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

Anastasia Mavridou[1], Emmanouela Stachtiari[2], Simon Bliudze[1], Anton Ivanov[1], Panagiotis Katsaros[2], and Joseph Sifakis[1]

[1] École polytechnique fédérale de Lausanne, Switzerland; firstname.lastname@epfl.ch
[2] Aristotle University of Thessaloniki, Greece; {emmastac,katsaros}@csd.auth.gr

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*. We illustrate the following three steps of the design approach: 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 DFinder from the BIP tool-set. 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 by the model.

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[3] 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.

[3] CubeSat [**?**] is a standard for the design of nano- and picosatellites.

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 [**?**]. Other projects simply structure their code in C/C++ and then extensively test it, maybe using some analysis tools such as `lint` [**?**]. Others use SysML [**?**] to describe the system as a whole [**?**] 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 [**?**,**?**], imposing behavioural contracts on individual components [**?**] or by applying automatic transformations to obtain executable code from formally defined high-level models [**?**].

Following this latter approach, a notion of *architectures* was proposed in [**?**] 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* [**?**,**?**] 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.

The notion of architectures proposed in [**?**] is based on the Behaviour-Interaction-Priority (BIP) [**?**] 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 [**?**], state reachability analysis and an interface with the `nuXmv` model checker [**?**]. In the CubETH project [**?**], BIP was used to design logic for the operation of a satellite, executed on the on-board computer [**?**]. 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 paper, we have analysed the BIP model obtained in [**?**] 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 paper 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 identified architecture styles, illustrates our approach through the design of a corresponding BIP model and presents the verification process and results. Section 4 discusses the related work. Section 5 concludes the paper.

2 Architecture-based design approach

Our approach relies on the BIP framework [**?**] 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 [**?**]. Third, *priorities* are used to impose scheduling constraints and to resolve conflicts when multiple interactions are enabled simultaneously.

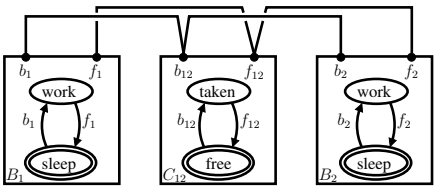


Fig. 1: Mutual exclusion model in BIP

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 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.

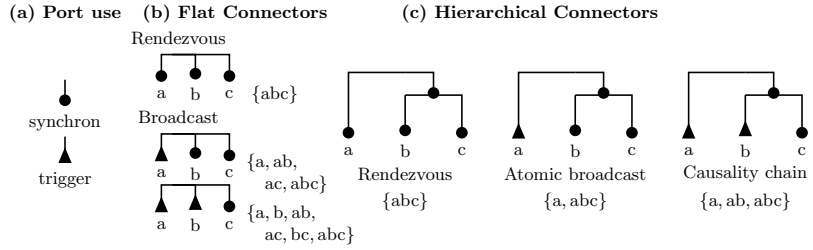


Fig. 2: Flat and hierarchical BIP connectors

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*.[4] Figure 3 shows an architecture that enforces the mutual exclusion property $\mathtt{AG}\neg(cs_1 \wedge cs_2)$ on any two components with interfaces $\{b_1, f_1\}$ and $\{b_2, f_2\}$, satisfying the CTL formula $\mathtt{AG}\big(f_i \rightarrow \mathtt{A}[\neg cs_i \ \mathtt{U} \ b_i]\big)$, 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 '$\oplus$' [**?**]. 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.

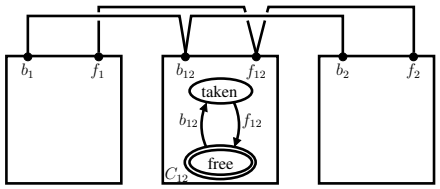


Fig. 3: Mutual exclusion architecture

Although the architecture in Fig. 3 can only be applied to a set of precisely two components, it 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* [**?**] to specify such *architecture styles*, as described in the next section. (See Fig. 6 in Sect. 3.1 for the diagram of the style generalising the architecture in Fig. 3.)

[4] The precise definition of the refinement relation is beyond the scope of this paper.

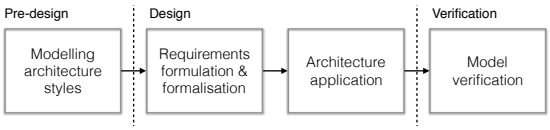


Fig. 4: Architecture-based design flow

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* [**?**] 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 '$\texttt{sc}[x, y]$' to mean that the same multiplicity or degree is applied to each port instance of p. We write '$\texttt{mc}[x, y]$' 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 [**?**] 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 $\texttt{E}[\cdot\,\texttt{U}\,\cdot]$, $\texttt{A}[\cdot\,\texttt{U}\,\cdot]$, $\texttt{E}[\cdot\,\texttt{W}\,\cdot]$, $\texttt{A}[\cdot\,\texttt{W}\,\cdot]$ (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

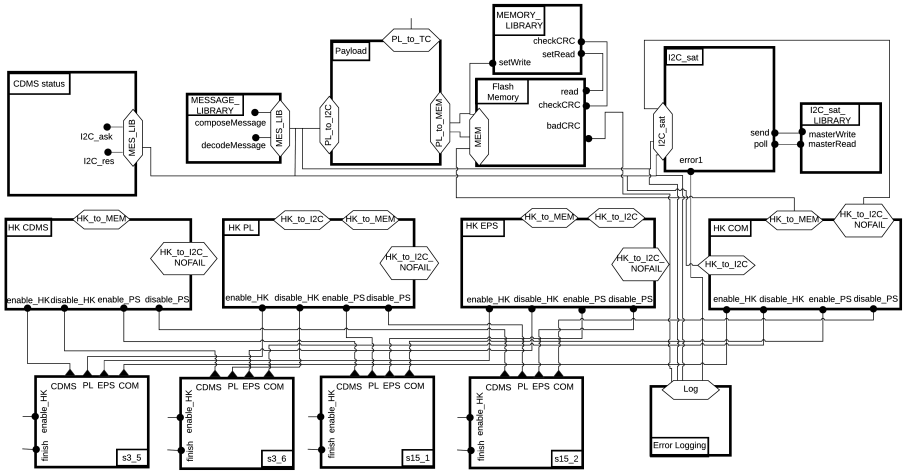


Fig. 5: The high-level interaction model

must hold. The intuition behind the letters is the following: the branch quantifiers are `A` (for "All") and `E` (for "Exists"); the temporal modalities 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 [**?**]. 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` 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 can be found in the technical report [**?**]. It 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

We have identified 9 architecture styles from the BIP model obtained in [**?**]. In this section, we present 5 styles (all styles are presented in the technical report [**?**]). 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 this case study, all characteristic properties are safety properties. Due to space limitations, in the next subsections, for all architecture styles except for Mutual exclusion, we omit their assumed properties. These can be found in the technical report [**?**].

The styles are specified by using architecture diagrams. Below, for the sake of clarity, we omit the port type cardinality if it is equal to 1. The cardinality of a component type is indicated right next to its name.

The Mutual exclusion style (Fig. 6) generalises the architecture in Fig. 3. It enforces mutual exclusion on a shared resource (see Sect. 2).

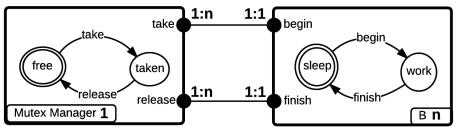


Fig. 6: Mutual exclusion style

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, hence, 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. The assumed and characteristic properties of this style were presented in Sect. 2.

The Client-Server style (Fig. 7) 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` and `offer2`. The `Client` has two port types `use` and `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`.

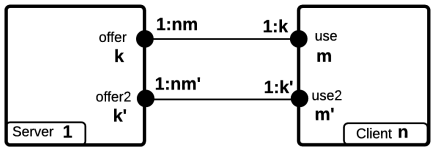


Fig. 7: Client-Server style

Two connector motifs connect `use` (resp. `use2`) with `offer` (resp. `offer2`). The multiplicity:degree constraints of `offer` and `use` are $1 : n\,m$ and $1 : k$, respectively. Since both multiplicities are 1, all connectors are 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`. An architecture of this style is shown in Fig. 12.

The characteristic property of this style is '*only one client can use a provided service at each time*', formalised by the CTL formula:

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

The Action flow style (Fig. 8) enforces a sequence of actions. It has one coordinator component of type `Action Flow Manager` and n parameter components of type `B`. The cyclic behaviour of the coordinator enforces an order on the actions of the operands. In the manager's behaviour, `abi` and `aei` stand for "action i begin" and "action i end".

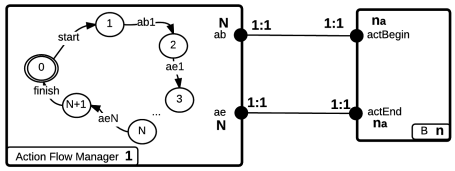


Fig. 8: Action flow style

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. An architecture of this style is shown in Fig. 11.

The characteristic property of this style 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*', 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 $seq_c(i)$.

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

The Failure monitoring style (Fig. 9) 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.

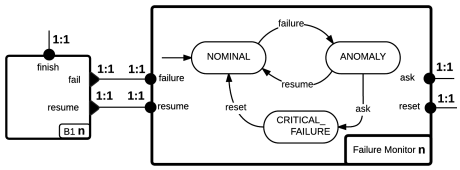


Fig. 9: Failure monitoring style

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. 13.

The characteristic property of this style is '*if a failure occurs, a finish happens only after a resume or reset*', formalised by the following CTL formula:

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

The Mode management style (Fig. 10) restricts the set of enabled actions 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.

`Mode Manager` has k states—one state per mode—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 `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

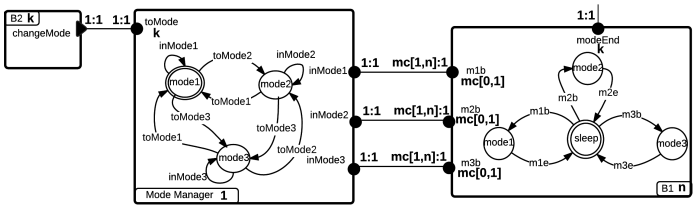


Fig. 10: Mode management style (component behaviour is shown for k=3)

Table 1: Representative requirements for CDMS status and HK_PL

ID	Description
CDMS-007	The CDMS shall periodically reset both the internal and external watchdogs and contact the EPS subsystem with a "heartbeat".
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.

type with cardinality k. `mib` stands for "mode i begin" and indicates that an action 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 `B1` components through an n-ary connector. An architecture of this style is shown in Fig. 14.

The characteristic property of this style is '*an action is only performed in a mode where it is allowed*', formalised by the following CTL formula:

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

3.2 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 CubETH functional requirements (Tab. 1). We additionally present the result of the composition of Client-Server and Mode management architectures. The full list of requirements is provided in [**?**].

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

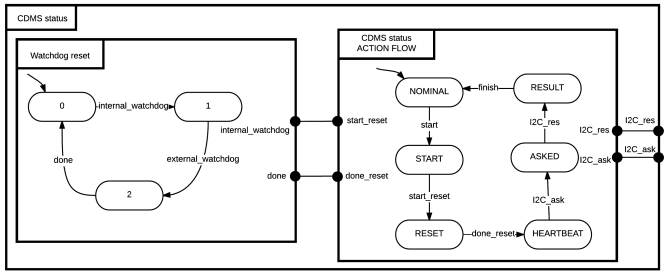
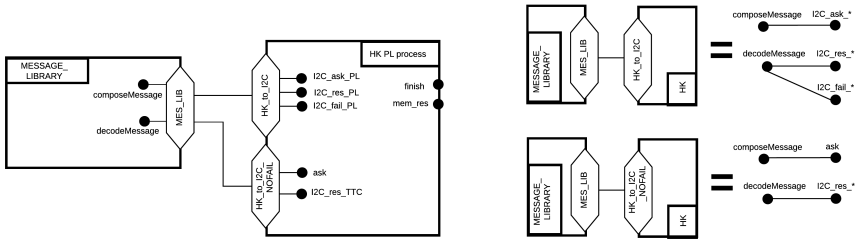


Fig. 11: Application of Action flow architecture



(a) Architecture application (b) Hexagons of Fig. 12a

Fig. 12: Application of Client-Server architecture

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. 12a, 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. 12a, we use hexagons to group interaction patterns of components. The meaning of these hexagons is explained in Fig. 12b.

Application of Failure monitoring architecture Requirement HK-005, presented in Tab. 1, suggests the application of the Failure monitoring architecture as shown in Fig. 13. The BIP model comprises the `HK PL process` operand and the `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 commu-

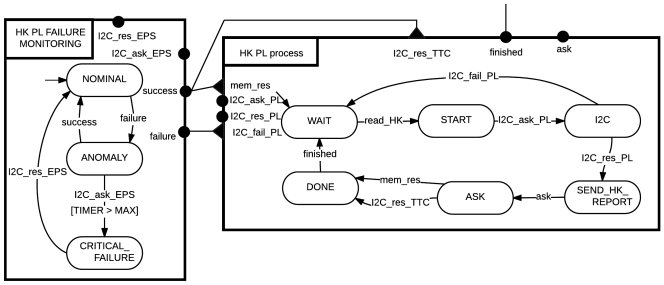


Fig. 13: Application of Failure monitoring architecture

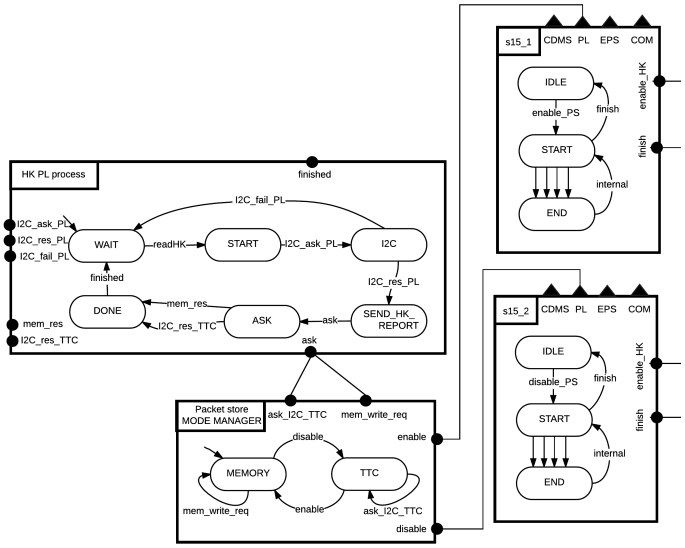


Fig. 14: Application of Mode management architecture

nication 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.

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. 14, 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.

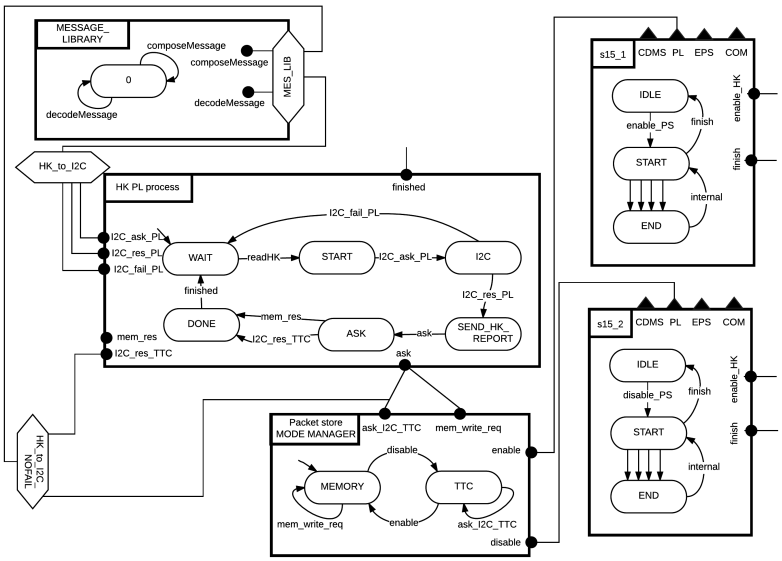


Fig. 15: Composition of Client-Server and Mode management architectures

Composition of architectures The architecture composition was formally defined in [**?**]. 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. 15 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. 12) 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. 14) 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. 15).

3.3 Model verification

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

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 [**?**] 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.

3.4 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[5] to translate our BIP models into NuSMV—the `nuXmv` input language [**?**].

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 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 [**?**]. 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)[6] project introduces the Onboard Software Reference Architecture (OSRA) [**?**] 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)[7] [**?**] 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

[5] `http://risd.epfl.ch/bip2nusmv`
[6] `http://savoir.estec.esa.int/`
[7] `http://taste.tuxfamily.org/`.

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

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)[8] toolset relies on an AADL variant with formally defined semantics called SLIM and provides means for a posteriori formal verification [**?**]. A formal semantics for the AADL has been defined in BIP, along with a translation of AADL models into the BIP language [**?**]. 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)[9] 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 [**?**], while the latter adopts our architecture-based design flow relying on BIP.

Although a number of frameworks exist for the specification of architectures [**?**,**?**,**?**], model design and code generation [**?**,**?**,**?**,**?**], and verification [**?**,**?**,**?**], 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 [**?**].

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

In [**?**], 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

[8] `http://compass.informatik.rwth-aachen.de/`.
[9] `https://es-static.fbk.eu/projects/forever/`

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 [**?**], 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 [**?**] 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 [**?**], 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 [**?**]—of 9 architecture styles, whereof 5 are presented in the paper (all styles are presented in the associated technical report [**?**]). 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 satellite OBSW design. 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. The most immediate consists in studying optimisation techniques, such as [**?**] to reduce the complexity overhead of the automatically generated models. 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 [**?**].

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. Finally, we are deeply grateful to the anonymous reviewers for their constructive comments.