule
- hw_error
- internal time
- modes and statuses

"Modes and statuses" stores the place where the BIP atoms representing mode and status were before the shutdown. That way, the BIP model can restart exactly where he left (an example is given in appendix E).
Possibly, maximums, minimums and average of the various report parameters can also be stored there.
Another possibility, for extra redundancy, is to save all those variables in three places in the MRAM and performing voting during startup to decrease data corruption probability. It is also possible to periodically backup those values in the NOR flash memory.

E Recovering from CDMS shut down and startup sequence

During the beginning of the BIP model execution, all the "initial transitions" are executed. Those are the transitions leading to the starting place of every atom. During them, it is possible to set all the parameters needed to prepare the hardware and to initialize all the variables. For example, setting up the external memories, the I2C busses, GPIO functions and so on.
Nonetheless on an embedded system where no operating system is used (the microcontroller is programmed "bare metal" here), it is better to set up some things even before launching the BIP engine. For example, clock options and watchdog must be set as early as possible, ideally right after the program startup. Therefore, it may be useful to add the possibility, in the BIP framework, to run some custom code before initializing and launching the engine and the model.

E.1 Shut down recovery

Since some non volatile memories are available on the CDMS, it is possible to store internal variables (see appendix D) to restart operations just as they where before the shutdown.
From a BIP model point of view it is important to know in which place all the "status" and "mode" atoms where before the shutdown. To accomplish this, the solution in figure 42 is proposed.

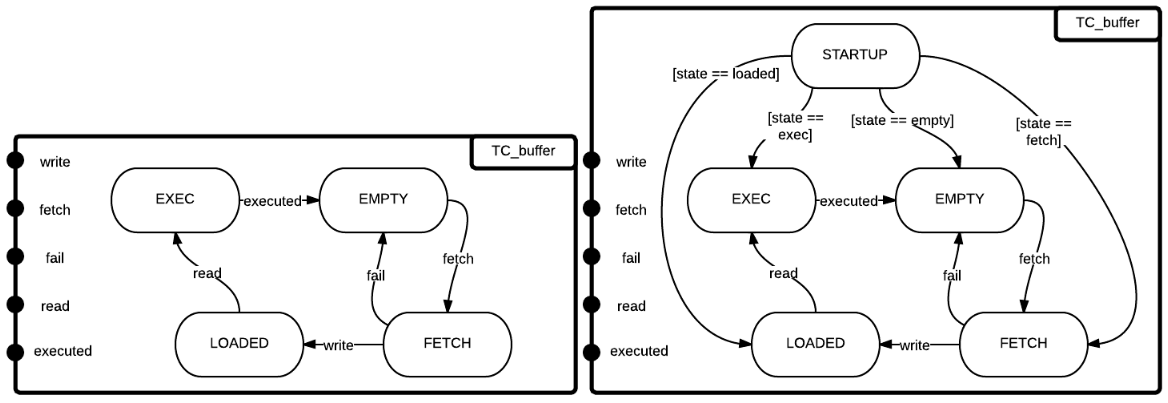


Figure 42: *Left, TC_buffer "status" atom. Right, modified TC_buffer status atom with STARTUP as initial state*

During the initial transition leading to STARTUP, the "state" variable would be extracted from the non volatile memory. That way, this compound can restart from where it left. The same procedure may be synchronized with triggered activities. For example, the connector leading from STARTUP to EXEC should be synchronized with the "CCSDS state machine" triggered activity to start the processing of the telecommand. Forgetting this can be another source of deadlocks.
It is obvious that a custom telecommand clearing the non volatile memory to allow a "clean" startup shall be introduced. This will be useful if the non volatile memory is corrupted and some activities are no more performing correctly.
Another option (more safe) is to hardwire a connection to the EPS to determine if the startup must be "clean" or not. In the first case, the BIP initial transitions would ignore the non volatile memory. In the second case it would be used. That way, if a startup with corrupted memory screws up the model completely, it is possible to place the satellite in safe mode (that means give control to the EPS), set the wired connection between it and the CDMS to "do not use non volatile memory" and only then restart the CDMS.
This scenario is plausible if the EPS detect that the CDMS is having too frequents crashes/deadlocks.

F Memory redundancy

In figure 43 a "drop-in" replacement for the flash_memory_module (section 7.11) is given. This new module can simply replace the flash_memory_module and allows the use of multiple memory devices. No changes in the external connectors and interfaces are required. The "to_main", "to_backup" and "to_emergency" ports are used to change the active memory. They can synchronize either with a custom telecommand or directly with an activity. The emergency atom associated with the "emergency" transition is the last resource for the satellite in case where every other memory fails. The C/C++ functions in this atom shall ignore the write and memory requests which are not essential. For example, it may store only a subset of the payload data and ignore all the rest (by not doing anything but a success transition).

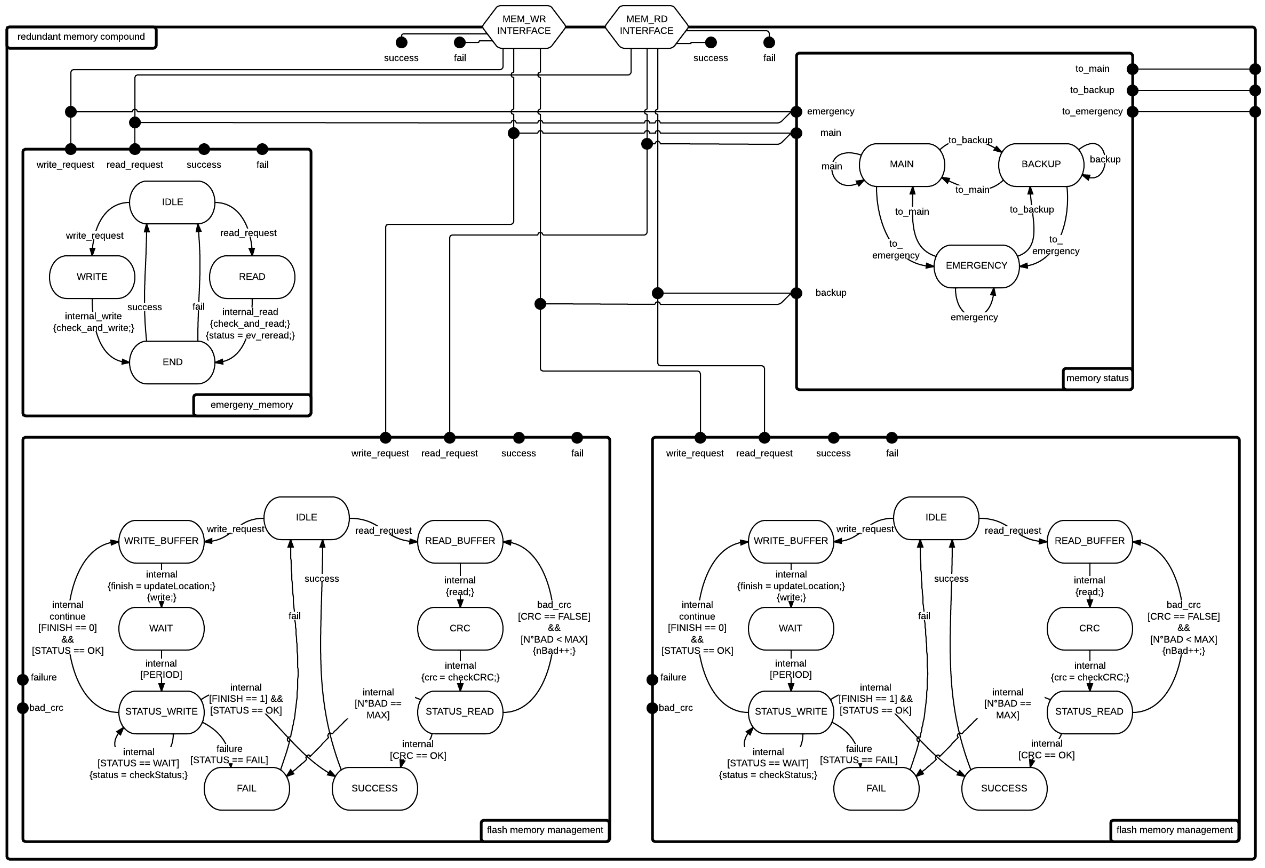


Figure 43: *BIP compound to manage multiple memory devices*

G BIP code structure and use

G.1 Time adaptation

The true BIP model has been adapted in order to be used in "pseudo real time". That means, an atom keeping track of the time has been added (timer.bip). Moreover for every transition where a time guard needs to be evaluated, a transition going from that place to itself and updating the deltaT since the last execution is introduced. This is not a good design practice, but it is used to have a meaningful code demonstration. Eventually, the real-time BIP engine should be used (when it will be upgraded).
Remember that all the time guards are represented in the model as: [PERIOD].

G.2 Content

The directory containing the project is called **ecssModel**. All the .bip files are in there. Each file represent either an activity compound or an ECSS service, with the exception of:
ports.bip: contains ports declarations.
connectors.bip: contains connectors declarations
master.bip: is the file where all the component are put together to form the complete BIP model.
The **extern_c** folder contains the externals C/C++ sources and headers files.
The output folder contains the C++ files and CMake instructions generated by the BIP parser. Inside this directory there is the build directory, containing the executable; **system**, the list of telecommands; **tc_list.txt** and the list of errors occurred during the last execution **errors.txt**.

G.3 How to use the BIP model

G.3.1 Building the BIP model

Remember that the BIP tools must be installed according to [7]. From the **ecssModel** directory, in the terminal, call:

```
#bipc.sh -I . -p master -d "master()" --gencpp-output output --gencpp\
-cc-I $PWD/extern_c --gencpp-follow-used-packages
```

master is the name of both the file and the compound where the whole model is described.
- -gencpp-output output means that the BIP model will be parsed in C++ .
- -gencpp-cc-I $PWD/extern_c is used to include sources and headers contained in the extern_c directory.
- -gencpp-follow-used-packages is necessary because the BIP model is spread across various documents. This line tells this fact to the BIP parser which is then able to patch everything together.

G.3.2 Compile

From the build directory inside of the output directory simply type (in the terminal) :

```
#cmake ..
#make
```

Please note that if the BIP code is changed, the step in section G.3.1 must be repeated. On the other hand, if just the C/C++ code in extern_c is changed, one can just hit "make" again.

G.3.3 Example

From the folder containing the BIP model:

```
#mkdir -p output/build
#bipc.sh -I . -p master -d "master()" --gencpp-output output --gencpp\
-cc-I $PWD/extern_c --gencpp-follow-used-packages
#cd output/build
#cmake ..
#make
#./system
```

G.3.4 Use the model

The model can run by simply typing:

```
#./system
```

from the build directory, in the terminal. Calling:

```
#./system -i
```

Allows a "step by step" execution of the model.
In order to be aware of what is happening in the model, a "pseudo real time" (as explained in section G.1) mode has been developed. Regularly, printf functions are called to show what the program is doing. Some keywords allows to select only the printf on which one is interested. In order to do that, follow TABLE 3.
Moreover, a tc_list.txt file is present in the built directory. Here one can write telecommands to be executed by the model. The available telecommands are the one described in this project. The format is :

apid,service,subservice,reception_acknowledge(1 or 0),end_acknowledge(1 or 0),option

Example:

1,17,1,1,1,0
1,128,1,1,1,0

An errors.txt file is generated in the build directory to show all the (simulated) hardware errors that have happened during the execution of the model.

Table 3: Keywords used to highlight only some processes of the BIP model. The description column explains what they represents. They can be used by typing: $ ./system | grep -i -e 'keyword1' -e 'keyword2' -e ...

Keyword	**Description**
I2C_satprint	Display normal messages coming from the I2C_sat peripheral
I2C_satcurrentprint	Display detailed internal progress of the I2C_sat peripheral
I2C_sensprint	Display normal messages coming from the I2C_sens peripheral
I2C_senscurrentprint	Display detailed internal progress of the I2C_sens peripheral
flashprint	Display normal messages coming from the external memory compound
flashcurrentprint	Display detailed internal progress external memory
ADCSprint	Display the events in the ADCS algorithm compound
CDMSprint	Display the CDMS heartbeat
HK_COMprint	Display the COM subsystem status HK
HK_EPSprint	Display the EPS subsystem status HK
HK_PLprint	Display the PL subsystem status HK
HK_internalprint	Display the CDMS internal status HK
ECSSprint	Display the progress in the telecommand handling
tcreceiverprint	Display the progress in the telecommand reception
payloadprint	Display the events in the payload board
sens1print	Display the operations of sensor1
sens2print	Display the operations of sensor2
TMprint	Display telemetry messages
TCprint	Display telecommand related messages, i.e. when a TC begins execution
service13print	Display progress of service 13
service15print	Display progress of service 15

References

[1] L. Masson *CDMS for CubETH* 2013

[2] CH-B-ICD-EPFL-0-2-CubETH ICD

[3] ECSS-E-70-41A

[4] Sara C. Spangelo, David Kaslow, Chris Delp, Bjorn Cole, Louise Anderson, Elyse Fosse, Brett Sam Gilbert, Leo Hartman, Theodore Kahn, James Cutler *Applying Model Based Systems Engineering (MBSE) to a Standard CubeSat*, 2012

[5] Samuel F. Hishmeh, Tyler J. Doering, James E. Lumpp, Jr., *Design of Flight Software for the KySat CubeSat Bus*, 2008

[6] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis, Verimag Laboratory, *Rigorous Component-Based System Design Using the BIP Framework* Verimag laboratory 2011

[7] http://www-verimag.imag.fr/TOOLS/DCS/bip/doc/latest/html/language.html

[8] Saddek Bensalem, Lavindrade Silva, Felix Ingrand and Rongjie Yan, *A Verifiable and Correct by Construction Controller for Robot Functional Levels*, 2011

[9] Ananda Basu, Matthieu Gallien, Charles Lesire, Thanh-Hung Nguyen, Saddek Bensalem, Felix Ingrand and Joseph Sifakis, *Incremental Component-Based Construction and Verification of a Robotic System* 2011

[10] Tesnim Abdellatif, Jacques Combaz and Joseph Sifakis, *Model-Based Implementation of Real-Time Applications*, 2010

[11] Sebastien Bornot, Joseph Sifakis and Stavros Tripakis, *Modeling Urgency in timed systems*, 1998

[12] https://launchpad.net/gcc-arm-embedded

[13] http://www.silabs.com/products/mcu/pages/simplicity-studio.aspx

[14] G. LePivain *CubETH: Software design of the CDMS* 2013