ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Architecture-based Design: A Satellite On-Board Software Case Study

EPFL IC IIF RiSD Technical Report
EPFL-REPORT-221156
http://infoscience.epfl.ch/record/221156

Anastasia Mavridou, Emmanouela Stachtiari, Simon Bliudze, Anton Ivanov, Panagiotis Katsaros, Joseph Sifakis

October 28, 2016

Abstract: In this case study, we apply the architecture-based design approach to the control software of the CubETH satellite. Architectures are a means for ensuring global coordination properties and thus, achieving correctness of complex systems by construction. The design approach comprises three main steps: 1) definition of a domain-specific taxonomy of architecture styles; 2) design of the software model by applying architectures to enforce the required properties; 3) deadlock-freedom analysis of the resulting model. We provide a taxonomy of architecture styles for satellite on-board software, formally defined by architecture diagrams in the BIP component-based framework. We show how architectures are instantiated from the diagrams and applied to a set of atomic components. Deadlock-freedom of the resulting model is verified using the DFinder tool from the BIP tool-set. Finally, we provide additional validation of our approach by using the nuXmv model checker to verify that the properties enforced by the architectures are, indeed, satisfied in the resulting software model.

```
@TechReport{EPFL-REPORT-221156,
   affiliation = {EPFL},
   author      = {Mavridou, Anastasia and Stachtiari, Emmanouela and
                  Bliudze, Simon and Ivanov, Anton and Katsaros, Panagiotis
                  and Sifakis, Joseph},
   details     = {http://infoscience.epfl.ch/record/221156},
   oai-id      = {oai:infoscience.epfl.ch:221156},
   oai-set     = {report},
   submitter   = {221727},
   title       = {Architecture-based {D}esign: {A} {S}atellite
                  {O}n-{B}oard {S}oftware {C}ase {S}tudy},
   unit        = {RISD},
   year        = 2016
}
```

Contents

1 Introduction

Satellites and other complex systems become increasingly software-dependent. Even nanosatellites have complexity that can be compared to scientific instruments launched to Mars. Standards exist for hardware parts and designs, and they can be found as commercial off the shelf (COTS) components. On the contrary, software has to be adapted to the payload and, consequently, hardware architecture selected for the satellite. There is not a rigorous and robust way to design software for CubeSats[1] or small satellites yet.

Flight software safety is of paramount importance for satellites. In harsh radiation environments, performance of COTS components is often affected by proton particles. For example, the I2C bus, which is commonly used in CubeSats due to its low energy consumption and wide availability in COTS chips, is well known in space community for its glitches. Although error correcting algorithms are widely implemented across all subsystems and interfaces, the use of the bus by the components requires careful coordination to ensure correct operation. Needless to say, software correctness must be established before launch.

To the best of our knowledge, most flight software for university satellites is written in C or C++, without any architectural thinking. A notable exception is a recent effort at Vermont Tech to use SPARK, a variant of Ada amenable to static analysis [14]. Other projects simply structure their code in C/C++ and then extensively test it, maybe using some analysis tools such as `lint` [27]. Others use SysML [34] to describe the system as a whole [33] 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 verification and validation of software behaviour.

Satellite on-board software and, more generally, all modern software systems are inherently concurrent. They consist of components that—at least on the conceptual level—run simultaneously and share access to resources provided by the execution platform. Embedded control software in various domains commonly comprises, in addition to components responsible for taking the control decisions, a set of components driving the operation of sensing and actuation devices. These components interact through buses, shared memories and message buffers, leading to resource contention and potential deadlocks compromising mission- and safety-critical operations.

The intrinsic concurrent nature of such interactions is the root cause of the sheer complexity of the resulting software. Indeed, in order to analyse the behaviour of such a software system, one has to consider all possible interleavings of the operations executed by its components. Thus, the complexity of software systems is exponential in the number of their components, making a posteriori verification of their correctness practically infeasible. An alternative approach consists in ensuring correctness by construction, through the application of well-defined design principles [4, 20], imposing behavioural contracts on individual components [8] or by applying automatic transformations to obtain executable code from formally defined high-level models [32].

Following this latter approach, a notion of *architectures* was proposed in [2] to formalise design patterns for the coordination of concurrent components. Architectures provide means for ensuring correctness by construction by enforcing global properties characterising the coordination between components. An architecture can be defined as an operator $\mathcal{A}$ that, applied to a set of components $\mathcal{B}$, builds a composite component $\mathcal{A}(\mathcal{B})$ meeting a characteristic property Φ. Composability is based on an associative, commutative and idempotent architecture composition operator $\oplus$: architecture composition preserves the safety properties enforced by the individual architectures. *Architecture styles* [23, 25] are families of architectures sharing common characteristics such as the type of the involved components and the characteristic properties they enforce. Architecture styles define all architectures for an arbitrary set of components that satisfy some minimal assumptions on their interfaces.

[1]CubeSat [15] is a standard for the design of nano- and picosatellites.

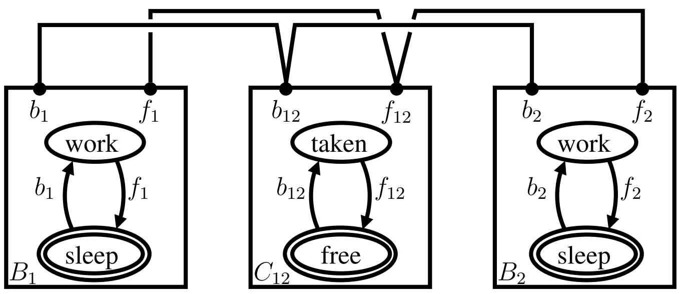


Figure 1: Mutual exclusion model in BIP

The notion of architectures proposed in [2] is based on the Behaviour-Interaction-Priority (BIP) [6] framework for the component-based design of concurrent software and systems. BIP is supported by a tool-set comprising translators from various programming models into BIP, source-to-source transformers as well as compilers for generating code executable by dedicated engines. Furthermore, the BIP tool-set provides tools for deadlock detection [7], state reachability analysis and an interface with the `nuXmv` model checker [10]. In the CubETH project [31], BIP was used to design logic for the operation of a satellite, executed on the on-board computer [29]. Although some properties were shown a posteriori to hold by construction, due to the use of a high-level modelling language instead of plain C/C++ code, the BIP model was designed in an ad-hoc manner, without consideration for any particular set of requirements.

In the case study presented in this technical report, we have analysed the BIP model obtained in [29] and identified a number of recurring patterns, which we formalised as architecture styles. We have identified a representative sub-system of the CubETH control software, which has a complete set of functional requirements, and redesigned from scratch the corresponding BIP model using the architecture styles to discharge these requirements *by construction*. We have used the DFinder tool to verify that the resulting model is free from deadlocks. Finally, we provide additional validation of our approach by using the `nuXmv` model checker to verify that the architectures applied in the design process do, indeed, enforce the required properties.

The rest of the technical report is structured as follows. Section 2 presents a brief overview of BIP and the architecture-based design approach. Section 3 presents the case study, the 9 identified architecture styles, illustrates our approach through the design of a corresponding BIP model and presents the complete list of functional requirements and CTL properties, the verification process and results. Section 4 discusses the related work. Section 5 concludes the report.

2 Architecture-based design approach

Our approach relies on the BIP framework [6] for component-based design of correct-by-construction applications. BIP provides a simple, but powerful mechanism for the coordination of concurrent components by superposing three layers. First, component *behaviour* is described by Labelled Transition Systems (LTS) having transitions labelled with *ports*. Ports form the interface of a component and are used to define its interactions with other components. Second, *interaction models*, i.e. sets of interactions, define the component coordination. Interactions are sets of ports that define allowed synchronisations between components. An interaction model is defined in a structured manner by using connectors [9]. Third, *priorities* are used to impose scheduling constraints and to resolve conflicts when multiple interactions are enabled simultaneously.

Figure 1 shows a simple BIP model for mutual exclusion between two tasks. It has two components B_1, B_2 modelling the tasks and one coordinator component C_{12}. Initial states of

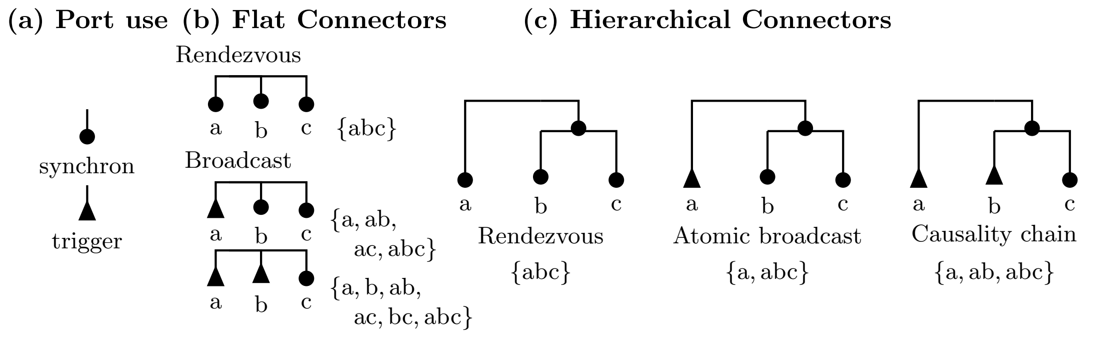


Figure 2: Flat and hierarchical BIP connectors

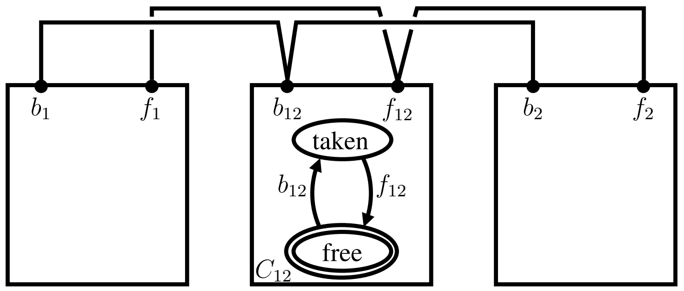


Figure 3: Mutual exclusion architecture

the components are shown with double lines. The four binary connectors synchronise each of the actions b_1, b_2 (resp. f_1, f_2) of the tasks with the action b_{12} (resp. f_{12}) of the coordinator.

Connectors define sets of interactions based on the synchronisation attributes of the connected ports, which may be either *trigger* or *synchron* (Fig. 2a). If all connected ports are synchrons, then synchronisation is by *rendezvous*, i.e. the defined interaction may be executed only if all the connected components allow the transitions of those ports (Fig. 2b), If a connector has at least one trigger, the synchronisation is by *broadcast*, i.e. the allowed interactions are all non-empty subsets of the connected ports comprising at least one of the trigger ports (Fig. 2b). More complex connectors can be built hierarchically (Fig. 2c).

An architecture can be viewed as a BIP model, where some of the atomic components are considered as *coordinators*, while the rest are *parameters*. When an architecture is applied to a set of components, these components are used as *operands* to replace the parameters of the architecture. Clearly, operand components must refine the corresponding parameter ones—in that sense, parameter components can be considered as *types*.[2] Figure 3 shows an architecture that enforces the mutual exclusion property $\texttt{AG}\neg(cs_1 \wedge cs_2)$ on any two components with interfaces $\{b_1, f_1\}$ and $\{b_2, f_2\}$, satisfying the CTL formula $\texttt{AG}\big(f_i \rightarrow \texttt{A}[\neg cs_i \ \texttt{W}\ b_i]\big)$, where cs_i is an atomic predicate, true when the component is in the critical section (e.g. in the state `work`, for B_1, B_2 of Fig. 1). Composition of architectures is based on an associative, commutative and idempotent architecture composition operator '$\oplus$' [2]. If two architectures $\mathcal{A}_1$ and $\mathcal{A}_2$ enforce respectively safety properties Φ_1 and Φ_2, the composed architecture $\mathcal{A}_1 \oplus \mathcal{A}_2$ enforces the property $\Phi_1 \wedge \Phi_2$, that is both properties are preserved by architecture composition.

Although the architecture in Fig. 3 can only be applied to a set of precisely two components, it

[2]The precise definition of the refinement relation is beyond the scope of this technical report.

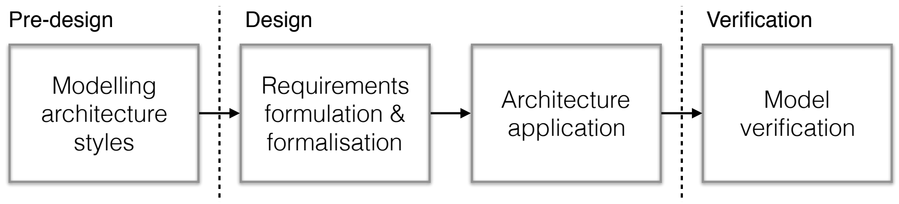


Figure 4: Architecture-based design flow

is clear that an architecture of the same *style*—with n parameter components and $2n$ connectors—could be applied to any set of operand components satisfying the above CTL formula. We use *architecture diagrams* [25] to specify such *architecture styles*, as described in the next section. (See Fig. 6 in Sect. 3.1.1 for the diagram of the style generalising the architecture in Fig. 3.)

The architecture-based design approach consists of the three stages illustrated in Fig. 4. First, architecture styles relevant for the application domain—in our case, nano- and picosatellite on-board software—are identified and formally modelled. Ideally, this stage is only realised once for each application domain. The remaining stages are applied for each system to be designed. In the second, design stage, requirements to be satisfied by the system are analysed and formalised, atomic components realising the basic functionality of the system are designed (components previously designed for other systems can be reused) and used as operands for the application of architectures instantiated from the styles defined in the first stage. The choice of the architectures to apply is driven by the requirements identified in the second stage. Finally, the resulting system is checked for deadlock-freedom. Properties, which are not enforced by construction through architecture application, must be verified a posteriori. In this case study, we illustrate all steps of this process, except the requirement formalisation.

In the first stage, we use *architecture diagrams* [25] to model the architecture styles identified in the case study. An architecture diagram consists of a set of *component types*, with associated cardinality constraints representing the expected number of instances of each component type and a set of *connector motifs*. Connector motifs, which define sets of BIP connectors, are non-empty sets of *port types*, each labelled as either a trigger or a synchron. Each port type has a *cardinality* constraint representing the expected number of port instances per component instance and two additional constraints: *multiplicity* and *degree*, represented as a pair $m : d$. Multiplicity constrains the number of instances of the port type that must participate in a connector defined by the motif; degree constrains the number of connectors attached to any instance of the port type.

Cardinalities, multiplicities and degrees are either natural numbers or intervals. The interval attributes, '`mc`' (multiple choice) or '`sc`' (single choice), specify whether these constraints are uniformly applied or not. Let us consider, a port type p with associated intervals defining its multiplicity and degree. We write '$\mathtt{sc}[x, y]$' to mean that the same multiplicity or degree is applied to each port instance of p. We write '$\mathtt{mc}[x, y]$' to mean that different multiplicities or degrees can be applied to different port instances of p, provided they lie in the interval.

For the specification of behavioural properties enforced by architecture styles, as well as those assumed for the parameter components, we use the Computation Tree Logic (CTL). We only provide a brief overview, referring the reader to the classical textbook [3] for a complete and formal presentation. CTL formulas specify properties of execution trees generated by LTSs. The formulas are built from atomic predicates on the states of the LTS, using the several operators, such as `EX`, `AX`, `EF`, `AF`, `EG`, `AG` (unary) and $\mathtt{E}[\cdot\,\mathtt{U}\,\cdot]$, $\mathtt{A}[\cdot\,\mathtt{U}\,\cdot]$, $\mathtt{E}[\cdot\,\mathtt{W}\,\cdot]$, $\mathtt{A}[\cdot\,\mathtt{W}\,\cdot]$ (binary). Each operator consists of a quantifier on the branches of the tree and a temporal modality, which together define when in the execution the operand sub-formulas must hold. The intuition behind the letters is the following: the branch quantifiers are `A` (for "All") and `E` (for "Exists"); the temporal modalities

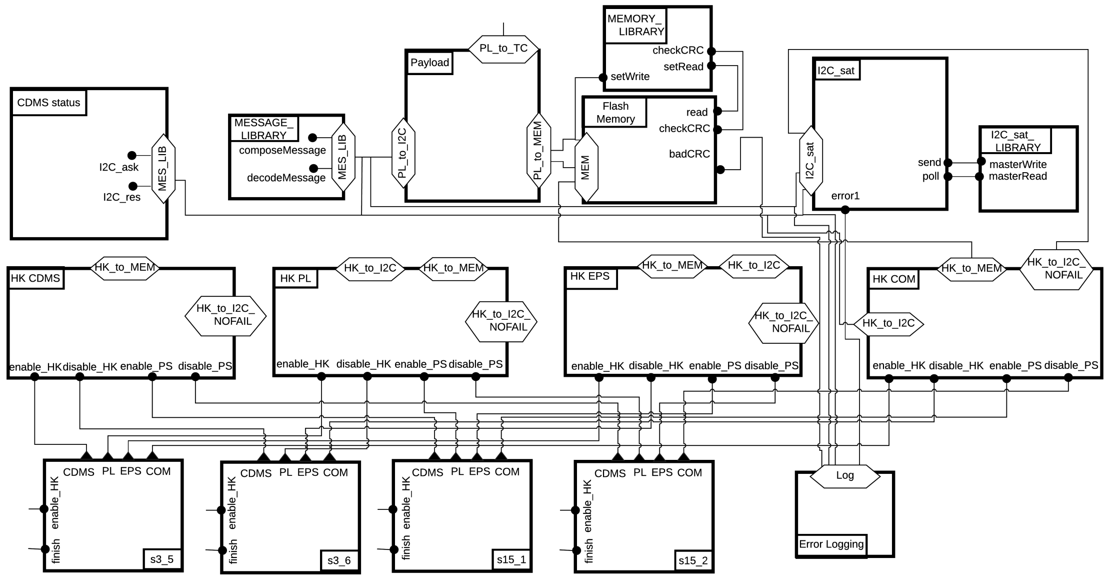


Figure 5: The high-level interaction model

are `X` (for "neXt"), `F` (for "some time in the Future"), `G` (for "Globally"), `U` (for "Until") and `W` (for "Weak until"). A property is satisfied if it holds in the initial state of the LTS. For instance, the formula $\mathtt{A}[p\,\mathtt{W}\,q]$ specifies that in *all execution branches* the predicate p must hold *up to the first state* (not including this latter), where the predicate q holds. Since we used the weak until operator `W`, if q never holds, p must hold forever. As soon as q holds in one state of an execution branch, p need not hold any more, even if q does not hold. On the contrary, the formula $\mathtt{AG}\,\mathtt{A}[p\,\mathtt{W}\,q]$ specifies that the subformula $\mathtt{A}[p\,\mathtt{W}\,q]$ must hold in *all branches at all times.* Thus, p must hold whenever q does not hold, i.e. $\mathtt{AG}\,\mathtt{A}[p\,\mathtt{W}\,q] = \mathtt{AG}\,(p \vee q)$.

3 Case study

CubETH is a nanosatellite based on the CubeSat standard [15]. It contains 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.

This case study is focused on the software running on the `CDMS` subsystem and in particular on the following subcomponents of `CDMS`: 1) `CDMS status` that is in charge of resetting internal and external watchdogs; 2) `Payload` that is in charge of payload operations; 3) three `Housekeeping` components that are used to recover engineering data from the `EPS`, `PL` and `COM` subsystems; 4) `CDMS Housekeeping` which is internal to the `CDMS`; 5) `I2C_sat` that implements the I^2C protocol; 6) `Flash memory management` that implements a non-volatile flash memory and its write-read protocol; 7) the `s3_5`, `s3_6`, `s15_1` and `s15_2` services that are in charge of the activation or deactivation of the housekeeping component actions; 8) `Error Logging` that implements a RAM region that is accessible by many users and 9) the `MESSAGE_LIBRARY`, `MEMORY_LIBRARY` and `I2C_sat_LIBRARY` components that contain auxiliary C/C++ functions.

A high-level BIP model of the case-study is shown in Fig. 5. For the sake of simplicity, we omit some of the connectors. In particular, we show the connectors involving the `HK_to_MEM`, `HK_to_I2C`

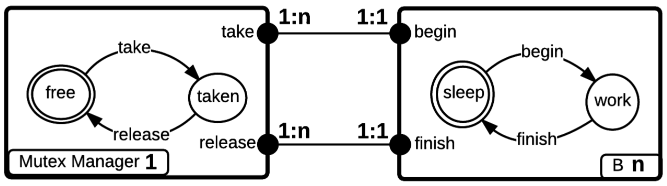


Figure 6: Architecture diagram of the Mutual exclusion style

and `HK_to_I2C_NOFAIL` interfaces of the `HK_COM` subsystem, but we omit the respective connectors involving the other three Housekeeping subsystems. The `MESSAGE_LIBRARY`, `MEMORY_LIBRARY`, `I2C_sat_LIBRARY`, `s3_5`, `s3_6`, `s15_1` and `s15_2` components are atomic. The rest are composite components, i.e. *compounds*.

The full BIP model of the case study comprises 22 operand components and 27 architectures that were generated from the architecture styles presented in the next subsection.

3.1 A taxonomy of architecture styles for on-board software

Since the identified architecture styles represent recurring patterns of satellite on-board software, the usage of the presented taxonomy is not limited to this case study. The identified styles can also be used for the design and development of other satellite on-board systems.

For each architecture style, we have studied two groups of properties: 1) *assumed properties* that the operand components must satisfy so that the architecture can be successfully applied on them and 2) *characteristic properties* that are properties the architecture imposes on the system. In the CubETH case study all characteristic properties are safety properties.

We use simple and interval architecture diagrams to describe the 9 architecture styles. For the sake of simplicity of the presentation, in the next subsections, we omit the cardinality of a port type if it is equal to 1. The cardinality of a component type is indicated right next to its name.

3.1.1 Mutual exclusion style

The Mutual exclusion architecture style enforces mutual exclusion on a shared resource. The unique—due to the cardinality being 1—coordinator component, `Mutex manager`, manages the shared resource, while n parameter components of type `B` can access it. The multiplicities of all port types are 1 and therefore, all connectors are binary. The degree constraints require that each port instance of a component of type `B` be attached to a single connector and each port instance of the coordinator be attached to n connectors. The behaviours of the two component types enforce that once the resource is acquired by a component of type `B`, it can only be released by the same component.

- **Assumed property of operands:** '*a component exits its critical section after* ***`finish`*** *and cannot enter it again until* ***`begin`***'

  $$\forall\, i \leqslant n, \mathtt{AG}\big(finish[i] \rightarrow \mathtt{A}[\neg cs[i] \ \mathtt{W}\ begin[i]]\big),$$

  where $cs[i]$ is an atomic predicate that evaluates to true when the $B[i]$ component is in the critical section (e.g. in state `work` for the behaviour shown in Figure 6).

- **Characteristic property of architecture:** '*no two components are in their critical section simultaneously*'

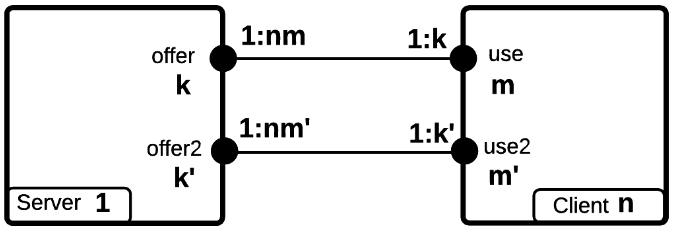


Figure 7: Architecture diagram of the Client-Server style

$$\mathtt{AG}\neg(\bigvee_{i\neq j\in[1,n]} cs[i] \wedge cs[j])$$

3.1.2 Client-server style

The Client-server architecture style ensures that only one client can use a service offered by the server at each time. It consists of two parameter component types `Server` and `Client` with 1 and n instances, respectively. In the diagram of Fig. 7, the server provides two services through port types `offer`, `offer2`. Client has two corresponding port types `use`, `use2`. Since the cardinalities of `offer` and `offer2` are k and k', respectively, each component instance of type `Server` has k port instances of type `offer` and k' port instances of type `offer2`. Similarly, each component instance of type `Client` has m port instances of type `use` and m' port instances of type `use2`.

Two connector motifs connect `use` (resp. `use2`) with `offer` (resp. `offer2`). The multiplicity/degree constraint of `offer` is $1 : n\,m$. The multiplicity/degree constraint of `use` is $1 : k$. Since both multiplicities are 1, all connectors are again binary. Because of the degree constraints, each port instance of `use` must be attached to k connectors, while each port instance of `offer` must be attached to $n\,m$ connectors, i.e. all port instances of `use` are connected to all port instances of `offer`.

- **Assumed property of operands:** '*the services are provided synchronously, i.e. as atomic actions*'. This is not a temporal property. The duration that a client uses a service is abstracted to be equal to 0.

- **Characteristic property of architecture:** '*only one client can use a provided service at each time*'

$$\forall\, i, j \leqslant n, \forall\, p \leqslant k,\ \mathtt{AG}\big(\neg \mathit{Client}[i].\mathit{use}[p] \wedge \mathit{Client}[j].\mathit{use}[p]\big)\,,$$
$$\forall\, i, j \leqslant n, \forall\, p \leqslant k,\ \mathtt{AG}\big(\neg \mathit{Client}[i].\mathit{use2}[p] \wedge \mathit{Client}[j].\mathit{use2}[p]\big)\,.$$

3.1.3 Action flow style

The Action flow style enforces a sequence of actions. It has one coordinator component of type `Action Flow Manager` and n parameter components of type `B`. Each component type has four port types: `start`, `actBegin`, `actEnd`, `finish`. The cyclic behaviour of the coordinator enforces an order on the actions of the operands. In the behaviour of the manager, `abi` and `aei` (instances of `actBegin` and `actEnd`, resp.) stand for "action i begin " and "action i end".

Each operand component c of type B provides n_a^c port instances of type `actBegin` and of type `actEnd`. Notice that n_a^c might be different for different operands of type B. The cardinalities of port types `ab` and `ae` are both equal to $N = \sum_{c:B} n_a^c$, where the sum is over all operands of type B. The multiplicity and degree constraints require that there be only binary connectors.

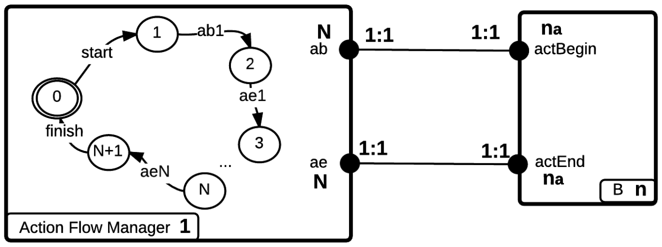


Figure 8: Architecture diagram of the Action flow style

- **Assumed property of operands:** '*each action is explicitly terminated by* `actEnd` *and no other action can be started until then*'

$$\forall i, j \leqslant n_a^c,\ \mathtt{AG}\big(B[c].actBegin[i] \to \mathtt{AX}\ \mathtt{A}[B[c].executing[i] \land \neg B[c].executing[j]\ \mathtt{W}\ B[c].actEnd[i]]\big)\,,$$
$$\forall i \leqslant n_a^c,\ \mathtt{AG}\big(B[c].actEnd[i] \to \mathtt{AX}\ \mathtt{A}[\neg B[c].executing[i]\ \mathtt{W}\ B[c].actBegin[i]]\big)\,,$$

where $B[c].executing[i]$ is an atomic predicate that evaluates to true when the $B[c]$ component is executing action i.

- **Characteristic property of architecture** is the conjunction of: a) '*on each action flow's execution, every action begins only after its previous action has ended*' b) '*on each flow execution, every action occurs at most once*' c) '*the flow finishes only after the last action has ended*', d) '*if an action ends, it can end only after it begins again*' formalised by the following CTL formulas, in which the index i denotes the position of an action in the action flow.

  We consider the following mappings:

  - from indices to components $seq_c : [1, N] \to C$, where C is a set containing all operands that execute an action;
  - from indices to actions $seq_a : [1, N] \to A$, where A is a set containing all actions of the operands,

  such that the action $seq_a(i)$ belongs to the component $seq_c(i)$.

$$\forall 1 < i \leqslant N,\ \mathtt{AG}\big(start \to \mathtt{AX}\ \ \mathtt{A}\big[\neg B[seq_c(i)].actBegin[seq_a(i)]\ \mathtt{W}\ B[seq_c(i)].actEnd[seq_a(i-1)]\big]\big)\,,$$
$$\forall 1 \leqslant i \leqslant N,\ \mathtt{AG}\big(B[seq_c(i)].actBegin[seq_a(i)] \to \mathtt{AX}\ \ \mathtt{A}\big[\neg B[seq_c(i)].actBegin[seq_a(i)]\ \mathtt{W}\ start\big]\big)\,,$$
$$\mathtt{AG}\big(start \to \mathtt{AX}\ \ \mathtt{A}[\neg finish\ \mathtt{W}\ B[seq_c(i)].actEnd[N]]\big)\,,$$
$$\forall 1 \leqslant i \leqslant N,\ \mathtt{AG}\big(B[seq_c(i)].actEnd[seq_a(i)] \to \mathtt{AX}\ \ \mathtt{A}\big[\neg B[seq_c(i)].actEnd[seq_a(i)]\ \mathtt{W}\ B[seq_c(i)].actBegin[seq_a(i)]\big]\big)\,.$$

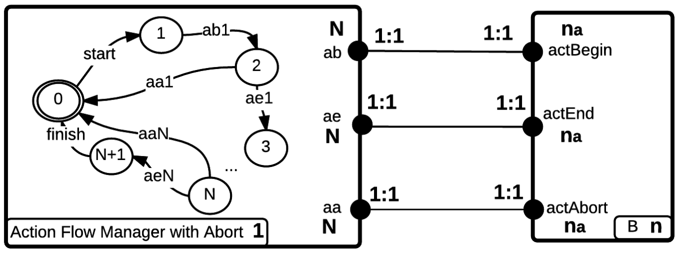


Figure 9: Architecture diagram of the Action flow with abort style

3.1.4 Action flow with abort style

The Action flow with abort architecture style is very similar to the Action flow style. It enforces a sequence of N actions. However, each operand has the ability to abort an action. If a component aborts, the behaviour of the manager is reset back to its initial state. In this style, both component types have an additional port type `actAbort`. In the coordinator behaviour, `aai` stands for "action i abort".

- **Assumed property of operands:** '*each action is explicitly terminated by `actEnd` or `actAbort` and no other action can be started until then*'

$$\forall i, j \leqslant n_a^c,\ \mathtt{AG}\big(B[c].actBegin[i] \to \mathtt{AX}\ \mathtt{A}\big[B[c].executing[i] \wedge \neg B[c].executing[j] \ \mathtt{W}\ B[c].actEnd[i] \vee actAbort[i]\big]\big)\,,$$
$$\forall i \leqslant n_a^c,\ \mathtt{AG}\big(B[c].actEnd[i] \vee actAbort[i] \to \mathtt{AX}\ \mathtt{A}[\neg B[c].executing[i]\ \mathtt{W}\ B[c].actBegin[i]]\big)\,,$$

  where $B[c].executing[i]$ is an atomic predicate that evaluates to true when the $B[c]$ component is executing action i.

- **Characteristic property of architecture** is the conjunction of first three characteristic properties of Action flow with the properties: e) '*if an action is ended or aborted, it can end or abort only after it begins again*' and f) '*when an action is aborted, an action can be executed only after the flow is reset*'.

  We consider the following mappings:

  – from indices to components $seq_c : [1, N] \to C$, where C is a set containing all operands that execute an action;

  – from indices to actions $seq_a : [1, N] \to A$, where A is a set containing all actions of the operands,

  such that the action $seq_a(i)$ belongs to the component $seq_c(i)$.

$$\forall 1 \leqslant i \leqslant N,\ \mathtt{AG}\big(B[seq_c(i)].actEnd[seq_a(i)] \vee B[seq_c(i)].actAbort[seq_a(i)] \to \mathtt{AX}\ \ \mathtt{A}\big[\neg B[seq_c(i)].actAbort[seq_a(i)] \wedge \neg B[seq_c(i)].actEnd[seq_a(i)]\ \mathtt{W}\ B[seq_c(i)].actBegin[seq_a(i)]\big]\big)\,,$$
$$\forall 1 \leqslant i, j \leqslant N,\ \mathtt{AG}\big(B[seq_c(i)].actAbort[seq_a(i)] \to \mathtt{AX}\ \ \mathtt{A}\big[\neg B[seq_c(j)].actBegin[seq_a(j)]\ \mathtt{W}\ start\big]\big)\,.$$

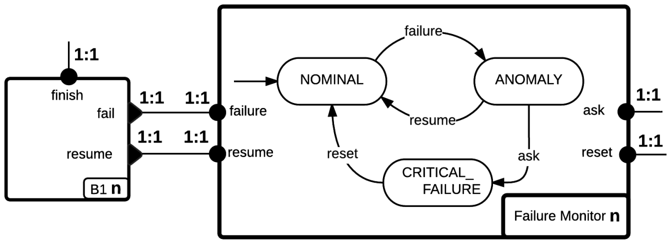


Figure 10: Failure monitoring style

3.1.5 Failure monitoring style

The Failure monitoring (Fig. 10) provides monitor components that observe the state of other components. It consists of n coordinator components of type `Failure Monitor` and n parameter components of type `B1`. The cardinality of all port types is 1. Multiplicities and degrees require that each `B1` component instance be connected to its dedicated `Failure monitor` instance.

A `B1` component may enter the following three states: `NOMINAL`, `ANOMALY` and `CRITICAL_FAILURE`. When in `NOMINAL` state, the component is performing correctly. If the component cannot be reached, or if the engineering data is not correct the component enters the `ANOMALY` state. If a fixed time has passed in which the component has remained in `ANOMALY`, the component enters the `CRITICAL_FAILURE` state. An architecture of this style is shown in Fig. 17.

- **Assumed property of operands:** is '*B1 will not execute any actions between fail and resume*'.

$$\forall c \leqslant n,\ \mathtt{AG}\big(\mathit{B1}[c].\mathit{fail} \rightarrow \mathtt{AX}\ \mathtt{A}\big[(\mathit{B1}[c].\mathit{pause}\ \mathtt{W}\ \mathit{B1}[c].\mathit{resume})\big]\ ,$$

  where $\mathit{B1}[c].\mathit{pause}$ is an atomic predicate that evaluates to false when the component $B1[c]$ executes any action other than resume.

- **Characteristic property of architecture:** '*if a failure occurs, a finish happens only after a resume or reset*'

$$\forall c \leqslant n,\ \mathtt{AG}\big(\mathit{B1}[c].\mathit{fail} \rightarrow \mathtt{AX}\ \ \mathtt{A}\big[\neg \mathit{B1}[c].\mathit{finish}\ \mathtt{W}\ (\mathit{B1}[c].\mathit{resume} \vee \mathit{reset})\big]\big)\ .$$

3.1.6 Mode management style

The Mode management style restricts the set of enabled actions, i.e. the actions that can be executed, according to a set of predefined modes. It consists of one coordinator of type `Mode Manager`, n parameter components of type `B1` and k parameter components of type `B2`. Each `B2` component *triggers* the transition of the `Mode Manager` to a specific mode. The coordinator manages which actions of the `B1` components can be executed in each mode.

The behaviour of the `Mode Manager` has k states, one state per mode. `Mode Manager` has a port type `toMode` with cardinality k and k port types `inMode` with cardinality 1. Each port instance of type `toMode` must be connected through a binary connector with the `changeMode` port of a dedicated `B2` component.

`B1` has k port types `modeBegin` with cardinality $\mathtt{mc}[0,1]$. In other words, a component instance of `B1` might have any number of port instances of types `modeBegin` from 0 until k. `B1` has also a `modeEnd` port type with cardinality k. `mib` stands for "mode i begin" and indicates that an action

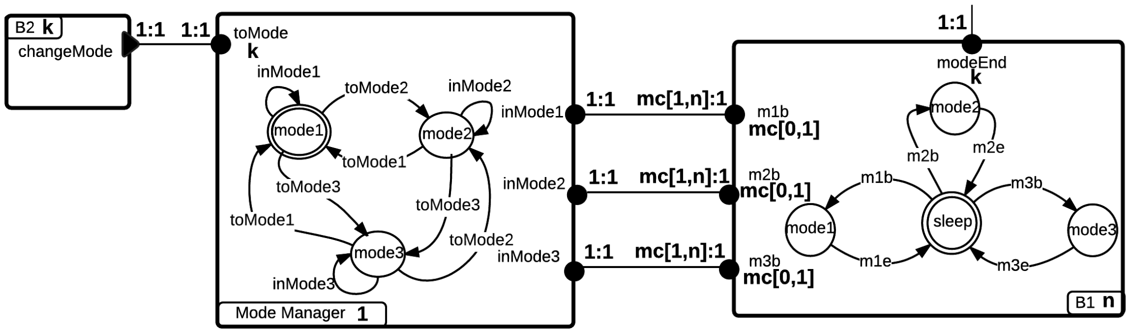


Figure 11: Architecture diagram of the Mode management style (component behaviour is shown for k=3)

that is enabled in mode i has begun its execution. `mie` stands for "mode i end" and indicates that an action that is enabled in mode i has finished its execution. Each `inMode` port instance of the `Mode Manager` must be connected with the corresponding `modeBegin` port instances of all `B2` components through an n-ary connector.

- **Assumed property of operands:** '*a component of type* `B1` *executes actions allowed in mode i only after it enters mode i*'

$$\forall\, i \leqslant k, \mathtt{AG}\big(m[i]e \rightarrow \mathtt{A}[\neg \mathit{mode}[i] \ \mathtt{W}\ m[i]b]\big)\,,$$

  where $\mathit{mode}[i]$ is an atomic predicate that evaluates to true when a $B1$ component is performing an action allowed in mode i.

- **Characteristic property of architecture:** '*an action is only performed in a mode where it is allowed*'

$$\forall i \leqslant k,\ \mathtt{AG}\big(B1.m[i]b \rightarrow \mathit{ModeManager.inMode}[i]\big)\,.$$

3.1.7 Buffer management style

The Buffer management style controls the access of a set of producers and consumers to a buffer. It consists of a single coordinator component of type `Buffer Manager`, n parameter components of type `Producer` and m parameter components of type `Consumer`. The `Buffer Manager` restricts the behaviour of producers by allowing them to write data to the buffer only if the buffer is not `full`. Similarly, the `Buffer Manager` restricts the behaviour of consumers by allowing them to retrieve data from the buffer only if the buffer is not `empty`.The cardinalities of all port types are 1. The multiplicity and degree constraints require that each `Producer` component instance and `Consumer` component instance be connected to the `Buffer manager` component instance through binary connectors.

- **Assumed property of operands:** no assumptions

- **Characteristic property of architecture** is the conjunction of the following properties: 1)'*data can be stored to the buffer only if the buffer is not full*'; 2)'*data can be retrieved from the buffer only if the buffer is not full*'

$$\mathtt{AG}\big(\mathit{BufferManager.enabled(full)} \rightarrow \mathtt{AX}\ \mathtt{A}\big[\neg \mathit{BufferManager.put}\ \mathtt{W}\ \mathit{BufferManager.get}\big]\big)\,,$$

$$\mathtt{AG}\big(\mathit{BufferManager.enabled(empty)} \rightarrow \mathtt{AX}\ \mathtt{A}\big[\neg \mathit{BufferManager.get}\ \mathtt{W}\ \mathit{BufferManager.put}\big]\big)\,,$$

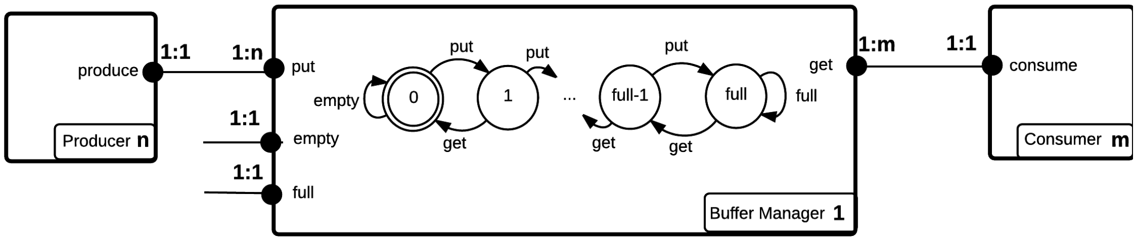


Figure 12: Architecture diagram of the Buffer management style

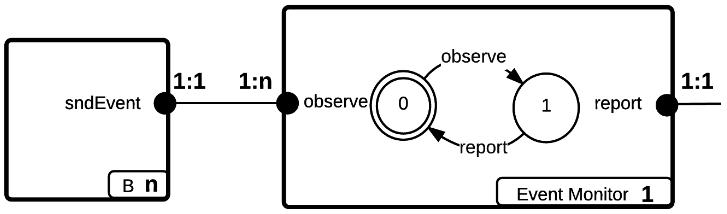


Figure 13: Architecture diagram of the Event monitoring style

where *enabled*(*empty*) and *enabled*(*full*) are atomic predicates that evaluate to true when the *empty* and *full* actions of the *Buffer manager* are enabled.

3.1.8 Event monitoring style

The Event monitoring style is a special case of the Buffer management architecture style for a buffer of size 1 and consumer cardinality equal to 1. The Event monitoring style provides a monitor component that tracks events of other components. For each event, the monitor generates a report and sends it to a dedicated service component. The style consists of a single coordinator component of type `Event Monitor` and n parameter components of type `B`. The cardinalities of all port types are 1. The multiplicity and degree constraints require that each `B` component instance be connected to the `Event monitor` component instance. All connectors must be binary.

- **Assumed property of operands:** no assumptions
- **Characteristic property of architecture:** '*if an event is sent to the monitor, another event can be sent to the monitor only after a report is generated*'

$$\forall c, c' \leqslant n,\ \mathtt{AG}\big(B[c].\mathit{sndEvent} \rightarrow \mathtt{AX}\ \mathtt{A}\big[\neg B[c'].\mathit{sndEvent}\ \mathtt{W}\ \mathit{EventMonitor.report}\big]\big)\,.$$

3.1.9 Priority management style

The Priority management style enforces a priority protocol on the set of actions of n components. It consists of a single coordinator component of type `Priority Manager` and n parameter components of type `B`. The `Priority Manager` checks first whether the action with the highest priority can be executed. If this action is enabled, it executes it and returns to its initial state. If this action is not enabled, it checks whether the action with the second highest priority can be executed. If this action is enabled, it executes it and returns to its initial state. If this action is not enabled, it checks whether the action with the third highest priority can be executed, etc. The cardinalities of the `action` and `noAction` port types are n. The cardinalities of all other port

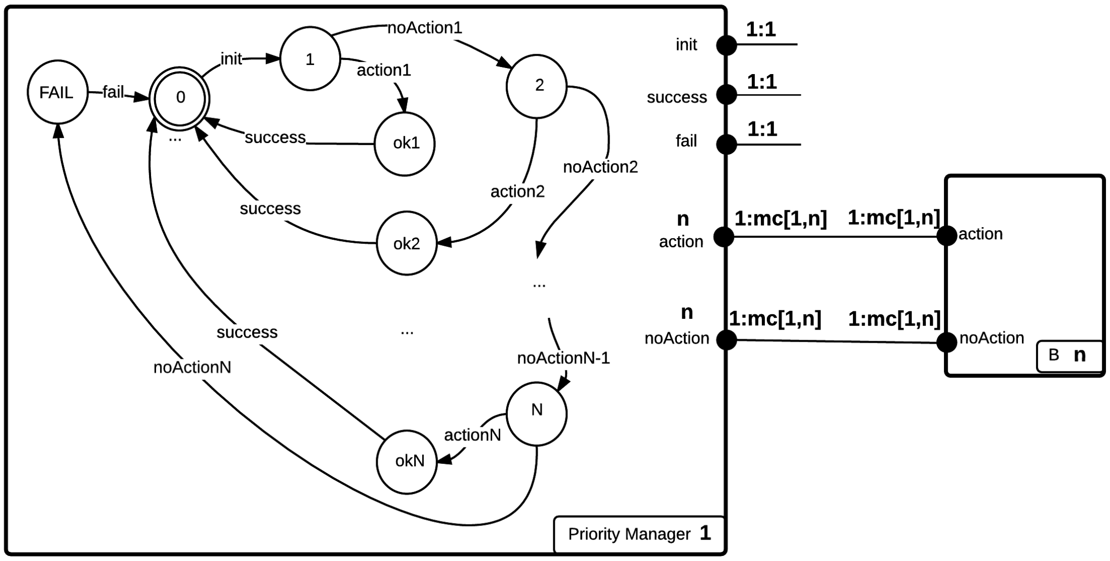


Figure 14: Architecture diagram of the Priority management style

types are 1. The multiplicity and degree constraints require that each `B` component instance be connected to the `Priority manager` component instance through binary connectors.

- **Assumed property of operands:** '*`action` and `noaction` are mutually exclusive actions*'

$$\forall c \leqslant n,\ \mathtt{AG}\big(\mathtt{enabled}(B[c].action)\ \mathtt{XOR}\ \mathtt{enabled}(B[c].noaction)\big)\,,$$

  where $\mathtt{enabled}(B[i].action)$ (resp. $B[i].action$) is an atomic predicate that evaluates to true when the $B[i].action$ (resp. $B[i].action$) action is enabled.

- **Characteristic property of architecture** is the conjunction of the following properties: 1)'*a component cannot execute `action` $i+1$ unless `noaction` i (previous action with higher priority) has been fired*'; 2)'*a component cannot execute `noaction` $i+1$ unless `noaction` i (previous action with higher priority) has been fired*'; 3) '*a new cycle can start only after there is a `success` or `fail`*'; 4) '*once a cycle starts, `success` can be executed only after an `action` is executed*'; 5) '*once an `action` is executed, `fail` cannot be executed unless there is an `init`.*

$$\forall i < n,\ \mathtt{AG}\big(PriorityManager.init \to \mathtt{A}\big[\neg B[i+1].action\ \mathtt{W}\ B[i].noaction\big]\big)\,,$$
$$\forall i < n,\ \mathtt{AG}\big(PriorityManager.init \to \mathtt{A}\big[\neg B[i+1].noaction\ \mathtt{W}\ B[i].noaction\big]\big)\,,$$
$$\mathtt{AG}\big(PriorityManager.init \to \mathtt{AX}\ \mathtt{A}\big[\neg PriorityManager.init\ \mathtt{W}$$
$$(PriorityManager.success\ \vee\ PriorityManager.fail)\big]\big)\,,$$
$$\mathtt{AG}\big(PriorityManager.init \to \mathtt{A}\big[\neg PriorityManager.success\ \mathtt{W}\ (\bigvee_{i=1}^{n}\ B[i].action\big]\big)\,,$$
$$\forall i < n,\ \mathtt{AG}\big(B[i].action \to \mathtt{A}\big[\neg PriorityManager.fail\ \mathtt{W}\ PriorityManager.init\big]\big)\,.$$

3.2 Full list of requirements

The following table contains the complete list of functional requirements of the case-study.

ID	**Description**
CDMS-001	The CDMS shall be connected to the following sub-systems: Payload (PL), Communication (COM), Electrical Power Subsystem (EPS) through an I2C bus.
CDMS-003	The CDMS shall supervise the correct execution of the software functions on the other subsystems. If a sensor or subsystem indicates anomalous signals the CDMS shall ask the EPS for a reset of the malfunctioning hardware.
CDMS-004	The CDMS shall be able to save its status in order to resume correct operations following an unexpected reset.
CDMS-006	The CDMS shall manage the data generated from the payload and housekeeping routines in a non volatile memory.
CDMS-007	The CDMS shall periodically reset both the internal and external watchdogs and contact the EPS subsystem with a "heartbeat".
PL-001	The Payload shall be able to add a scenario to the payload board.
PL-002	The Payload shall be able to execute scenario telecommand.
PL-003	The Payload shall be able to abort any operation on the payload and data transfer to transfer data from the payload to the non volatile memory.
PL-004	The Payload shall be able to check the advancement of the payload board internals algorithms
PL-005	The Payload shall be able to track the upload, execution and result retrieval of a scenario and enable the corresponding actions.
PL-006	The Payload subsystem shall have the following modes: IDLE, SCENARIO_READY, STARTED and RESULT_READY.
PL-007	The payload shall operate when it is not in IDLE mode.
PL-008	In SCENARIO_READY mode a scenario shall be loaded on the payload board.
PL-009	In STARTED mode, the payload data acquisition shall begin.
PL-010	The payload shall poll the payload board to check if its memory is full. If the memory is full, the payload shall change to RESULT_READY mode.
PL-011	In RESULT_READY mode, the data shall be transferred to the CDMS non-volatile memory. If the data retrieval is not finished, payload shall continue the payload data acquisition until the data retrieval is completed.
HK-001	The CDMS shall have a Housekeeping activity dedicated to each subsystem.
HK-003	When line-of-sight communication is possible, housekeeping information shall be transmitted through the COM subsystem.
HK-004	When line-of-sight communication is not possible, housekeeping information shall be written to the non-volatile flash memory.
HK-005	A Housekeeping subsystem shall have the following states: NOMINAL, ANOMALY, and CRITICAL_FAILURE.
HK-006	In NOMINAL state, the subsystem shall perform correctly.

HK-007	If a process failure occurs or if the engineering data are not correct the subsystem shall enter the ANOMALY state.
HK-008	After MAX[3] seconds in ANOMALY, the subsystem shall enter the CRITICAL_FAILURE state.
HK-009	In CRITICAL_FAILURE state, the subsystem shall contact the EPS and demand a restart of the malfunctioning subsystem.
HK-010	During NOMINAL operation the subsystem shall be contacted to retrieve engineering data.
I2C-001	A single user shall send one message at a time.
I2C-002	I2C_sat shall implement the I^2C protocol.
Log-001	Every time a hardware error is produced, it shall be stored in a memory region in the RAM.
Log-002	The dedicated RAM region shall be read and written atomically.
Mem-001	The Central Software System shall have a dedicated Flash Memory Manager activity for managing flash memory operations.
Mem-002	Flash memory shall be read and written atomically.
Mem-003	Flash Memory Manager shall return the SUCCESS or FAIL status for each requested operation.
Mem-004	Upon a read request, the Flash Memory Manager shall read the data from the flash memory and perform the Circular Redundancy Check (CRC).
Mem-005	If CRC fails, the Flash Memory Manager shall reread the data from the flash memory.
Mem-006	For the same read request, the number of attempts by the Flash Memory Manager to read data from the flash memory shall have a value not larger than the parameter MAX_FM_READS.
Mem-007	If the number of attempts by the Flash Memory Manager to read data from the flash memory exceeds MAX_FM_READS, the read operation shall be abandoned and a failure shall be reported.

Table 1: Complete list of requirements

3.3 BIP model design by architecture application

We illustrate the architecture-based approach on the `CDMS status`, `MESSAGE_ LIBRARY` and `HK PL` components. In particular, we present the application of Action flow, Mode management, Client-Server and Failure monitoring architectures to discharge a subset of the CubETH functional requirements presented in the previous section. We additionally present the result of the composition of Client-Server and Mode management architectures.

3.3.1 Application of Action flow architecture

Requirement CDMS-007, presented in Tab. 1, describes the functionality of `CDMS status`. The corresponding BIP model is shown in Fig. 15. `Watchdog reset` is an operand component, which is responsible for resetting the internal and external watchdogs. `CDMS status ACTION FLOW` is the coordinator of the architecture applied on `Watchdog reset` that imposes the following order of

[3]MAX is a parameter. Its value must be fixed through analysis or simulation.

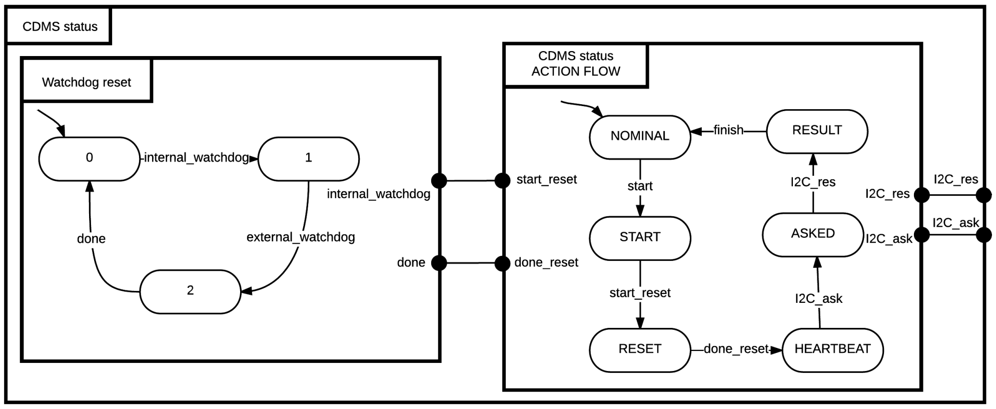
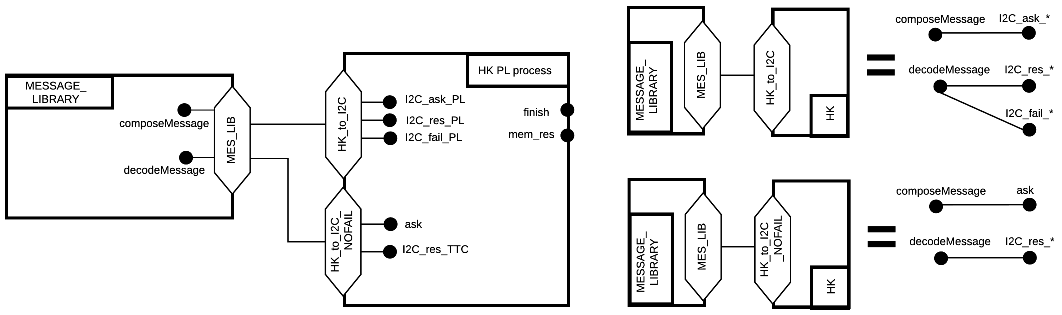


Figure 15: Application of Action flow architecture



(a) Architecture application

(b) Hexagons of Fig. 16a

Figure 16: Application of Client-Server architecture

actions: 1) internal watchdog reset; 2) external watchdog reset; 3) send heartbeat and 4) receive result.

3.3.2 Application of Client-Server architecture

Requirements HK-001 and HK-003, presented in Tab. 1, suggest the application of the Client-Server architecture on the `HK PL`, `HK CDMS`, `HK EPS` and `HK COM` housekeeping compounds (Fig. 5). The four housekeeping compounds are the clients of the architecture. In Fig. 16a, we show how Client-Server is applied on the `HK PL process` component, which is a subcomponent of `HK PL`. `HK PL process` uses the `composeMessage` and `decodeMessage` C/C++ functions of the `MESSAGE_LIBRARY` component to encode and decode information transmitted to and from the `COM` subsystem. Thus, the `MESSAGE_LIBRARY` is a server used by the `HK PL process` client. To enhance readability of figures in Fig. 16a, we use hexagons to group interaction patterns of components. The meaning of these hexagons is explained in Fig. 16b.

3.3.3 Application of Failure monitoring architecture

Requirement HK-005, presented in Tab. 1, suggests the application of the Failure monitoring architecture as shown in Fig. 17. The BIP model comprises the `HK PL process` operand and the

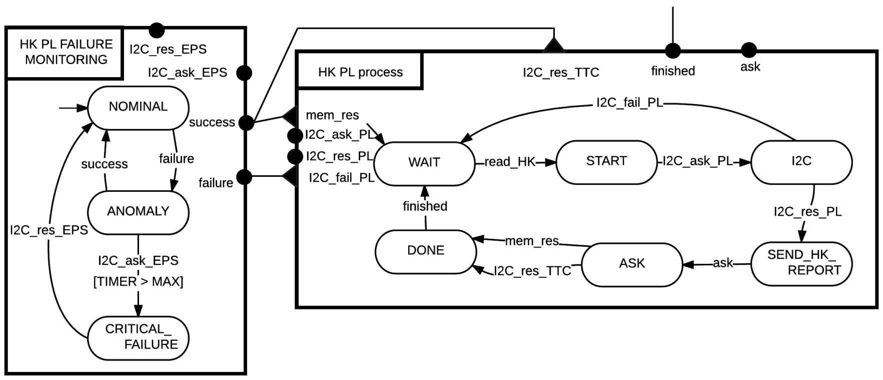


Figure 17: Application of Failure monitoring architecture

HK PL FAILURE MONITORING coordinator. The success port of HK PL FAILURE MONITORING is connected with the mem_res and I2C_res_TTC ports of HK PL process. The failure port of HK PL FAILURE MONITORING is connected with the I2C_fail_PL port of HK PL process. The HK PL process component executes 6 actions in the following order: 1) start procedure; 2) ask Payload for engineering data; 3) receive result from Payload or (in case of fail) abort; 4) if line of sight communication is possible send data to COM, if line of sight communication is not possible make a write request to the memory; 5) depending on action 4 either receive COM result or memory result and 6) finish procedure.

3.3.4 Application of Mode management architecture

Requirements HK-003 and HK-004, presented in Tab. 1, suggest the application of a Mode management architecture with two modes: 1) TTC mode, in which line of sight communication is possible and 2) MEMORY mode, in which line of sight communication is not possible. The corresponding BIP model, shown in Fig. 18, comprises the HK PL process, s15_1 and s15_2 operands and the Packet store MODE MANAGER coordinator. During NOMINAL operation, the Payload subsystem is contacted to retrieve engineering data. Depending on the mode of Packet store MODE MANAGER, those data is then sent to the non-volatile memory, i.e. mem_write_req transition, or directly to the COM subsystem, i.e. ask_I2C_TTC transition. The mode of Packet store MODE MANAGER is triggered by the s15_1, s15_2 services.

3.3.5 Composition of architectures

The architecture composition was formally defined in [2]. Here, we provide only an illustrative example. Combined application of architectures to a common set of operand components results in merging the connectors that involve ports used by several architectures. For instance, Fig. 19 shows the composition of Client-Server and Mode management architectures. The HK PL process component is a sub-component of HK PL. The application of the Client-Server architecture (Fig. 16) connects its port ask with the port composeMessage of MESSAGE_LIBRARY through the MES_LIB-HK_to_I2C interface with a binary connector. Similarly, the application of the Mode management architecture (Fig. 18) connects the same port with the port ask_I2C_TTC of Packet store MODE MANAGER with another binary connector. The composition of the two architectures results in the two connectors being merged into the ternary connector ask-ask_I2C_TTC-composeMessage (Fig. 19).

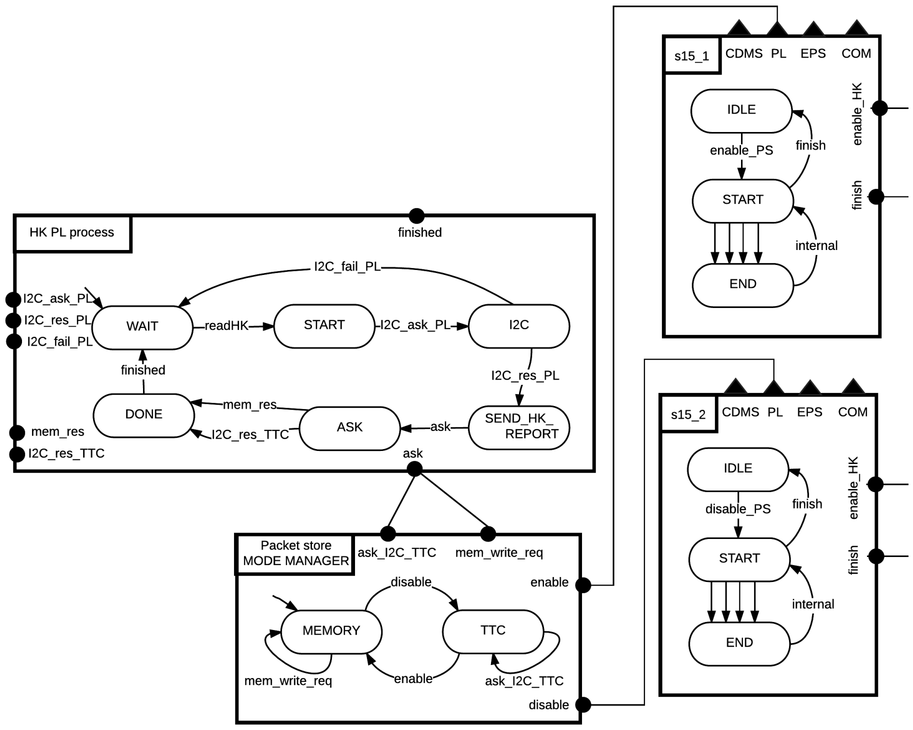


Figure 18: Application of Mode management architecture

3.4 Complete BIP model of the case study

In this section, we present the full componentization of the case study. The complete model consists of 22 operand components and 27 coordinator components. We omit the presentation of the `CDMS status` compound which was already presented in Sect. 3.3.1. We have shown in Fig. 5 the high level interaction model of the case study. As discussed earlier, we use hexagons to group interaction patterns of components. The meaning of connections between the hexagons of Fig. 5 is explained in Fig. 20.

3.4.1 Libraries

The model includes three library components that contain helper C/C++ functions. Fig. 21 shows the `I2C_sat_LIBRARY`, the `MEMORY_LIBRARY` and the `MESSAGE_LIBRARY` components. All of them are operands. `I2C_sat_LIBRARY` contains functions that allow communication on the `I2C_sat` bus. `I2C_sat_LIBRARY` contains functions that allow writing, reading and checking the CRC in the non-volatile flash memory. `I2C_sat_LIBRARY` allows a user to compose a message that is going to be send over the `I2C_sat` bus and also to decode a message received from the `I2C_sat` bus.

3.4.2 Housekeeping report enable, housekeeping report disable

The model includes two service components, namely `s3_5` and `s3_6` that are in charge of the activation and deactivation of the housekeeping subsystems. Both of them are operand components.

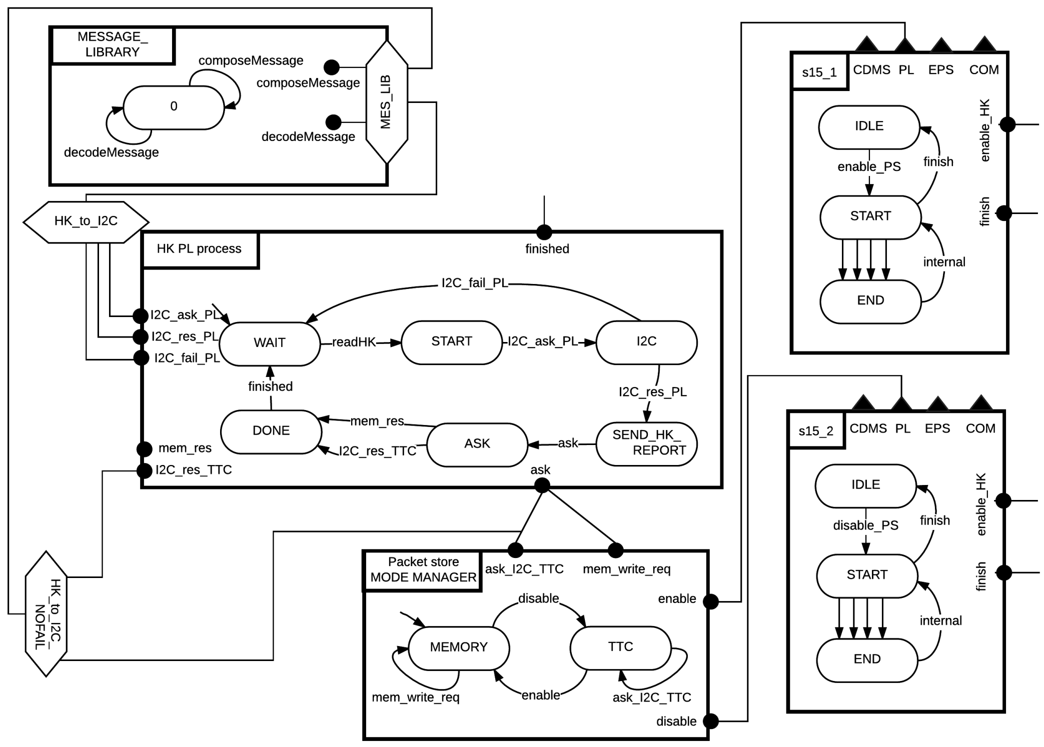


Figure 19: Composition of Client-Server and Mode management architectures

3.4.3 Packet storage enable and packet storage disable

The model includes two service components, namely `s15_1` and `s15_2` that modify the destination of the housekeeping data which is either the non volatile memory or the `COM` subsystem. Both of them are operand components.

3.4.4 Error logging

The `Error logging` compound, shown in Fig. 24, protects a specific RAM region that is accessible by many users. Every time a hardware error is produced it is stored in an array managed by this compound. No two users can log an error at the same time. It is composed of an operand component `Log` and a Mutual exclusion coordinator `Log MUTEX`.

3.4.5 Payload

The `Payload` (PL) compound, shown in Fig. 25, is in charge of payload operations. It is composed of the following subcomponents: 1) `s128_1` which adds a scenario to the payload board; 2) `s128_4` which executes a scenario telecommand; 3) `128_5` which aborts a payload operation; 4) `data_transfer` which transfers data from the payload to the non volatile memory; 5) `status_verification` which checks the advancement of the payload board internal algorithms and 6) and a `PL mode management` coordinator.

The `s128_1`. `s128_4`, `s128_5` and the `status_verification` components consist of a Mutual exclusion coordinator and a `128_* process` operand. The `data_transfer` component consists of a

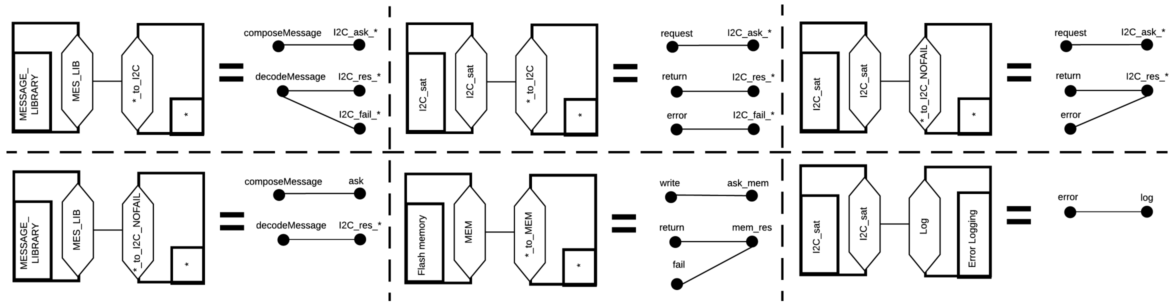
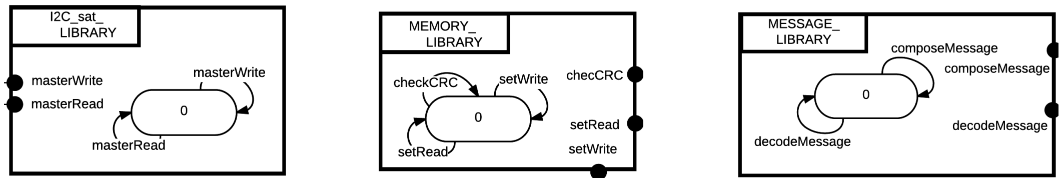
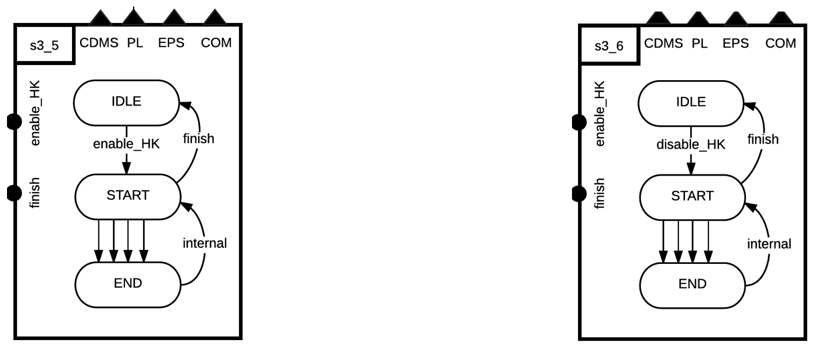


Figure 20: Meaning of connections between hexagons of Fig. 5



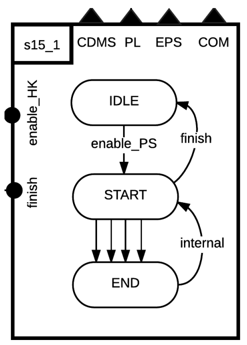
(a) The I2C_sat library component (b) The memory library component (c) The message library component

Figure 21: Library components

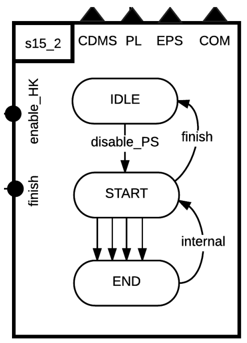


(a) The s3_5 component (b) The s3_6 component

Figure 22: Housekeeping report enable and disable



(a) The s15_1 component



(b) The s15_2 component

Figure 23: Packet storage enable and disable

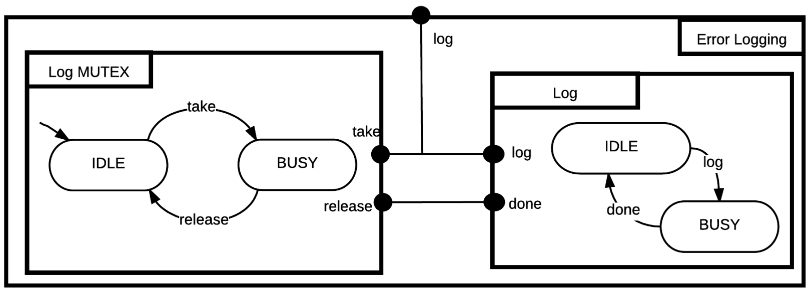


Figure 24: The error logging compound

Mode manager coordinator and the `data_transfer process` operand. The `status_verification` component consists of a Mutex coordinator and the `status_verification process` operand.

`Payload` has four modes of operation (`IDLE`, `SCENARIO_READY`, `STARTED` and `RESULT_READY`), which comprise the states of the `PL mode management` coordinator. In `IDLE` mode, the payload does not operate. In `SCENARIO_READY` mode, a scenario is loaded on the payload board. In `STARTED` mode, the payload data acquisition begins. The `status_verification` component polls the payload board to check whether its memory is full. Once the memory is full, the mode changes to `SCENARIO_READY`. `SCENARIO_READY` mode, the data is transferred to the non volatile flash memory, through the `data_transfer` component.

3.4.6 Housekeeping subsystems

There are three Housekeeping subsystems, namely `HK PL`, shown in Fig. 26, `HK EPS`, shown in Fig. 27, and `HK COM`, shown in Fig. 28, that are in charged of recovering engineering from the `PL`, `EPS`, `COM` subsystems of the satellite, respectively. The are composed of the same subcomponents: 1) a `HK MUTEX` coordinator; 2) an `HK process` operand; 3) an `HK MODE MANAGER` coordinator; 4) an `Packet store MODE MANAGER` coordinator and 5) a `HK FAILURE MONITORING` coordinator.

The `HK MUTEX` component ensures mutual exclusion on the access of the subsystem. During `NOMINAL` operation, a 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 MODE MANAGER` coordinator. The `HK MODE MANAGER` inhibits the HK process.

3.4.7 Internal Housekeeping

The `HK CDMS` compound, shown in Fig. 29, is internal to the `CDMS` subsystem. It is very similar to the rest of Housekeeping subsystems. The difference is that the Housekeeping data of the `HK CDMS` is not retrieved through the `I2C` bus but through internal processes (e.g. GPIO and state registers). The Failure monitoring coordinator is also removed because the `EPS` subsystem directly monitors the `HK CDMS` through the heartbeats.

3.4.8 I2C_sat

The `I2C_sat` compound, shown in Fig. 30, implements the `I2C_sat` bus protocol. The request transition is enabled as soon as a user wants to send a message through the `I2C` bus. It consists of the `I2C_sat read` operand, the `I2C_sat MODE MANAGEMENT` coordinator and the `I2C_sat write` operand. Mutual exclusion is ensured by the fact that this compound is the only one implementing the use of the `I2C` peripheral. Thus, once a request is issued no other user can access the bus until it has retuned to the `IDLE` state of `I2C_sat write`.

The `send` transition of `I2C_sat write` sends a message to the selected slave on the line. The `poll` transition of `I2C_sat read` fetches an answer from the slave.

3.4.9 Flash memory management

The reading and writing procedures to the external non volatile NOR flash memory are represented by the compound shown in Fig. 31. The memory device can be read and written through the write and read transitions. The `Flash Memory MUTEX` compound ensures that the memory is accessed by the users atomically. It consists of two operands (`Flash Memory Write` and `Flash Memory Read`), two Mode Manager coordinators and a Mutual exclusion coordinator.

After a write request, `Flash Memory Write` takes the buffer prepared by the user and starts writing it in the memory. From `WAIT` to `STATUS_WRITE` there is an internal transition which adds a delay for a certain period of time. This is the minimum time to wait for the memory device

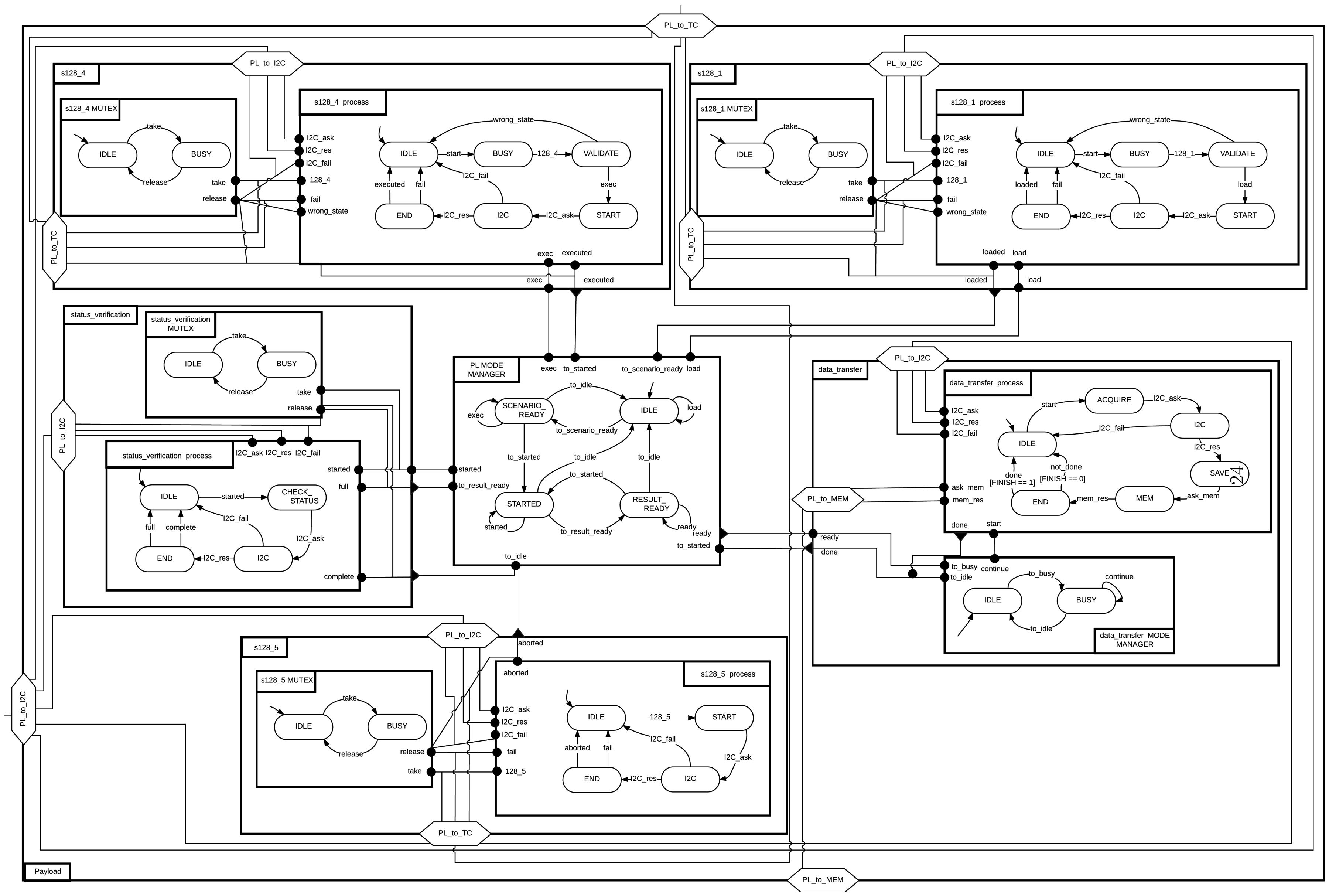


Figure 25: The Payload compound

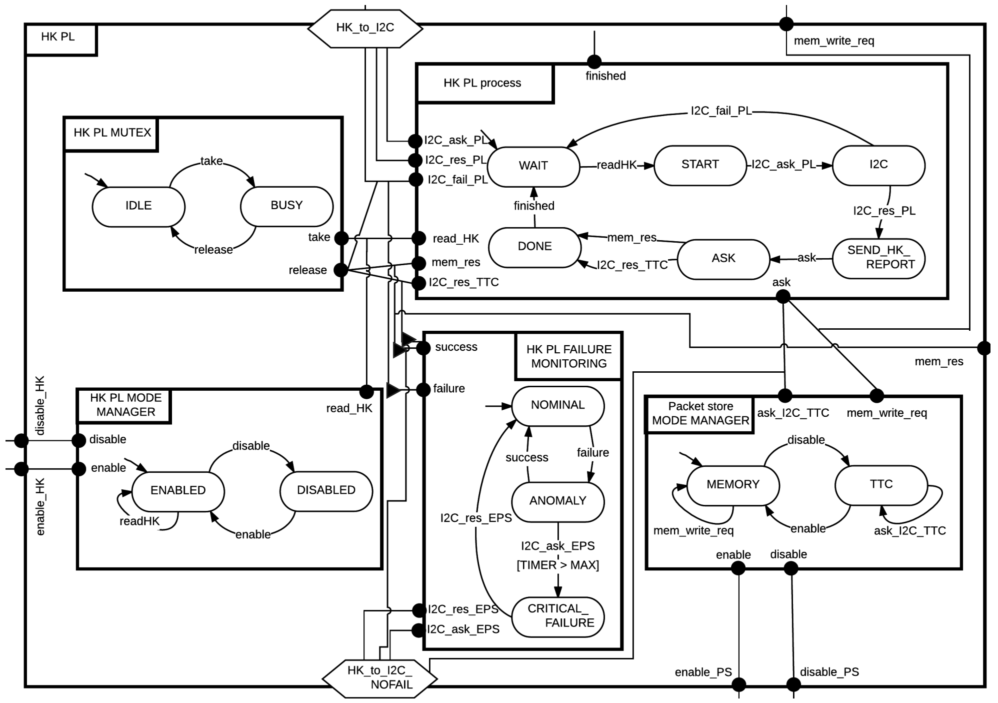


Figure 26: The Housekeeping Payload compound

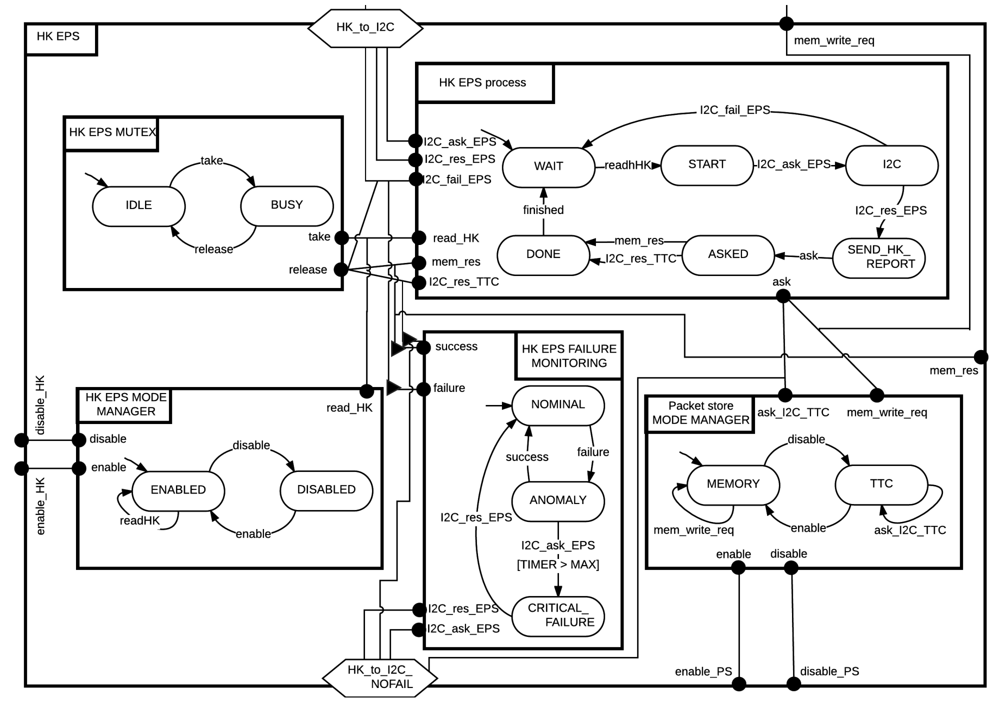


Figure 27: The Housekeeping EPS compound

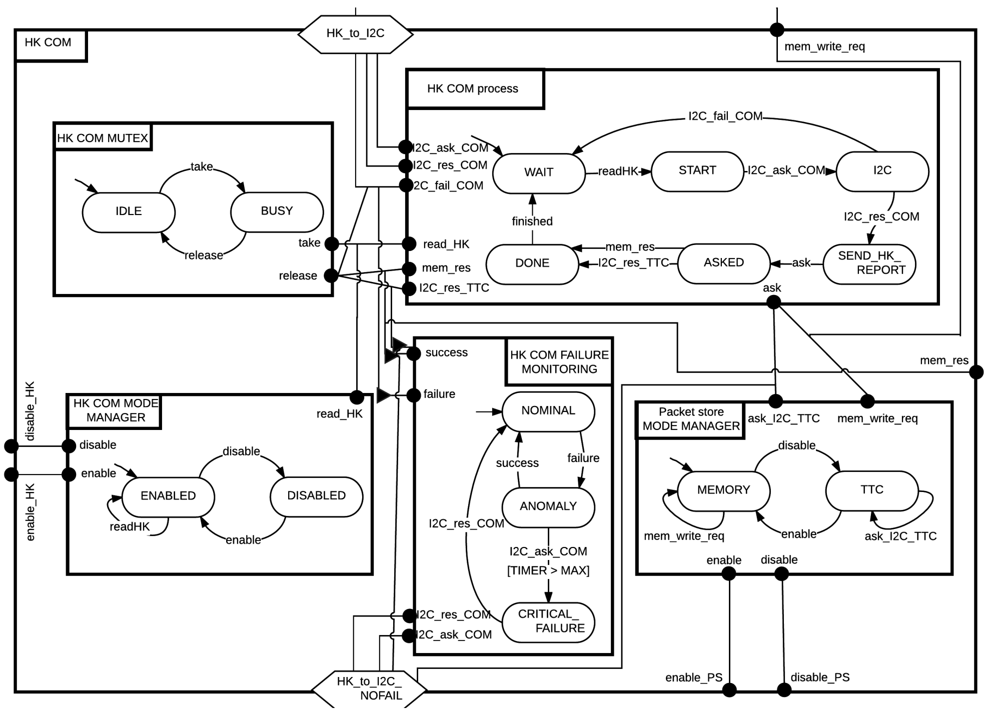


Figure 28: The Housekeeping COM compound

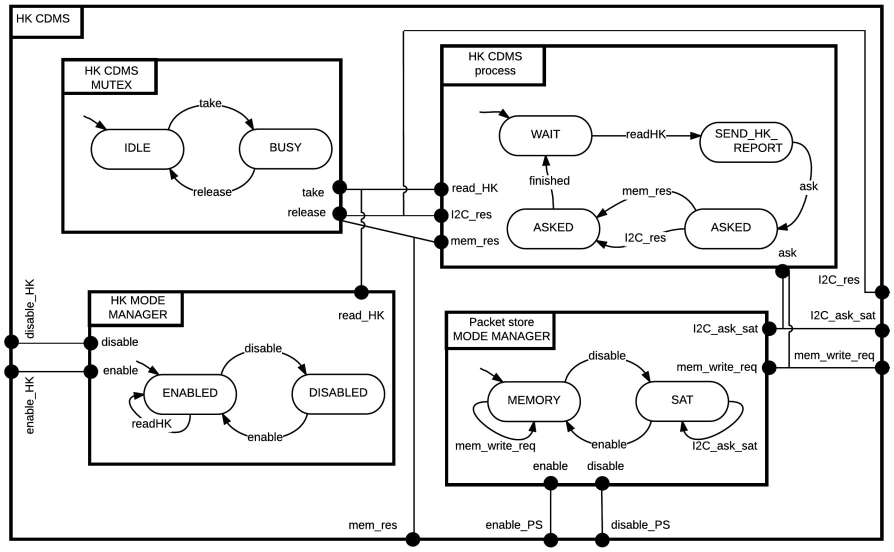


Figure 29: The Internal Housekeeping compound

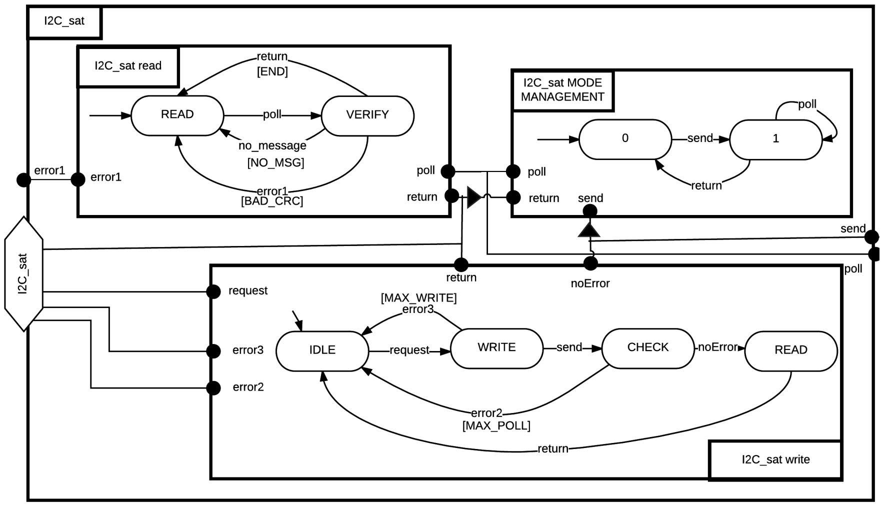


Figure 30: The I2C_sat compound

to effectively write on the buffer. If the buffer is bigger than the write sector, then the this procedure continues (synchronization of `write` with the `continue` of the `Write MODE MANAGER`). If there is no more data to write and the procedure was successful, the `ok_write` transition is fired and the coordinator enters the `DONE state`. The internal `finish` transition leads back to the `WRITE_BUFFER` state. If the memory does not perform as expected, e.g. internal timeout, internal error, the writing procedure is aborted with the `fail` transition.

After a read request, `Flash Memory Read` reads a part of the memory and puts data in the buffer indicated by the user. After the region is read, a CRC check is performed. If the result is declared as "bad", the memory region is read again (synchronization of `read` with the `continue` of the `Read MODE MANAGER`). After a maximum number of times is reached, the fail transition is fired. If the CRC check is successful, the `ok_read` transition is fired to the `DONE` state and the internal `finish` transition leads back to the `READ_BUFFER` state.

3.5 Full list of properties of the CubETH subsystems

We present the complete set of characteristic properties for the architectures applied in each CubETH subsystem. For brevity, we use symbolic names as predicates in the formulas. These symbolic names include a) a characteristic port that participates in the interaction and b) the Payload subcomponent in which the port is localized, whenever it is necessary for disambiguation. Note that we use components s128_4_takeTC1, s128_4_takeTC2, s128_1_takeTC1, s128_1_takeTC2, s128_5_takeTC1, s128_5_takeTC2 which fulfill the assumption that they have a critical state 'TAKE', in order to specify characteristic properties for the Mutual Exclusion management on the resources s128_4, s128_1 and s128_5, respectively. The concrete components that attempt to access these resources are external to the subsystems of the CubETH case study, thus we denote them in the characteristic properties using the aforementioned abstract components. the same applies for the components status_verifier1, status_verifier2, HKPL_reader1, HKPL_reader2, HKEPS_reader1, HKEPS_reader2, HKCOM_reader1, HKCOM_reader2, HKCDMS_reader1 and HKCDMS_reader2.

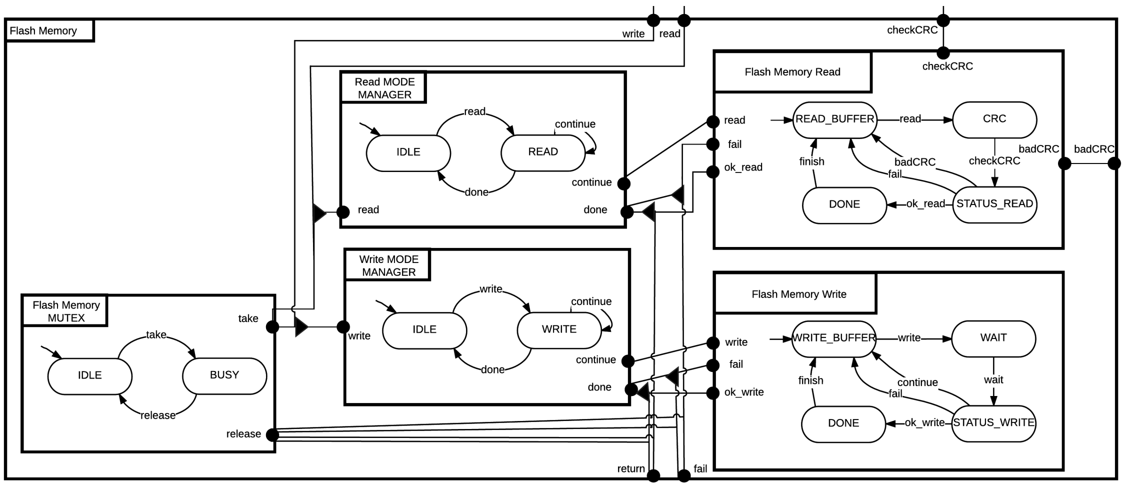


Figure 31: The flash memory compound

3.5.1 Properties of the Payload subsystem

-

Mutual Exclusion in s128_4

MX.1 AG (¬ (s128_4_takeTC1_inState(TAKE) ∧ s128_4_takeTC2_inState(TAKE)))

Mutual Exclusion in s128_1

MX.1 AG (¬ (s128_1_takeTC1_inState(TAKE) ∧ s128_1_takeTC2_inState(TAKE)))

Mutual Exclusion in s128_5

MX.1 AG (¬ (s128_5_takeTC1_inState(TAKE) ∧ s128_5_takeTC2_inState(TAKE)))

Mode Management in data_transfer

MD.1 AG(dat_trans_start → DTModeManager_inState(BUSY))

Mode Management in payload

MD.1i AG(s128_4_exec → (PLModeManager_inState(SCENARIO_READY))

MD.1ii AG(stat_ver_started → (PLModeManager_inState(STARTED))

MD.1iii AG(s128_4_load → PLModeManager_inState(IDLE))

MD.1iv AG(dat_trans_to_busy → PLModeManager_inState(RESULT_READY)))

Mutual Exclusion management in Status Verification

MX.1 AG (¬ (status_verifier1_inState(START) ∧ status_verifier2_inState(START)))

3.5.2 Properties of the I2C_sat subsystem

Mode management in I2C_sat

MD.1 AG(I2C_sat__process_poll $\rightarrow$ I2CsatModeManagement_inState(S1))

3.5.3 Properties of the HK PL subsystem

Mode Management in HK PL Packet Store

MD.1i AG (HKPL_mem_write_req $\rightarrow$ HKPL_PSModeMngment_inState(MEMORY))

MD.1ii AG (HKPL_ask_I2C_TTC $\rightarrow$ HKPL_PSModeMngment_inState(TTC))

Mode Management in HK PL

MD.1 AG (HKPL_read_HK $\rightarrow$ HKPL_ModeMngment_inState(ENABLED))

Mutual Exclusion management in HK PL

MX.1 AG ($\neg$ (HKPL_HK_reader1_inState(READ) $\wedge$ HKPL_HK_reader2_inState(READ)))

Failure Monitoring in HK PL

FM.1 AG (HKPL_I2C_fail_error $\rightarrow$ AX A[$\neg$ HKPL_finish W ((HKPL_I2C_res_TTC $\vee$ HKPL_mem_res) HKPL_I2C_res_EPS)]

3.5.4 Properties of the HK EPS subsystem

Mode Management in HK EPS Packet Store

MD.1i AG (HKEPS_mem_write_req $\rightarrow$ HKEPS_PSModeMngment_inState(MEMORY))

MD.1ii AG (HKEPS_ask_I2C_TTC $\rightarrow$ HKEPS_PSModeMngment_inState(TTC))

Mode Management in HK EPS

MD.1 AG (HKEPS_read_HK $\rightarrow$ HKEPS_ModeMngment_inState(ENABLED))

Mutual Exclusion management in HK EPS

MX.1 AG ($\neg$ (HKEPS_reader1_inState(READ) $\wedge$ HKEPS_reader2_inState(READ)))

Failure Monitoring in HK EPS

FM.1 AG (HKEPS_I2C_fail_error $\rightarrow$ AX A[$\neg$ HKEPS_finish W ((HKEPS_I2C_res_TTC $\vee$ HKEPS_mem_res) HKEPS_I2C_res_EPS)]

3.5.5 Properties of the HK COM subsystem

Mode Management in HK COM Packet Store

MD.1i AG (HKCOM_mem_write_req → HKCOM_PSModeMngment_inState(MEMORY))

MD.1ii AG (HKCOM_ask_I2C_TTC → HKCOM_PSModeMngment_inState(TTC))

Mode Management in HK COM

MD.1 AG (HKCOM_read_HK → HKCOM_ModeMngment_inState(ENABLED))

Mutual Exclusion management in HK COM

MX.1 AG (¬ (HKCOM_reader1_inState(READ) ∧ HKCOM_reader2_inState(READ)))

Failure Monitoring in HK COM

FM.1 AG (HKCOM_I2C_fail_error → AX A[¬ HKCOM_finish W ((HKCOM_I2C_res_TTC ∨ HKCOM_mem_res) HKCOM_I2C_res_EPS)]

3.5.6 Properties of the HK CDMS subsystem

Mode Management in HK CDMS Packet Store

MD.1i AG (HKCDMS_mem_write_req → HKCDMS_PSModeMngment_inState(MEMORY))

MD.1ii AG (HKCDMS_ask_sat → HKCDMS_PSModeMngment_inState(SAT))

Mode Management in HK CDMS

MD.1 AG (HKCDMS_read_HK → HKCDMS_ModeMngment_inState(ENABLED))

Mutual Exclusion management in HK CDMS

MX.1 AG (¬ (HKCDMS_reader1_inState(READ) ∧ HKCDMS_reader2_inState(READ)))

3.5.7 Properties of the Flash Memory subsystem

Mode Management in Flash Memory Read

MD.1 AG (FlashMem_read → MEMRD_ModeMngment_inState(READ))

Mode Management in Flash Memory Write

MD.1 AG (FlashMem_write → MEMWR_ModeMngment_inState(WRITE))

Mutual Exclusion management in Flash Memory

MX.1 AG (¬ (data_transfer_proc_inState(MEM) ∧ HKPL_proc_inState(WAIT) ∧ HKEPS_proc_inState(WAIT) ∧ HKCOM_proc_inState(WAIT) ∧ HKCDMS_proc_inState(WAIT)))

3.5.8 Properties of the CDMS status subsystem

Action Flow in CDMS status

AF.1 AG (CDMS_ActFlow_start → AX A [¬ CDMS_I2C_ask W reset_done])

AF.2i AG (internal_watchdog → AX A [¬ internal_watchdog W CDMS_ActFlow_start])

AF.2ii AG (CDMS_I2C_ask → AX A [¬ CDMS_I2C_ask W CDMS_ActFlow_start])

AF.3 AG (CDMS_ActFlow_start → AX A [¬ CDMS_ActFlow_finish W CDMS_I2C_res])

3.5.9 Properties of the Error Logging subsystem

Mutual Exclusion management in Error Logging

MX.1 AG (¬ (I2C_SAT_proc_error1 ∨ I2C_SAT_proc_error2 ∨ I2C_SAT_proc_error3)
∧ (dataTransfer_proc_mem_res)
∧ (HKCDMS_proc_mem_res)
∧ (HKPL_proc_mem_res ∨ HKPL_proc_I2C_fail_PL)
∧ (HKEPS_proc_mem_res ∨ HKEPS_proc_I2C_fail_EPS)
∧ (HKCOM_proc_mem_res ∨ HKCOM_proc_I2C_fail_COM)
∧ (MEMRD_proc_fail ∨ MEMRD_proc_ok_read ∨ MEMRD_proc_bad_CRC)
))

3.6 Model verification

Recall (Sect. 2) that safety properties imposed by architectures are preserved by architecture composition [2]. Thus, all properties that we have associated to the CubETH requirements are satisfied *by construction* by the complete model of the case study example.

Architectures enforce properties by restricting the joint behaviour of the operand components. Therefore, combined application of architectures can generate deadlocks. We have used the `D-Finder` tool [7] to verify deadlock-freedom of the case study model. `D-Finder` applies compositional verification on BIP models by over-approximating the set of reachable states, which allows it to analyse very large models. The tool is sound, but incomplete: due to the above mentioned over-approximation it can produce false positives, i.e. potential deadlock states that are unreachable in the concrete system. However, our case study model was shown to be deadlock-free without any potential deadlocks. Thus, no additional reachability analysis was needed.

3.7 Validation of the approach

The key advantage of our architecture-based approach is that the burden of verification is shifted from the final design to architectures, which are considerably smaller in size and can be reused. In particular, all the architecture styles that we have identified for the case study are simple. Their correctness—enforcing the characteristic properties—can be easily proved by inspection of the coordinator behaviour. However, in order to increase the confidence in our approach, we have conducted additional verification, using `nuXmv` to verify that the characteristic properties of the architectures are, indeed, satisfied. We used the `BIP-to-NuSMV` tool[4] to translate our BIP models into NuSMV—the `nuXmv` input language [10].

Verification of the complete model with `nuXmv` did not succeed, running out of memory after four days of execution. Thus, we repeated the procedure (BIP-to-NuSMV translation and verification using `nuXmv`) on individual sub-systems. All connectors that crossed sub-system boundaries

[4]`http://risd.epfl.ch/bip2nusmv`

Table 2: Statistics of models and verification

Model	**Tool**	**Components**	**Connectors**	**RSS**	**Deadlocks**	**Properties**
CubETH	D-Finder	49	155	-	0	-
Payload	nuXmv	13	42	8851	0	9
I2C_sat	nuXmv	4	12	52	0	1
HK PL	nuXmv	11	12	77274	0	5
HK EPS	nuXmv	11	12	77274	0	5
HK COM	nuXmv	11	12	77274	0	5
HK CDMS	nuXmv	10	9	12798	0	5
Flash Memory	nuXmv	6	15	44	0	3
CDMS status	nuXmv	3	6	8	0	4
Error Logging	nuXmv	2	2	2	0	1

RSS = Reachable State Space

were replaced by their corresponding sub-connectors. This introduces additional interactions, hence, also additional execution branches. Since no priorities are used in the model, this modification does not suppress any existing behaviour. Notice that the CTL properties enforced by the presented architecture styles use only universal quantification (A) over execution branches. Hence, the above approach is a sound abstraction, i.e. the fact that the properties were shown to hold in the sub-systems immediately entails that they also hold in the complete model. The complete list of CTL formulas is presented in [24]. Table 2 presents the complexity measures of the verification, which was carried out on an Intel Core i7 at 3.50GHz with 16GB of RAM. Notice that component count in sub-systems adds up to more than 49, because some components contribute to several sub-systems.

4 Related work

The European Space Agency (ESA) advocates a model-based design flow rather than a document-centric approach. To this end, a series of funded research initiatives has delivered interesting results that are worth mentioning. The Space Avionics Open Interface Architecture (SAVOIR)[5] project introduces the On-board Software Reference Architecture (OSRA) [21] that imposes certain structural constraints through the definition of the admissible types of software components and patterns of interaction among their instances. The ASSERT Set of Tools for Engineering (TASTE)[6] [30] is more appropriate for the detailed software design and model-based code generation. In TASTE, the architectural design is captured through a graphical editor that generates a model in the Architecture Analysis & Design Language (AADL). However, the AADL semantics is not formally defined, which inhibits it from being used for rigorous design or formal verification purposes. The Correctness, Modeling and Performance of Aerospace Systems (COMPASS)[7] toolset relies on an AADL variant with formally defined semantics called SLIM and provides means for a posteriori formal verification [13]. A formal semantics for the AADL has been defined in BIP, along with a translation of AADL models into the BIP language [17]. The rigorous design approach based on correct-by-construction steps is applied in the Functional Requirements and Verification Techniques for the Software Reference Architecture (FoReVer)[8] and the Catalogue of System and Software Properties (CSSP) projects. The former initiative advocates a top-down design flow by imposing behavioural contracts on individual components [8], while the latter adopts

[5]`http://savoir.estec.esa.int/`

[6]`http://taste.tuxfamily.org/`.

[7]`http://compass.informatik.rwth-aachen.de/`.

[8]`https://es-static.fbk.eu/projects/forever/`

our architecture-based design flow relying on BIP.

Although a number of frameworks exist for the specification of architectures [26, 28, 36], model design and code generation [1, 6, 12, 34, 35], and verification [11, 16, 18, 22], we are not aware of any that combine all these features. In particular, to the best of our knowledge, our approach is the first application of requirement-driven correct-by-construction design in the domain of satellite on-board software, which relies on requirements to define a high-level model that can be directly used to generate executable code for the satellite control [29].

BIP has previously been used for the design of control software. The applications closest to ours are the initial design of the CubETH [29] and the DALA robot [5] control software. While the latter design followed a predefined software architecture (in the sense of [4]), the former was purely ad-hoc. Neither was driven by a detailed set of requirements.

In [19], the authors describe the interfacing of Temporal Logic Planning toolbox (TuLiP) with the JPL Statechart Autocoder (SCA) for the automatic generation of control software. The TuLiP toolbox generates from statechart models from high-level specifications expressed as formulas of particular form in the Linear Temporal Logic (LTL). SCA is then used to generate Python, C or C++ code from the obtained statecharts. This approach is grounded in formal semantics, it provides correctness guarantees through the automatic synthesis of control behaviour. Furthermore, the transition through statecharts allows the use of graphical tools to visualise the controller behaviour. However, it also has some limitations. Most notably, it focuses exclusively on the synthesis of one controller component and is not easily amenable to the holistic design of complete software systems involving concurrent components.

5 Conclusion and future work

Based on previous work [29], we have analysed the command and data management sub-system (CDMS) of the CubETH nanosatellite on-board software (OBSW), concentrating primarily on safety and modularity of the software. Starting from a set of informal requirements, we have used the architecture-based approach [2] to design a BIP model of the CDMS sub-system. We have illustrated the key steps of the BIP model design, discussed and evaluated the verification and validation procedures.

The architecture-based approach consists in the application of a number of architectures starting with a minimal set of atomic components. Each architecture enforces *by construction* a characteristic safety property on the joint behaviour of the operand components. The combined application of architectures is defined by an associative and commutative operator [2], which guarantees the preservation of the enforced properties. Since, architectures enforce properties by restricting the joint behaviour of the operand components, combined application of architectures can lead to deadlocks. Thus, the final step of the design process consists in verifying the deadlock-freedom of the obtained model. The key advantage of this approach is that the burden of verification is shifted from the final design to architectures, which are considerably smaller in size and can be reused. This advantage is illustrated by our verification results: while model-checking of the complete model was inconclusive, verification of deadlock-freedom took only a very short time, using the `D-Finder` tool.

The main contribution of the presented work is the identification and formal modelling—using architecture diagrams [25]—of 9 architecture styles. Architecture styles represent recurring coordination patterns: those identified in the case study have been reused in the framework of a collaborative project funded by ESA and can be further reused in other satellite OBSW.

The case study serves as a feasibility proof for the use of architecture-based approach in OBSW design for nano- and picosatellites, as well as scientific instruments. The modular nature of BIP allows iterative design for satellites in development and component reuse for subsequent missions.

The automatic generation of C++ code provided by the BIP tool-set enables early prototyping and validation of software functionality even before the hardware platform is completely defined, also contributing to portability of designs. Indeed, the only non-trivial action required in order to use a different target platform is to recompile the BIP engine.

This case study opens a number of directions for future work. In the framework of the ESA project, we are currently developing a tool for the automatic application and composition of architectures and a GUI tool for ontology-based specification of user requirements. We plan to integrate these, together with the BIP framework, into a dedicated tool-chain for OBSW design, providing requirement traceability and early validation. We also plan to expand our taxonomy of architecture styles and study the application of parametrised model checking techniques for their formal verification. Finally, it would be interesting to extend the architecture-based approach to real-time systems. Composability of real-time architectures will require a notion of non-interference similar to that used to ensure the preservation of liveness properties in [2].

Acknowledgements

The work presented in this paper was partially funded by the ESA CSSP project (contract no. 4000112344/14/NL/FE). We would like to thank Andreas Jung, Marcel Verhoef and Marco Panunzio for the instructive discussions in the framework of this project, which have contributed to the clarification of some of the ideas realised in this case study.

References

[1] F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[2] P. Attie et al. A general framework for architecture composability. In *SEFM 2014*, volume 8702 of *LNCS*, pages 128–143. Springer, 2014.

[3] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Soft. Eng. Addison-Wesley Professional, 3rd edition, October 2012.

[5] A. Basu et al. Incremental component-based construction and verification of a robotic system. In *ECAI 2008*, pages 631–635. IOS Press, 2008.

[6] A. Basu et al. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, May 2011.

[7] S. Bensalem et al. D-Finder 2: Towards efficient correctness of incremental design. In *NFM'11*, volume 6617 of *LNCS*, pages 453–458. Springer, 2011.

[8] A. Benveniste et al. Contracts for system design. Research Report RR-8147, INRIA, November 2012.

[9] S. Bliudze and J. Sifakis. The algebra of connectors—structuring interaction in BIP. *IEEE Transactions on Computers*, 57(10):1315–1330, 2008.

[10] S. Bliudze et al. Formal verification of infinite-state BIP models. In *ATVA'15*, volume 9364 of *LNCS*, pages 326–343. Springer, November 2015.

[11] R. Bloem et al. RATSY – A new requirements analysis tool with synthesis. In *CAV'10*, volume 6174 of *LNCS*, pages 425–429. Springer, 2010.

[12] J.-L. Boulanger et al. *SCADE: Language and Applications.* Wiley-IEEE Press, 1st edition, 2015.

[13] M. Bozzano et al. Spacecraft early design validation using formal methods. *Reliability Engineering & System Safety*, 132:20–35, 2014.

[14] C. Brandon and P. Chapin. A SPARK/Ada CubeSat control program. In *Reliable Software Technologies*, volume 7896 of *LNCS*, pages 51–64. Springer, 2013.

[15] California Polytechnic State University. *CubeSat Design Specification Rev. 13*, 2014. Available online: `http://www.cubesat.org/s/cds_rev13_final2.pdf`.

[16] R. Cavada et al. The nuXmv symbolic model checker. In *CAV'14*, volume 8559 of *LNCS*, pages 334–342. Springer, 2014.

[17] Y. Chkouri et al. Translating AADL into BIP — Application to the verification of real-time systems. In *MODELS 2008*, pages 5–19. Springer, 2009.

[18] A. Cimatti, M. Dorigatti, and S. Tonetta. OCRA: A tool for checking the refinement of temporal contracts. In *ASE 2013*, pages 702–705, November 2013.

[19] S. Dathathri et al. Interfacing TuLiP with the JPL Statechart Autocoder: Initial progress toward synthesis of flight software from formal specifications. In *IEEE AeroSpace*, 2016.

[20] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994.

[21] A. Jung, M. Panunzio, and J.-L. Terraillon. On-board software reference architecture. Technical Report TEC-SWE/09-289/AJ, SAVOIR Advisory Group, 2010.

[22] J.-S. Kim and D. Garlan. Analyzing architectural styles with Alloy. In *ROSATEA '06*, pages 70–80. ACM, 2006.

[23] A. Mavridou et al. Configuration logics: Modelling architecture styles. In *FACS 2015*, volume 9539 of *LNCS*, pages 256–274. Springer, 2015.

[24] A. Mavridou et al. Architecture-based Design: A Satellite On-Board Software Case Study. Technical Report 221156, EPFL, September 2016. https://infoscience.epfl.ch/record/221156.

[25] A. Mavridou et al. Architecture diagrams: A graphical language for architecture style specification. In 9th *ICE,*, volume 223 of *EPTCS*, pages 83–97, 2016.

[26] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[27] C. Mitchell et al. Development of a modular command and data handling architecture for the KySat-2 CubeSat. In *2014 IEEE Aerospace Conference*, pages 1–11. IEEE, March 2014.

[28] M. Ozkaya and C. Kloukinas. Are we there yet? Analyzing architecture description languages for formal analysis, usability, and realizability. In *SEAA 2013*, pages 177–184. IEEE, 2013.

[29] M. Pagnamenta. Rigorous software design for nano and micro satellites using BIP framework. Master's thesis, EPFL, 2014. https://infoscience.epfl.ch/record/218902.

[30] M. Perrotin et al. *TASTE: A Real-Time Software Engineering Tool-Chain Overview, Status, and Future*, pages 26–37. Springer, 2012.

[31] S. Rossi et al. CubETH magnetotorquers: Design and tests for a CubeSat mission. In *Advances in the Astronautical Sciences*, volume 153, pages 1513–1530, 2015.

[32] J. Sifakis. Rigorous system design. *Foundations and Trends® in Electronic Design Automation*, 6(4):293–362, 2012.

[33] S. C. Spangelo et al. Model based systems engineering (MBSE) applied to Radio Aurora Explorer (RAX) CubeSat mission operational scenarios. In *2013 IEEE Aerospace Conference*, pages 1–18. IEEE, mar 2013.

[34] SysML. `http://www.sysml.org`.

[35] L. K. Wells and J. Travis. *LabVIEW for Everyone: Graphical Programming Made Even Easier*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[36] E. Woods and R. Hilliard. Architecture description languages in practice session report. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA05)*, pages 243–246. IEEE Computer Society, 2005.