Verification of concurrent design patterns with data

Simon Bliudze, Ludovic Henrio, Eric Madelaine

Verification of concurrent design patterns with data

Simon Bliudze[1], Ludovic Henrio[2], and Eric Madelaine[3]

[1] Inria Lille – Nord Europe, Villeneuve d'Ascq, France
simon.bliudze@inria.fr
[2] Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
ludovic.henrio@ens-lyon.fr
[3] Université Côte d'Azur, Inria, CNRS, I3S, 06902 Sophia-Antipolis, France
eric.madelaine@inria.fr

Abstract. We provide a solution for the design of safe concurrent systems by compositional application of verified design patterns—called *architectures*—to a small set of functional components. To this end, we extend the theory of architectures developed previously for the BIP framework with the elements necessary for handling data: definition and operations on data domains, syntax and semantics of composition operators involving data transfer. We provide a set of conditions under which composition of architectures preserves their characteristic safety properties. To verify that individual architectures do enforce their associated properties, we provide an encoding into open pNets, an intermediate model that supports SMT-based verification. The approach is illustrated by a case study based on a previously developed BIP model of a nanosatellite on-board software.

Keywords: Symbolic verification, composition, safety, interaction models.

1 Introduction

BIP (Behaviour-Interaction-Priority) [7] is a framework for the component-based design of concurrent software and systems. In particular, the BIP tool-set comprises compilers for generating C/C++ code, executable by linking with one of the dedicated engines, which implement the BIP operational semantics [14]. BIP ensures that any property that holds on a BIP model will also hold on the generated code. The notion of BIP *architecture* was proposed in [5] as a mechanism for ensuring correctness by construction during the design of BIP models. Architectures can be viewed as operators transforming BIP models. They formalise design patterns, which enforce global properties characterising the coordination among the components of the system. The architecture-based design process in BIP takes as input a set of components providing basic functionality of the system and a set of temporal properties that must be enforced in the final system. For each property, a corresponding architecture is identified and applied to the

model, adding coordinator components and modifying the synchronisation patterns between components. In [5], it was shown that application of architectures is compositional w.r.t. safety properties, i.e. if two architectures guarantee two properties, their composition ensures the conjunction of the properties but [5] did not consider properties depending on data.

This article goes one step further in the proof of properties and in the compositionality of architectures, but this step is a significant one: the compositional verification. To prove properties of BIP architectures it is necessary to have a representation of the BIP architecture in a verifiable format. The verification problem has two unbounded parameters: 1) By nature, architectures have holes and are meant to interact with the interfaces of the component that will fill the hole; the properties must hold for all (well-typed) components that can be put inside the hole; 2) BIP interactions can transmit data, and properties might be dependent of the data, the domain of the data is generally huge or unbounded and the values of transmitted data might have a significant impact on the properties. We propose to rely on a translation of BIP architectures into *open pNets*.

Parameterised Networks of synchronised automata (pNets) is a formalism for defining behavioural specification of distributed systems based on a parameterised and hierarchical model. It inherited from the work of Arnold on synchronisation vectors [3]. It has been shown in previous work [28] that pNets can represent the behavioural semantics of a system including value-passing and many kinds of synchronisation methods, including various constructs and languages for distributed objects. The VerCors platform uses pNets to design and verify distributed software components [19,27]. There is no bound on the number of elements inside a pNets or the valuation of parameters. When restricted to finite instantiations, it becomes possible to us pNets for finite model-checking approaches. Closed pNets were used to encode fully defined programs or systems, while open pNets have "holes", playing the role of process parameters. Such open systems can represent composition operators or structuring architectures. It is possible to reason, in an SMT engine, on the symbolic automaton that represents the behaviour of a pNets with holes and that communicates values [36]. The encoding of open pNets into Z3 that is under development is the starting point of this article. We benefit from the possibility to reason on a pNet in an SMT engine in order to prove properties on BIP architectures.

The main contributions of this paper are: 1) The addition of data to the theory of BIP architectures, including a theorem about preservation of data dependent properties by compositions. 2) An encoding of architectures with data into open pNets, allowing for analysis of their temporal properties using pNet's software tools. The paper is illustrated by a running example based on the failure monitor architecture from the CubETH nanosatellite on-board software [34]. This running example also relies on the *maximal progress* assumption, whereby larger interactions are preferred to smaller ones. Due to space limitations, we only discuss this informally. However, proofs of the results provided in the appendix formally account for maximal progress.

The rest of the paper is structured as follows. In Sect. 2, we present notations and background material on pNets. The theory of architectures with data is presented in Sect. 3. In Sect. 4, we present the encoding into open pNets and discuss verification of the running example. Section 5 discusses related work. Section 6 concludes the paper.

2 General Notations and pNets Previous Results

Notations. We extensively use indexed structures over some countable indexed sets, which are equivalent to mappings over the countable set. Thus, $a_i^{i\in I}$ denotes a family of elements a_i indexed over the set I. This notation defines both I the set over which the family is indexed (called *range*), and a_i the elements of the family. E.g., $a^{i\in\{3\}}$ is the mapping with a single entry a at index 3 ; also abbreviated $(3{\mapsto}a)$. When this is not ambiguous, we shall use notations for sets, and typically write "indexed set over I", even though formally we should speak of maps; and write $x \in a_i^{i\in I}$ to mean $\exists i \in I.\, x = a_i$. An empty family is denoted $\emptyset$.

We assume the existence of a term algebra $\mathcal{T}_{\Sigma,\mathcal{V}}$, where Σ is the signature of the data and action constructors, and $\mathcal{V}$ a set of *variables*. Within $\mathcal{T}_{\Sigma,\mathcal{V}}$, we distinguish a set of *data expressions* $\mathbb{E}_\mathcal{V}$, e ranges over expressions; and a set of *Boolean expressions* $\mathbb{B}_\mathcal{V} \subseteq \mathbb{E}_\mathcal{V}$, g (guards) ranges over Boolean expressions. On top of $\mathbb{E}_\mathcal{V}$ we build the *action algebra* $Act_\mathcal{V}$, with $Act_\mathcal{V} \subseteq \mathcal{T}_{\Sigma,\mathcal{V}}$. We define $\mathbb{A}_\mathcal{V}$ as the set of variable assignments of the form: $(x_i := e_i)^{i\in I}$ and let u range over sets of assignments. The function $vars(t)$ identifies the set of variables in a term.

We assume the existence of a universal data domain given as a partially-ordered set $(\mathbb{D}, \leqslant)$, potentially encompassing several copies of any given data type with different orders. We assume that $(\mathbb{D}, \leqslant)$ comprises both the unordered set of Booleans $\mathbb{B} = (\{\mathtt{tt}, \mathtt{ff}\}, \emptyset)$ and the naturally ordered one $\mathbb{B}^\leqslant = (\{\mathtt{tt}, \mathtt{ff}\}, \{\mathtt{ff} \leqslant \mathtt{tt}\})$, and similarly for integer and real numbers; as well as the set of intervals ordered by inclusion. When speaking of an ordered sort, e.g. $\mathbb{B}^\leqslant$, we will assume that it forms a meet-semilattice and denote by $\wedge$ the meet operator.

For a set of variables $V \subseteq \mathcal{V}$, we denote $\mathbb{D}^V \stackrel{def}{=} \{\sigma : V \to \mathbb{D}\}$ the set of *valuations* of the variables in V and let σ range over valuations. Valuations extend canonically to expressions, denoted $\sigma(e)$. We define:

$$\sigma\left[(x_i := e_i)^{i\in I}\right](x) \stackrel{def}{=} \begin{cases} \sigma(x), & \text{if } x \notin x_i^{i\in I}\,, \\ \sigma(e_i), & \text{if } x = x_i, \text{for some } i \in I\,. \end{cases}$$

For two valuations $\sigma^1, \sigma^2 : V \to \mathbb{D}$, we denote $\sigma^1 \triangle \sigma^2 \stackrel{def}{=} \left\{x \in V \,\middle|\, \sigma^1(x) \neq \sigma^2(x)\right\}$ the set of variables that are assigned different values by the two valuations. As usual, we write $\sigma^1 \leqslant \sigma^2$ iff $\sigma^1(x) \leqslant \sigma^2(x)$, for all $x \in V$. An expression e is *monotonic* if, for any two valuations σ^1, σ^2, $\sigma^1 \leqslant \sigma^2$ implies $\sigma^1(e) \leqslant \sigma^2(e)$. Similarly, an assignment $(x_i := e_i)^{i\in I}$ is monotonic if all expressions $e_i^{i\in I}$ are monotonic. We denote $\mathbb{B}_V^\leqslant \subset \mathbb{B}_V$, $\mathbb{E}_V^\leqslant \subset \mathbb{E}_V$ and $\mathbb{A}_V^\leqslant \subset \mathbb{A}_V$ the sets of monotonic Boolean and generic expressions and assignments, respectively.

Open pNets. This section briefly describes pNets, see [29] for more complete description. pNets are tree-like structures, where the leaves are either *parameterised labelled transition systems (pLTSs)*, expressing the behaviour of basic processes, or *holes*, used as placeholders for unknown processes. Nodes of the tree are synchronising artefacts using a set of *synchronisation vectors* that express the possible synchronisation between parameterised actions of some components.

A pLTS is a labelled transition system with variables occurring inside states, actions, guards, and assignments. Variables of each state are pairwise disjoint. Each transition label of a pLTS consists of a parameterised action, a guard and an update assignment. The parameters of actions are either input variables or expressions. Input variables are bound when the action occurs; they accept any value (of the correct type), thus providing a to input data from the environment. Expressions are computed from the values of other variables. They allow providing aggregated values to the environment, without exposing all the underlying variables. We define the set of parameterised actions a pLTS can use (a ranges over action labels): $\alpha = a(?x_i^{i\in I}, e_j^{j\in J})$, where $?x_i^{i\in I}$ are input variables, $e_j^{j\in J}$ are expressions.

Definition 1 (pLTS). *A pLTS is a tuple* $pLTS \triangleq \langle\!\langle S, s_0, \rightarrow \rangle\!\rangle$ *where:* S *is a set of states;* $s_0 \in S$ *is the initial state;* $\rightarrow \subseteq S \times L \times S$ *is the transition relation and* L *is the set of labels of the form* $\langle \alpha,\ g,\ u \rangle$*, where* α *is a parameterised action,* $\alpha \in Act_{\mathcal{V}}$*;* $g \in \mathbb{B}_{\mathcal{V}}$ *is a guard over variables of the source state and the action, and* $u \in \mathbb{A}_{\mathcal{V}}$ *assigns updated value for variables in the destination state.*

A pNet composes several pNets, pLTSs, and holes. A pNet exposes global actions resulting from the synchronisation of internal actions in some sub-pNets, and some actions of the holes. As holes are process parameters, synchronisation with a hole has an obvious concrete meaning when a process is put inside the hole and emits the action. We also define a semantics for open pNets with holes where open transitions express the fact that a pNet can performs a transition provided one or several holes emit some actions. This synchronisation is specified by *synchronisation vectors* expressing the synchronous interaction between actions inside sub-pNets and holes, data transmission is expressed classically using action parameters. Actions involved in the synchronisation vectors do not need to distinguish input variables, i.e. they have the form $a(Expr_j^{j\in J})$.

Definition 2 (pNets). *A pNet is a hierarchical structure where leaves are pLTSs and holes:* $Q \triangleq pLTS \mid \langle\!\langle Q_i^{i\in I}, J, SV_k^{k\in K} \rangle\!\rangle$ *where*

- $Q_i^{i\in I}$ *is the family of sub-pNets;*
- J *is a set of indexes, called* holes*.* I *and* J *are* disjoint*:* $I \cap J = \emptyset$*,* $I \cup J \neq \emptyset$
- $SV_k^{k\in K}$ *is a set of synchronisation vectors.* $\forall k \in K, SV_k = \alpha_l^{l\in I_k \uplus J_k} \rightarrow \alpha'_k[g_k]$*, where* $\alpha'_k \in Act_{\mathcal{V}}$*,* $I_k \subseteq I$*,* $J_k \subseteq J$*, and* $vars(\alpha'_k) \subseteq \bigcup_{l\in I_k \uplus J_k} vars(\alpha_l)$*. The global action is* α'_k*,* g_k *is a guard associated to the vector.*

The set of holes $\mathrm{Holes}(Q)$ *of a pNet is the indexes of the holes of the pNet itself plus the indexes of all the holes of its subnets (we suppose those indexes disjoints). A pNet* Q *is* closed *if it has no hole:* $\mathrm{Holes}(Q) = \emptyset$*; else it is said*

to be open. *The set of* leaves *of a pNet is the set of all pLTSs occurring in the structure, as an indexed family of the form* $\text{Leaves}(Q) = \langle\!\langle pLTS_i \rangle\!\rangle^{i \in L}$.

The semantics of an open pNet is expressed as an automaton where each transition coordinates the actions of several holes, the transition occurs if some predicates hold, and can involve state modifications.

Definition 3 (Open transition). *An* open transition *over a set of holes* J *and a set of states* $\mathcal{S}$ *is a structure of the form:*

$$\frac{\beta_j^{j \in J}, g, u}{s \xrightarrow{\alpha} s'}$$

Where $s, s' \in \mathcal{S}$ *and* $\beta_j \in Act_{\mathcal{V}}$ *is an action of the hole* j*;* α *is the resulting global action;* g *is a predicate over the different variables of the terms, labels, and states* β_j*,* s*,* α*.* $u \in \mathbb{A}_{\mathcal{V}}$ *is a set of assignments that are the effects of the transition. Open transitions are identified modulo logical equivalence on their predicate.*

The red dotted rule expresses the implication stating that if the holes perform the designated actions and the condition g is verified, then the variables are modified and the state changes. This implication however uses a simple logic with the expressive power given by the predicate algebra (it must include logical propositions and equality). Proposition and inference rules of the paper use a standard logic, while predicates inside the open transitions should use a more restricted logic, typically a logic that could be handled mechanically and expressed by terms that can be encoded in a simple syntax. Open transitions express in a symbolic way, transitions that are not only parameterised with variables but also actions of not yet known processes.

Definition 4 (Open automaton). *An* open automaton *is a tuple* $(J, \mathcal{S}, s_0, \mathcal{T})$ *where:* J *is a set of indices,* $\mathcal{S}$ *is a set of states and* s_0 *an initial state among* $\mathcal{S}$*,* $\mathcal{T}$ *is a set of open transitions and for each* $t \in \mathcal{T}$ *there exist* J' *with* $J' \subseteq J$*, such that* t *is an open transition over* J'*, and* $\mathcal{S}$*.*

The semantics of an open pNet is an open automaton where the states are tuples of states of the pLTSs at the leaves, denoted $\triangleleft \ldots \triangleright$. Each open transition between two states contains 1) the actions of the holes involved in the transition, 2) a guard built from the synchronisation vectors coordinating the holes and the transitions involved; 3) assignments and global state change defined by the pLTSs transitions involved; 4) a global action defined by the synchronisation vector.

Example 1 (An open transition). The open transition

$$\frac{\{E \mapsto ask\}, t \in z, \{t := t + 1\}}{\triangleleft 11 \triangleright \xrightarrow{ask} \triangleleft 11 \triangleright}$$

emits a global action ask defined by the synchronisation vector $\langle timeout_T, timeout_C, -, ask \rangle \rightarrow ask$. It requires the hole at label E to fire an ask action, with the condition $t \in z$. In this case, the global pNet loops on the state $\triangleleft 11 \triangleright$ that

has internal variables t and z local to the pLTS T (hence not appearing in the synchronisation vector). The variable t is updated to the new value $t+1$. Figure 2 shows the complete pNet, whereas Fig. 3 shows a complete open automaton.

We used pNets to define a behavioural semantics for distributed components [2] that allows the verification of correctness properties by model-checking. More recently, a bisimulation theory has been formalised for open pNets [29].

3 The theory of architectures with data

This section presents the extension of the theory of architectures [5] with data and briefly discusses a special case of priority models, called *maximal progress*. These extensions require us to define the framework in a manner that would allow formulating and proving the property preservation result (Th. 1 below). In [5], this result is obtained by requiring, in the definition of architecture composition, that an interaction among coordinated components be only possible if both architectures "agree" that it should be enabled. With respect to data, the main difficulty lies in ensuring that this "agreement" extends to the transferred data values. A trivial extension would allow an interaction only if the data values proposed by both architectures coincide. As this requirement is too restrictive, we go beyond by assuming the data domains to be ordered and taking the *meet* of the proposed values. The property preservation result then holds independently of the proposed values, provided that guards and update assignments are monotonic.

An important insight is that, although the requirement that guards and update assignments be monotonic appears to be a limitation, it is, in fact, a generalisation of the usual setting. Indeed, the usual settings, where data domains are not ordered, can be recovered here by considering trivial partial orders with no two distinct elements being comparable. In such case, *all expressions* are trivially monotonic.

The intuition behind the proof of the preservation of safety properties in [5] is simple. The composition of two architectures combines the "constraints" that they impose on the possible executions of the system: as stated above, an interaction is only enabled if both architectures "agree". In [6], it is shown that this intuition extends well to priorities in the *offer semantics* of BIP. However, this is not the case in the classical semantics. In this section, we informally discuss the special case of the maximal progress priority models, where property preservation does hold in the classical semantics of BIP.

Components and composition

Definition 5 (Component). *A component is a tuple* $(Q, q^0, V, \sigma^0, P, \varepsilon, \rightarrow)$, *where*

- Q *is a set of* states, *with* $q^0 \in Q$ *the* initial state,
- V *is a set of* component variables,

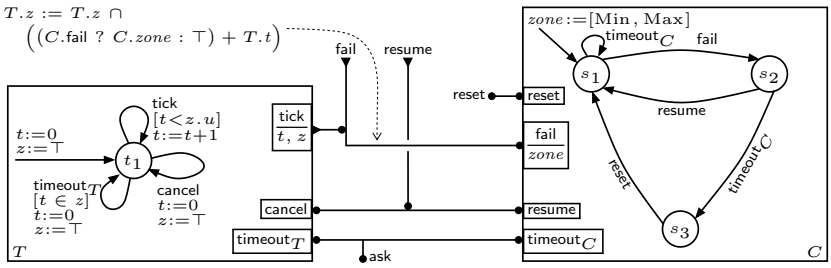


Fig. 1: The BIP specification of the Failure Monitor architecture

- $\sigma^0 : V \to \mathbb{D}$ *is an* initial valuation *of the component variables,*
- P *is a set of* ports*;* $\varepsilon : P \to 2^V$ *is the* set of variables exported by each port*,*
- $\to \subseteq Q \times (2^P \setminus \{\emptyset\}) \times \mathbb{B}_V^{\leqslant} \times \mathbb{A}_V^{\leqslant} \times Q$ *is a* transition relation*, with transitions labelled by* interactions*, i.e. triples consisting of a non-empty set of ports, a monotonic Boolean* guard *and a monotonic* update assignment*.*

We call the triple (V, P, ε) *the* interface *of the component.*[4] *Notations* $q \xrightarrow{a,g,u} q'$ *and* $q \xrightarrow{a,g,u}$ *are as usual; for a component* B*, we denote* Q_B*,* q_B^0*,* V_B*,* σ_B^0*,* P_B*, and* ε_B *the corresponding constituents of* B*. We will skip the index on the transition relations* $\to$*, since it is always clear from the context.*

In this paper, we use a refined version of the Failure Monitor architecture from [34] as a running example. Although Fig. 1 shows the full definition of this architecture, we will explain its various elements progressively. Fig. 1 shows components T(imer) and C(ontrol), with interfaces $\big(\{t, z\}, \{\mathsf{tick}, \mathsf{cancel}, \mathsf{timeout}_T\}, \{\mathsf{tick} \mapsto \{t, z\}\}\big)$, and $\big(\{zone\}, \{\mathsf{reset}, \mathsf{fail}, \mathsf{resume}, \mathsf{timeout}_C\}, \{\mathsf{fail} \mapsto \{zone\}\}\big)$ respectively. Variable t is implicitly assumed to be of type Integer (with trivial ordering). Variables $z \stackrel{def}{=} [z.l, z.u]$ and $zone \stackrel{def}{=} [zone.l, zone.u]$ are of type Integer Interval ordered by interval inclusion.

Component *behaviour* is defined by states and transitions. The initial states t_1 and s_1, and valuations $\sigma_T^0 = \{t \mapsto 0, z \mapsto \top\}$, $\sigma_C^0 = \{zone \mapsto [\mathrm{Min}, \mathrm{Max}]\}$ are shown by the incoming arrows $\xrightarrow{t:=0, z:=\top} t_1$ and $\xrightarrow{zone:=[\mathrm{Min},\mathrm{Max}]} s_1$ where $\top = (-\infty, +\infty)$. The constants Min and Max are the parameters of the architecture.

Transitions are labelled with ports of the corresponding components, Boolean guards and update assignments on local variables. E.g., the loop transition $t_1 \xrightarrow{\mathsf{tick}, [t<z.u], t:=t+1} t_1$. The guards and update assignments of the transitions of C are omitted. By default, an omitted guard is $\mathtt{tt}$ and an omitted assignment is empty $\emptyset$. Clearly, all guards and update assignments are monotonic.

[4] Only exported variables, belonging to a $\varepsilon(p)$, appear in the component interface (see Def. 7). We omit here this separation between internal and exported variables.

Definition 6 (Component semantics). *The* open semantics *of a component* $B = (Q, q^0, V, \sigma^0, P, \varepsilon, \to)$ *is the LTS denoted* $[B] = (S, s^0, \to)$*, where* $S = Q \times \mathbb{D}^V$*,* $s^0 = (q^0, \sigma^0)$ *and* $\to$ *is the minimal transition relation satisfying the rule*

$$\frac{q \xrightarrow{a,g,u} q' \qquad \sigma \models g \qquad \sigma' = \tilde{\sigma}[u] \qquad \sigma \triangle \tilde{\sigma} \subseteq \varepsilon(a)}{(q,\sigma) \xrightarrow{a,\tilde{\sigma}} (q',\sigma')}. \tag{1}$$

The closed semantics *of* B *is given by the LTS denoted* $[\![B]\!]$*, comprising only those transitions of* $[B]$*, where* $\tilde{\sigma} = \sigma$*.*

The use of the intermediate valuation $\tilde{\sigma}$ in the conclusion and the third premise of (1) allows some variables to get new values before the transition is fired. Thus the component is *open* to the exchange of data with its environment. However, the fourth premise states that only the variables exported through the ports participating in the interaction can be affected by the data transfer. The closed semantics excludes this possibility of data exchange.

Definition 7 (Interaction model). *For a finite set of component interfaces* $(V_i, P_i, \varepsilon_i)^{i \in I}$*, such that all* P_i *and all* V_i *are pairwise disjoint, let* $P = \bigcup_{i \in I} P_i$*,* $V = \bigcup_{i \in I} V_i$ *and* $\varepsilon : P \to 2^V$ *such that, for any* $p \in P_i$*,* $\varepsilon(p) = \varepsilon_i(p)$*.*

An interaction model over (V, P, ε) *is a set* $\Gamma \subseteq 2^P \times \mathbb{B}^{\leqslant}_V \times \mathbb{A}^{\leqslant}_V$*, such that, for any interaction* $(a, g, u) \in \Gamma$*, we have* $g \in \mathbb{B}^{\leqslant}_{\varepsilon(a)}$ *and* $u \in \mathbb{A}^{\leqslant}_{\varepsilon(a)}$*.*[5]

We assume that all sets of components and interfaces satisfy the disjointness assumption above. We call the *support* of a set of ports $a \subseteq P$, denoted $\mathrm{supp}(a)$, the set of the participating components. It is either the set $\{i \in I \mid a \cap P_i \neq \emptyset\}$ (for $P = \bigcup_{i=1}^n P_i$) or the set $\{B \in \mathcal{B} \mid a \cap P_B \neq \emptyset\}$ (for $P = \bigcup_{B \in \mathcal{B}} P_B$). The precise meaning of this notation will always be clear from the context.

Definition 8 (Composition). *The* composition *of a finite set of components* $\mathcal{B} = (Q_i, q_i^0, V_i, \sigma_i^0, P_i, \varepsilon_i, \to)^{i \in I}$ *with the interaction model* Γ *over* (V, P, ε) *is the component* $\Gamma(\mathcal{B}) = (Q, q^0, V, \sigma^0, P, \varepsilon, \to)$*, where* $Q = \prod_{i \in I} Q_i$*;* $q^0 = (q_i^0)^{i \in I}$*;* $\sigma^0 : V \to \mathbb{D}$ *is such that, for any* $v \in V_i$*,* $\sigma^0(v) = \sigma_i^0(v)$*; and* $\to$ *is the minimal transition relation satisfying the rule*

$$\frac{\begin{array}{c}\forall i \in \mathrm{supp}(a), q_i \xrightarrow{a \cap P_i, g_i, u_i} q_i' \qquad \forall i \notin \mathrm{supp}(a), q_i = q_i' \\ g' = g \wedge \bigwedge_{i \in \mathrm{supp}(a)} g_i \qquad u' = u; u_i^{i \in \mathrm{supp}(a)} \qquad (a, g, u) \in \Gamma \qquad a \neq \emptyset\end{array}}{(q_i)^{i \in I} \xrightarrow{a, g', u'} (q_i')^{i \in I}}.$$

Intuitively, an interaction can be fired if all the involved components are ready to fire their corresponding transitions. The other components do not change their

[5] Notice that this definition allows $(\emptyset, \mathtt{tt}, \emptyset)$ and $(\emptyset, \mathtt{ff}, \emptyset)$ to be included in Γ.

states. Both the interaction guard and those of the participating transitions must be satisfied. The update assignment of the interaction is executed first, followed by those of the components.

Specifying interaction models as sets of sets of ports is not practical due to their potentially exponential size. An algebra of connectors was introduced in [14] in order to structure interactions in BIP models. Connectors are hierarchical, tree-like structures with component ports at the leaves. They define sets of interactions, based on the attributes of the nodes, which may be either *trigger* (triangles in Fig. 1) or *synchron* (bullets in Fig. 1). If all sub-connectors of a connector are synchrons, then an interaction is allowed by the connector only if each subconnector can contribute. If at least one of the sub-connectors is a trigger, then any interaction consisting of contributions of any set of sub-connectors *involving at least one of the triggers* is allowed. The interaction model is defined as the set of all interactions allowed by at least one of the connectors.

For instance, the connector $T.\mathsf{tick}\blacktriangleright\!\!-\!\!\bullet(\mathsf{fail}\blacktriangleright\!\!-\!\!\bullet C.\mathsf{fail})$ of Fig. 1 is a two-level hierarchical connector. In the subconnector $\mathsf{fail}\blacktriangleright\!\!-\!\!\bullet C.\mathsf{fail}$, the port fail is a trigger, whereas $C.\mathsf{fail}$ is a synchron. This subconnector allows two interactions: $\{\mathsf{fail}\}$ and $\{C.\mathsf{fail}, \mathsf{fail}\}$. Similarly, at the top level, $T.\mathsf{tick}$ is a trigger, and the subconnector is a synchron. The entire connector defines the following three interactions (observe that $\top + T.t = \top$ and $T.z \cap \top = T.z$): $\big(\{T.\mathsf{tick}\}, \mathtt{tt}, \emptyset\big)$, $\big(\{\mathsf{fail}, T.\mathsf{tick}\}, \mathtt{tt}, \emptyset\big)$, $\big(\{C.\mathsf{fail}, \mathsf{fail}, T.\mathsf{tick}\}, \mathtt{tt}, T.z := T.z \cap (C.zone + T.t)\big)$

In addition to interaction models, BIP relies on *priority models* that impose a strict partial order on interactions. Intuitively, an interaction can be fired only if all the higher-priority interactions available in the current state are disabled by their respective guards. In the next sections, we will implicitly assume application of the *maximal progress* priority μ, where $(a, g, u) \prec_\mu (b, h, w)$ iff $a \subset b$ and $a \neq b$. For instance, the port $T.\mathsf{tick}$ will never fire alone if the port fail is also enabled.

Architectures Architectures are partial BIP models, with *dangling* ports that serve as placeholders for the eventual connection with *operand* components.

Definition 9 (Architecture). *An* architecture *is a tuple* $A = (\mathcal{C}, V_A, P_A, \varepsilon_A, \Gamma)$*, where*

- P_A *and* V_A *are sets of ports and variables, respectively;*
- $\mathcal{C}$ *is a finite set of components (called* coordinators*), such that* $\bigcup_{C\in\mathcal{C}} P_C \subseteq P_A$ *and* $\bigcup_{C\in\mathcal{C}} V_C \subseteq V_A$*; ports in* $P_A \setminus \bigcup_{C\in\mathcal{C}} P_C$*, which do not belong to any of the coordinators are called* dangling*;*
- $\varepsilon_A : P_A \to 2^{V_A}$ *is an export function, such that* $\varepsilon_A(p) = \varepsilon_C(p)$*, for any* $C \in \mathcal{C}$ *and* $p \in P_C$ *and* $\varepsilon_A(p) \subseteq V_A \setminus \bigcup_{C\in\mathcal{C}} V_C$ *for any dangling port* p*; and*
- $\Gamma \subseteq 2^{P_A} \times \mathbb{B}^{\leqslant}_{V_A} \times \mathbb{A}^{\leqslant}_{V_A}$ *is an interaction model over* $(V_A, P_A, \varepsilon_A)$*.*

Definition 10 (Application of an architecture). *Let* $A = (\mathcal{C}, V_A, P_A, \varepsilon_A, \Gamma)$ *be an architecture and let* $\mathcal{B}$ *be a set of components, such that* $V_A \subseteq V \stackrel{def}{=} \bigcup_{B\in\mathcal{B}\cup\mathcal{C}} V_B$*,* $P_A \subseteq P \stackrel{def}{=} \bigcup_{B\in\mathcal{B}\cup\mathcal{C}} P_B$ *and* $\varepsilon_A(p) = V_A \cap \varepsilon_B(p)$*, for any* $B \in \mathcal{B}$

and $p \in P_A \cap P_B$. *The* application of the architecture A to the set of components $\mathcal{B}$ *is the component* $A(\mathcal{B}) \stackrel{def}{=} \mu\big((\Gamma \ltimes P)(\mathcal{C} \cup \mathcal{B})\big)$, *where* $\Gamma \ltimes P \stackrel{def}{=} \big\{(a, g, u) \,\big|\, a \subseteq P, (a \cap P_A, g, u) \in \Gamma\big\}$ *is the interaction model over* $(V, P, \varepsilon_A \cup \bigcup_{B \in \mathcal{B}} \varepsilon_B)$ *and* $\mu(\dots)$ *denotes the application of maximal progress.*

An architecture A enforces coordination constraints on the components in $\mathcal{B}$. The interface $(V_A, P_A, \varepsilon_A)$ of an architecture A contains all ports of the coordinators $\mathcal{C}$ and the dangling ports, which must belong to the components in $\mathcal{B}$. In the application $A(\mathcal{B})$, the ports belonging to P_A can only participate in the interactions defined by the interaction model Γ of A. Ports which do not belong to P_A are not restricted and can participate in any interaction. The definition of $\Gamma \ltimes P$ requires that an interaction from Γ be involved in every interaction belonging to $\Gamma \ltimes P$. To allow the ports from $P \setminus P_A$ to be fired independently in $A(\mathcal{B})$, one must have $(\emptyset, \mathtt{tt}, \emptyset) \in \Gamma$.

In our running example, there are four dangling ports. Intuitively, the architecture monitors the activation of the dangling port fail, then waits for a period comprised between Min and Max and, unless resume is activated, asks for a system reset through an invocation of the dangling port ask.

Definition 11 (Composition of architectures). *Let* $A_i = (\mathcal{C}_i, V_{A_i}, P_{A_i}, \varepsilon_{A_i}, \Gamma_i)$, *for* $i = 1, 2$, *be two architectures. The* composition *of* A_1 *and* A_2 *is the architecture* $A_1 \oplus A_2 = (\mathcal{C}_1 \cup \mathcal{C}_2, V_{A_1} \cup V_{A_2}, P_{A_1} \cup P_{A_2}, \varepsilon_{A_1} \cup \varepsilon_{A_2}, \Gamma)$, *where*

$$\Gamma = \big\{(a, g^1 \wedge g^2, u^1 \wedge u^2) \,\big|\, (a \cap P_{A_i}, g^i, u^i) \in \Gamma_i, \text{ for } i = 1, 2\big\}. \tag{2}$$

$\oplus$ is associative and commutative.

It is well known that, since violations of safety properties are characterised by finite executions, they can also be represented as state predicates: intuitively, a safety property corresponds to the predicate characterising the set of states, where this property is not violated.

For a component B, we denote $S_{[\![B]\!]}$ and $s^0_{[\![B]\!]}$ the corresponding constituents of $[\![B]\!]$ (see Definition 6).

Definition 12 (Properties). *Let* B *be a component. A* (safety) property *of* B *is a predicate* Φ *on* $S_{[\![B]\!]}$, *such that* $\big((q, \sigma) \models \Phi\big) \wedge (\sigma' \leqslant \sigma)$ *implies* $(q, \sigma') \models \Phi$. *A property* Φ *is* initial *if* $s^0_{[\![B]\!]} \models \Phi$.

Although we define properties as state predicates, any appropriate logic can be used to specify them. For instance, the property *"There is always a possibility to reset the system after a single failure"* (i.e. without additional failures having to occur in the meantime) enforced by the Failure Monitor architecture comprises the safety component that can be specified using CTL as AG $\big($fail $\rightarrow$ EX E$[\neg$fail W reset$]\big)$. An architecture enforces its characteristic property on its operand components. From this point of view, the set of coordinators is not relevant, neither are their states. Thus, properties enforced by architectures only involve the unrestricted composition of the operands:

Definition 13 (Enforcing properties). *Let $A = (\mathcal{C}, P_A, V_A, \varepsilon_A, \Gamma)$ be an architecture; let $\mathcal{B}$ be a set of components and Φ an initial property of their parallel composition $\Gamma_\parallel(\mathcal{B})$, with $\Gamma_\parallel = \{(a, \mathtt{tt}, \emptyset) \mid a \subseteq \bigcup_{B\in\mathcal{B}} P_B\}$. We say that A* enforces *Φ on $\mathcal{B}$ iff, for every state $s = (s_c, s_b)$ reachable in $[\![A(\mathcal{B})]\!]$, with $s_c \in \prod_{C\in\mathcal{C}} S_{[\![C]\!]}$ and $s_b \in \prod_{B\in\mathcal{B}} S_{[\![B]\!]}$, we have $s_b \models \Phi$.*

In the following, when we say that an architecture enforces some property Φ, Φ is supposed to be initial for the coordinated components. In [12], we formally define *upwards compatibility* that ensures property preservation when composing architectures. Informally, two architectures A_1 and A_2 are upwards compatible iff, whenever their composition involves the fusion of two interactions $a_1 = a \cap P_{A_1}$ and $a_2 = a \cap P_{A_2}$ (see (2)) and one, say a_1, is inhibited in a given state by a larger interaction $b_1 \supset a_1$, there exists an interaction $b_2 \supseteq a_2$ that can be fused with b_1 to form an interaction enabled in the same state.

Theorem 1 (Preserving enforced properties). *Let $\mathcal{B}$ be a set of components; let $A_i = (\mathcal{C}_i, V_{A_i}, P_{A_i}, \varepsilon_{A_i}, \Gamma_i)$, for $i = 1, 2$, be two upwards compatible architectures enforcing on $\mathcal{B}$ the properties Φ_1 and Φ_2 respectively. The composition $A_1 \oplus A_2$ enforces on $\mathcal{B}$ the property $\Phi_1 \wedge \Phi_2$.*

Theorem 1 implies that safe BIP systems can be designed *compositionally*: it is sufficient to verify that 1) the applied architectures do enforce their characteristic properties and 2) they are pairwise upwards compatible. To a large extent, the latter can be carried out syntactically by analysing the structure of the connectors that define the interaction models. The next section is devoted to the encoding of architectures into pNets, addressing item 1 by symbolic verification.

4 Encoding of architectures into open pNets

We define the encoding of BIP architectures into pNets by associating to each architecture $A = (\mathcal{C}, V_A, P_A, \varepsilon_A, \Gamma)$ with $C = (Q_C, q_C^0, V_C, \sigma_C^0, P_C, \varepsilon_C, \rightarrow)$, for each $C \in \mathcal{C}$, *and a partition $\mathcal{D} \subseteq 2^{P_A}$ of its dangling ports* (i.e. $\biguplus_{D\in\mathcal{D}} D = P_A \setminus \bigcup_{C\in\mathcal{C}} P_C$), the corresponding pNet $enc(A, \mathcal{D})$. For the sake of clarity, we define the encoding without any priority model. Then, we provide a brief sketch of the modifications necessary to encode maximal progress (implicitly assumed). Recall that Γ is an interaction model over the interface $(V_A, P_A, \varepsilon_A)$, i.e. these interface elements are implicitly involved in the definition of Γ. We define $enc(A, \mathcal{D}) \stackrel{def}{=} \langle\!\langle (enc(C))^{C\in\mathcal{C}}, \mathcal{D}, enc(\Gamma) \rangle\!\rangle$, where $enc(C)$ and $enc(\Gamma)$ are the encodings of a coordinator C and the interaction model Γ respectively.

Below, we present both the encodings of coordinators and interaction models. The key constraint is that we encode each connector by one synchronisation vector. This is necessary to 1) preserve the structure of the system and 2) allow the encoding of maximal progress.

Although somewhat technical, the encoding of coordinators is, in fact, pretty straightforward, comprising three key ideas: 1) we introduce an additional transition (hence also an additional state) to explicitly initialise the variables; 2) we

introduce additional input variables to manipulate the values provided to the coordinator by the rest of the system for all exported variables; and 3) following the classical technique [35], we simulate the absence of action by an additional loop transition.

The encoding of connectors (interaction models) is more involved. Since a connector represents a set of potential interactions, some ports may not participate in all of them. To encode this possibility, we introduce, for each port, an additional Boolean variable denoting whether the port participates in the interaction or not and, for each connector, a predicate characterising the interaction pattern. Intuitively, the semantics of *flat* BIP connectors [14] depends on the synchron/trigger annotations of ports. If all ports in a connector are synchrons, the only allowed interaction is that comprising all the ports, i.e. they all have to "agree to interact". This case corresponds precisely to the semantics of synchronisation vectors in pNets. If a connector has at least one trigger, then the allowed interactions are those that comprise at least one trigger, i.e. they must be "initiated by a trigger". In a hierarchical connector, these principles are applied recursively. Thus, we observe a "causality" relation among ports of a connector: participation of a synchron in an interaction *implies* that of a trigger. Causal Interaction Trees and Systems of Causal Rules, proposed in [15], formalise this causality relation and provide transformations from connectors to Boolean predicates and back. Since we use SMT techniques for the analysis of the resulting pNet, the encoding presented below is optimised to reduce the number of variables by treating separately the "top-level" triggers (e.g. $T.\mathsf{tick}$ in the connector $T.\mathsf{tick}\blacktriangleright\!\!-\!\!\bullet(\mathsf{fail}\blacktriangleright\!\!-\!\!\bullet C.\mathsf{fail})$ in Fig. 1).

Encoding the coordinators. The encoding of a coordinator C is a pLTS with: 1) an initial state and an init transition that initialises all the variables to those defined by the initial valuation σ^0_C of C, 2) an action algebra that matches the actions of the coordinator ports but adds, an additional Boolean action parameter, and also, for each exported variable x, a corresponding fresh input variable $?x'$ to allow updates during interactions, 3) pLTS transitions that reflect the original transitions of C with $\mathtt{tt}$ as parameter and 4) additional loop transitions marked by $\mathtt{ff}$. Formally, $enc(C) \stackrel{def}{=} \langle\!\langle S, s_0, \xrightarrow[enc(C)]{} \rangle\!\rangle$, such that

- $s_0 \notin Q_C$ is fresh and $S = Q_C \cup \{s_0\}$,
- $vars(s) = V_C \cup \{?x' \mid x \in \varepsilon_C(p), p \in a, s \xrightarrow{a}\}$, for all $s \in Q_C$, and $vars(s_0) = \emptyset$,
- let $uinit \stackrel{def}{=} \big(x := \sigma^0_C(x)\big)^{x \in V_C}$ and, for all $s \xrightarrow{a,g,u} s'$ with $u = (x := e_x)^{x \in V}$ (with $V \subseteq V_C$), let $u' \stackrel{def}{=} \big(x := e_x\big)^{x \in V \setminus \varepsilon_C(a)} \cup \big(x := e_x\big[{}^{?x'}/_{x}\big]\big)^{x \in V \cap \varepsilon_C(a)}$, and $\varepsilon'_C(a) \stackrel{def}{=} \{?x' \mid x \in \varepsilon_C(p), p \in a\}$,

$$\xrightarrow[enc(C)]{} \stackrel{def}{=} \Big\{ \Big(s, a\big(\varepsilon'_C(a), \varepsilon_C(a), \mathtt{tt}\big), g, u', s'\Big) \;\Big|\; s \xrightarrow{a,g,u} s' \Big\}$$
$$\cup \Big\{ \Big(s, a\big(\varepsilon'_C(a), \varepsilon_C(a), \mathtt{ff}\big), \mathtt{tt}, \emptyset, s\Big) \;\Big|\; s \in Q_C, \exists s' \in Q_C : s' \xrightarrow{a} \Big\}$$
$$\cup \big\{(s_0, \mathsf{init}, \mathtt{tt}, uinit, q^0_C)\big\}.$$

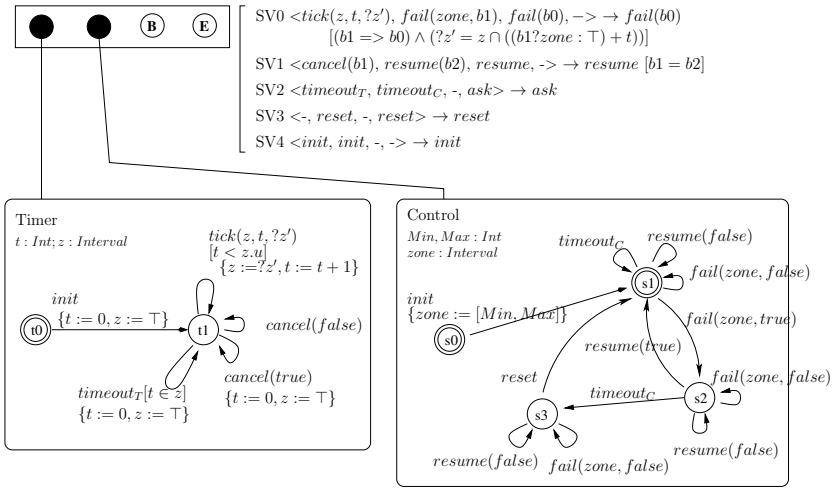


Fig. 2: The open pNet encoding the Failure Monitor architecture (Fig. 1) without the Max Progress priority model

The loop transitions marked by `ff` will be used in the encoding of connectors. Each BIP connector can define several interactions, i.e. ports involved in a connector need not necessarily always participate. On the contrary, each action in a pNet synchronisation vector must participate in the synchronisation. To address this difference, we use the classical approach where non-participation of a port in an interaction is simulated by an additional loop transition [35].

Figure 2 shows the encoding of the Failure Monitor architecture, including the encodings of the two coordinators, i.e. $enc(T)$ and $enc(C)$. Notice that the encoding in the figure is slightly optimised: some of the ports do not have an associated Boolean value, nor the additional loop transitions. We will explain this optimisation after we define the encoding of the interaction model.

Encoding the interaction model The holes in $enc(A, \mathcal{D})$ are indexed by the elements of the partition $\mathcal{D}$. For the encoding of our running example, we take $\mathcal{D} = \big\{\{\mathsf{fail}, \mathsf{resume}\}, \{\mathsf{ask}, \mathsf{reset}\}\big\}$. This corresponds to the intuition that the dangling ports fail and resume will be provided by a monitored component, whereas ask and reset correspond to the actions provided by the "environment" (other components of the system) that are invoked in case of a persistent failure. As for the encoding of the coordinators, in the synchronisation vectors of $enc(\Gamma)$, we will associate Boolean values to the actions corresponding to these ports.

The encoding of the interaction model is based on its representation as a set of connectors. Indeed, as illustrated by the Failure Monitor architecture in Fig. 1, each connector can define several allowed interactions, depending on its hierarchical structure and the use of synchrons and triggers.

We encode all interactions of a connector in one synchronisation vector. This will allow us to also encode the maximal progress priority model. We use the

additional Boolean values associated to each port by the encoding of coordinator components. For example, observe that the three ports in the connector $T.\mathsf{tick}\blacktriangleright\!\!-\!\!\bullet(\mathsf{fail}\blacktriangleright\!\!-\!\!\bullet C.\mathsf{fail})$ form a "causality chain": $C.\mathsf{fail}$ can only participate in an interaction if the dangling port fail participates, which in turn can only happen if $T.\mathsf{tick}$ does so. These dependencies can be rewritten as Boolean implications $C.\mathsf{fail} \Rightarrow \mathsf{fail}$ and $\mathsf{fail} \Rightarrow T.\mathsf{tick}$. The conjunction of these two implications can be used as a guard for the synchronisation vector encoding this connector.

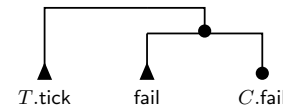
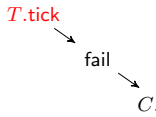
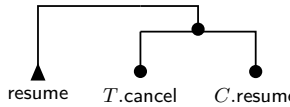
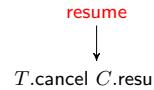
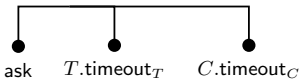
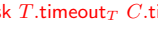
Within the scope of this connector, the port $T.\mathsf{tick}$ participates in all interactions. Furthermore, it is not involved in any other connector. Hence, the loop transition in $enc(T)$ labelled by $\mathsf{tick}(\mathtt{ff})$ can never be taken and, therefore, can be removed from the encoding. Since only the transition labelled by $\mathsf{tick}(\mathtt{tt})$ is ever taken, the implication $\mathsf{fail} \Rightarrow T.\mathsf{tick}$ is a tautology and can also be discarded.

We obtain the synchronisation vector SV0 shown in Fig. 2, where $b0$ and $b1$ are the Boolean values associated to the actions encoding the ports fail and $C.\mathsf{fail}$. The guard $b1 \Rightarrow b0$ encodes the causal relation between these ports. Notice that all three ports are present in the synchronisation vector. Figure 2 shows the four synchronisation vectors SV0–SV3 corresponding to the connectors in Fig. 1 and an additional vector SV4, synchronising the init transitions of the two pLTSs.

In the general case, the encoding relies on the causal semantics of the algebra of BIP connectors [15]. Disregarding the variables and data transfer, the Algebra of Connectors $\mathcal{AC}(P)$ [13] provides a syntactic notation for the BIP connectors. The causal semantics of the connectors, given in terms of the Algebra of Causal Interaction Trees $\mathcal{T}(P)$, elicits the causal dependencies through an encoding mapping $\tau : \mathcal{AC}(P) \to \mathcal{T}(P)$. Another mapping $R : \mathcal{T}(P) \to \mathcal{CR}(P)$ encodes causal interaction trees into systems of causal rules, which are Boolean implications similar to the ones in the example above. The $\mathcal{AC}(P)$, $\mathcal{T}(P)$ and $\mathcal{CR}(P)$ representations of the connectors in Fig. 1 are shown in Tab. 1 (elements shown in red can be removed for simplification as described in the example above).

Now we lift this encoding to the data-sensitive case. Below, we assume that, as in Fig. 1, the interaction model is defined by a set of connectors, annotated with Boolean guards and with update assignments. In particular, we assume that the guards and update assignments are well-defined for any interaction allowed by the connector. For example, the choice $C.\mathsf{fail}\ ?\ C.zone : \top$ in the update assignment $T.z := T.z \cap \big((C.\mathsf{fail}\ ?\ C.zone : \top) + T.t\big)$ associated to the connector $T.\mathsf{tick}\blacktriangleright\!\!-\!\!\bullet(\mathsf{fail}\blacktriangleright\!\!-\!\!\bullet C.\mathsf{fail})$ in Fig. 1 ensures that the assignment is well-defined independently of whether $C.\mathsf{fail}$ participates or not. Let us denote by $\gamma \subset \mathcal{AC}(P_A) \times \mathbb{B}^{\leqslant}_{V_A} \times \mathbb{A}_{V_A}$ the set of connectors in the architecture A and by P_x the set of ports involved in the connector $x \in \gamma$. Then, the interaction model defined by γ is $\Gamma = \{(a, g, u) \,|\, (x, g, u) \in \gamma, a \in \|x\|\}$ and the set of synchronisation vectors

Table 1: Algebraic representations of the connectors in Fig. 1

Connector	Causal Interaction Tree	System of Causal Rules
		C.fail ⇒ fail ∧ T.tick fail ⇒ T.tick T.tick ⇒ tt tt ⇒ T.tick
		C.resume ⇒ resume ∧ T.cancel T.cancel ⇒ resume ∧ C.resume resume ⇒ tt tt ⇒ resume
		C.timeout$_C$ ⇒ ask ∧ T.timeout$_T$ T.timeout$_T$ ⇒ ask ∧ C.timeout$_C$ ask ⇒ T.timeout$_T$ ∧ C.timeout$_C$ tt ⇒ ask ∧ T.timeout$_T$ ∧ C.timeout$_C$
		C.reset ⇒ reset reset ⇒ C.reset tt ⇒ reset ∧ C.reset

encoding Γ is $enc(\Gamma) \stackrel{def}{=} \{enc(x,g,u) \,|\, (x,g,u) \in \gamma\}$, with

$$enc(x,g,u) \stackrel{def}{=} \left(\left\{p\big(\varepsilon_A(p), \varepsilon'_A(p), b_p\big) \,\middle|\, p \in P_x \cap P\right\}\right)^{P \in P_C^{C \in \mathcal{C}} \cup \mathcal{D}} \to \alpha\big(b_p^{p \in P_x}\big)$$
$$\left[\bigwedge R(\tau(x))\big[{}^{b_p}/_{p}\big] \wedge \bigwedge_{(x:=e_x)\in u} (?x' = e_x) \wedge \bigwedge_{x \in \varepsilon_A(x), x \notin u} (?x' = x)\right], \quad (3)$$

with $b_p^{p \in P_x}$ fresh Boolean variables, α a fresh name, $\tau : \mathcal{AC}(P_A) \to \mathcal{T}(P_A)$ and $R : \mathcal{T}(P_A) \to \mathcal{CR}(P_A)$ the two mappings [15] discussed above and illustrated in Tab. 1, $\big[{}^{b_p}/_{p}\big]$ is the substitution that replaces in the expression that precedes it all occurrences of all p by corresponding variables b_p.

For the sake of clarity, we simplify the case study encoding in Fig. 2 by reusing the port names of the original architecture instead of fresh names α. This is made possible by the fact that each synchronisation vector involves at most one action of interest (see the properties in [12]).

In the following theorem, we claim an *isomorphism* between the open automaton semantics of a pNet encoding a BIP architecture and the LTS semantics of this architecture applied to a set of simple components. We omit the formal definition of this isomorphism relation. However, noting that open automata are, essentially, symbolic representations of automata with data, we can summarise it as follows: a transition belongs to the LTS iff a corresponding open transition belongs to the open automaton and the source and target data values of the LTS transition satisfy the predicate and implement the assignments of the open transition.

Theorem 2. *The open automaton* $[enc(A,\mathcal{D})]$ *corresponding to* $enc(A,\mathcal{D})$ *is isomorphic to the LTS* $[\![\Gamma\big(\mathcal{C}_A, (B_D)^{D \in \mathcal{D}}\big)]\!]$ *(see Def. 6), with, for each* $D \in \mathcal{D}$,

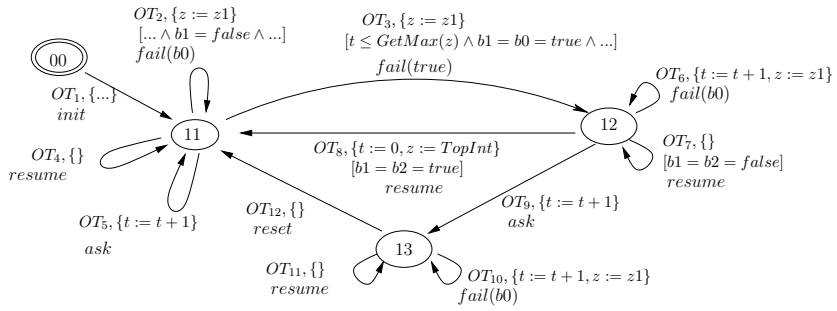


Fig. 3: The open Automaton of the Failure Monitor Architecture

the component $B_D \stackrel{def}{=} (\{q\}, q, V_D, \sigma_D^0, D, \varepsilon_D, \rightarrow)$*, with a fresh state* q *and*

$$V_D = \bigcup_{p \in D} \varepsilon_A(p)\,, \qquad \sigma_D^0(v) = \bot,\ \textit{for all } v \in V_D\,,$$
$$\rightarrow = \left\{(q, p, \mathtt{tt}, \emptyset, q) \,\middle|\, p \in D\right\}, \qquad \varepsilon_D(p) = \varepsilon_A(p),\ \textit{for all } p \in D\,.$$

Encoding of the maximal progress We only present the key idea, which consists in introducing an additional Boolean variable, for each port, for which we have introduced one in $enc(A, \mathcal{D})$. Intuitively, the Boolean variables introduced for the encoding of interaction models determine whether it is the original transition labeled by the port that is executed ($\mathtt{tt}$), or rather the corresponding self-loop, introduced by the encoding ($\mathtt{ff}$). The new variables determine whether *there is an original transition labelled by* p *that could be executed from the same state*, i.e., with b_p the variable introduced above for the encoding without maximal progress, $q \xrightarrow{p(\mathtt{tt}, b_p)} q'$ iff $\exists q'' : q \xrightarrow{p(b_p)} q''$ with $b_p = \mathtt{tt}$ in $enc(A, \mathcal{D})$. In the SV guard, we have to check whether *all ports leading to* p *in the causal interaction tree* $\tau(x)$ *can be fired* (see the second column of Tab. 1 for examples). If so, then p must be fired, i.e. $p(\mathtt{ff})$ must be blocked.

Practical experiments. The above encoding provides a mechanism for the symbolic verification of architectures using our existing tool [36] to compute the open automaton semantics of an open pNet. This tool computes open transitions from pLTS behaviours and synchronisation vectors of the pNet, then uses an SMT engine to check satisfiability of their predicates, minimising the size of the resulting automaton.

In Fig. 3 we show the full open automaton obtained from the pNet in Fig. 2. Due to space limitations, we do not show the details of the open transitions, but only the assignments of state variables, some useful parts of the predicates, and the resulting action; full details can be found in [12]. This automaton has 12 transitions, including those encoding various possible firing of some interactions, e.g. OT_2 and OT_3 for fail. Notice, however, that the global actions of these

open transitions have an additional Boolean parameter. In this context, model checking should be understood after application of the encoding, namely here the original fail event must be an effective "fail" of the hole "B", that is a $fail(true)$ action in the open automaton.

Model-checking of open-automaton is out of the scope of this paper, though the resulting automaton here is small enough to observe the kind of properties we can prove. It is clear that our encoding allows to test specific values of state-variables in formulas, like e.g.:

$$\mathtt{A}\,[(z = \top)\ \mathtt{W}\ \mathsf{fail}] \quad \wedge \quad \mathtt{AG}\,\big(\mathsf{reset} \rightarrow \mathtt{A}\,[(z = \top)\ \mathtt{W}\ \mathsf{fail}]\big). \tag{4}$$

that says that *as long as no failure has occured, the z variable of the T component has the value $\top$.*

But we can also (as long as we get a proper axiomatisation of our data operators in the SMT engine) handle more involved data properties, like here the fact that *a reset can only be requested within the specified delay after a failure:*

$$\begin{aligned}\forall T_0, T_1 \in \mathbb{Z},\ \mathtt{AG}\,\Big((\mathsf{fail} \wedge T.t = T_0) \rightarrow \\ \mathtt{A}\,\big[\big((\mathsf{ask} \wedge T.t = T_1) \rightarrow (T_1 - T_0 \in C.zone)\big)\ \mathtt{W}\ (\mathsf{reset} \vee \mathsf{resume})\big]\Big),\end{aligned} \tag{5}$$

Last, a detailed study shows that the safety property stated on page 10 does not hold, because of the fail loop on state 11. This is because we did not use the maximal progress assumption here. If we do, we get the corrected behaviour where OT_7 disappears, and OT_2 and OT_{10} are restricted to $b0 = \mathtt{ff}$. This one verifies all the properties listed above.

5 Related work

The design methodology based on BIP architectures is inspired by the notion of design patterns introduced in [24]. It is radically different insofar as BIP architectures posses formal semantics; their composition is well defined and preserves their characteristic properties. This is a relatively novel trend with few comparable works, whereof the most relevant is a theory formalising common architectural styles, such as *publisher-subscriber* or *blackboard*, proposed in [32,33].

Although direct verification of BIP models is possible [4,9,10,11,37], none of these previous works address compositional verification of parameterised BIP systems with data and maximal progress priorities achieved in the present paper.

From a broader perspective, basic research on behaviour models and verification algorithms for data-sensitive systems started in the nineties, with the seminal work of Hennessy, Lin, and their colleagues on value-passing systems with assignments [26,30,31]. Later, many different works addressing various classes of infinite-state systems and/or parameterised topologies have been published, using combinations of approaches, often including predicate abstraction and SMT

satisfiability (e.g. [1,16,20,21,25]). With respect to these, we use symbolic representations not only to get a finite representation of infinite spaces, but also to express the (data-sensitive) synchronisations with the environment, making our models suitable for compositional verification. Among these works, several have shown the capacity of the SMT engines (either Z3 or Yikes) as servers for solving verification conditions of the algorithms, for large case-studies (e.g. [18,22]).

As compositional proof of safety is difficult, some approaches rely on theorem proving to ensure the safety of component operations. Coqots and Pycots [17] even manage to prove the safety of reconfiguration procedures which are known to be highly difficult to verify, and massively parameterised. The approach relies on a high expertise and significant efforts from the user. Here, we rely on automatic verification thanks to the SMT solver but we cannot prove the safety of the reconfiguration procedure.

In [23], the authors propose a compositional proof system for distributed objects that is suitable for implementation within the KeY framework [8] and uses a Hoare logic approach. Compared to this approach, we do not deal with complex history-based specifications and use interaction specification and SMT-reasoning instead of Hoare logic.

6 Conclusion

BIP architectures are composition tools that enforce safety properties; the composition of architectures entails the composition of the associated properties. We have extended architectures with data-sensitive interactions, and proved that this extension still guarantees the preservation of safety properties by architecture composition, under reasonable assumptions. This extends the original compositional methodology offered by BIP architectures. Then we use pNets as a semantic formalism to encode architectures with data. pNet is a low level coordination model for open systems, in which composition preserves bisimulation equivalences. pNet is equipped with tools computing its behavioural semantics in terms of symbolic automata, allowing model-checking and equivalence checking with algorithms relying on SMT engines. As a result, we obtain automatic and compositional guarantees of safety properties with data where compositionality is given by the BIP architectures, and pNet tools provide automatic verification of the properties of each architecture.

The translational approach allows us to benefit from the methods and tools developed separately in BIP and pNets communities, avoiding the additional effort of designing the corresponding tools in both contexts from scratch.

The presented work opens a number of avenues for future work, among which the most immediate ones consist in 1) developing tools that would implement the discussed encoding and verification techniques; 2) studying the preservation of liveness properties by architecture composition under assumptions similar to those discussed in [5] and 3) generalisation to priority models other than maximal progress.

References

1. F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi. Universal guards, relativization of quantifiers, and failure models in model checking modulo theories. *JSAT*, 8(1/2):29–61, 2012.
2. R. Ameur-Boulifa, L. Henrio, O. Kulankhina, E. Madelaine, and A. Savu. Behavioural semantics for asynchronous components. *Journal of Logical and Algebraic Methods in Programming*, 89:1 – 40, 2017.
3. A. Arnold. Synchronised behaviours of processes and rational relations. *Acta Informatica*, 17:21–29, 1982.
4. L. Astefanoaei, S. B. Rayana, S. Bensalem, M. Bozga, and J. Combaz. Compositional verification of parameterised timed systems. In K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015*, volume 9058 of *LNCS*, pages 66–81. Springer, 2015.
5. P. Attie, E. Baranov, S. Bliudze, M. Jaber, and J. Sifakis. A general framework for architecture composability. *Formal Aspects of Computing*, 18(2):207–231, 2016.
6. E. Baranov. *A Semantic Framework for Architecture Modelling*. PhD thesis, EPFL, 2017.
7. A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, 2011.
8. B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Springer-Verlag, Berlin, Heidelberg, 2007.
9. S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis. D-Finder: A tool for compositional deadlock detection and verification. In *CAV*, pages 614–619, 2009.
10. S. Bensalem, A. Griesmayer, A. Legay, T.-H. Nguyen, J. Sifakis, and R. Yan. D-Finder 2: towards efficient correctness of incremental design. In 3^{rd} *int. conf. on NASA Formal methods*, NFM'11, pages 453–458, Berlin, 2011. Springer.
11. S. Bliudze, A. Cimatti, M. Jaber, S. Mover, M. Roveri, W. Saab, and Q. Wang. Formal verification of infinite-state BIP models. In B. Finkbeiner, G. Pu, and L. Zhang, editors, *13th Int. Symposium on Automated Technology for Verification and Analysis (ATVA 2015)*, volume 9364 of *LNCS*, pages 326–343. Springer, 2015.
12. S. Bliudze, L. Henrio, and E. Madelaine. Verification of concurrent design patterns with data. Technical report, Inria, 2019. To appear.
13. S. Bliudze and J. Sifakis. The Algebra of Connectors—Structuring interaction in BIP. In *Proceedings of the 7th ACM & IEEE Int. Conf. on Embedded Software, EMSOFT 2007*, pages 11–20, Salzburg, Austria, Oct. 2007. ACM SigBED.
14. S. Bliudze and J. Sifakis. The algebra of connectors—Structuring interaction in BIP. *IEEE Transactions on Computers*, 57(10):1315–1330, 2008.
15. S. Bliudze and J. Sifakis. Causal semantics for the algebra of connectors. *Formal Methods in System Design*, 36(2):167–194, 2010.
16. R. Bruni, D. de Frutos-Escrig, N. Martí-Oliet, and U. Montanari. Bisimilarity congruences for open terms and term graphs via tile logic. In C. Palamidessi, editor, *CONCUR 2000*, pages 259–274, Berlin, 2000. Springer.
17. J. Buisson, E. Calvacante, F. Dagnat, E. Leroux, and S. Martinez. Coqcots & Pycots: non-stopping components for safe dynamic reconfiguration. In *CBSE 2014: proceedings of the 17th international ACM Sigsoft symposium on Component-based software engineering*, page 1, Lille, France, June 2014.
18. D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, and A. Rivkin. Verification of data-aware processes via array-based systems (extended version). *CoRR*, abs/1806.11459, 2018.

19. A. Cansado and E. Madelaine. Specification and verification for grid component-based applications: from models to tools. In F. S. de Boer, M. M. Bonsangue, and E. Madelaine, editors, *FMCO 2008*, number 5751 in LNCS. Springer-Verlag, 2009.
20. R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv symbolic model checker. In A. Biere and R. Bloem, editors, *CAV*, pages 334–342, Cham, 2014. Springer.
21. A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. The kind 2 model checker. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification*, pages 510–517, Cham, 2016. Springer International Publishing.
22. A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. IC3 modulo theories via implicit predicate abstraction. *CoRR*, abs/1310.6847, 2013.
23. C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *The Journal of Logic and Algebraic Programming*, 81(3):227 – 256, 2012. The 22nd Nordic Workshop on Programming Theory (NWPT 2010).
24. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
25. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT model checking of array-based systems. In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, 2008*, pages 67–82, 2008.
26. M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Comput. Sci.*, 138(2):353–389, 20 Feb. 1995.
27. L. Henrio, O. Kulankhina, S. Li, and E. Madelaine. Integrated environment for verifying and running distributed components. In P. Stevens and A. Wąsowski, editors, *Fundamental Approaches to Software Engineering: 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings*, pages 66–83, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
28. L. Henrio, E. Madelaine, and M. Zhang. pNets: an Expressive Model for Parameterised Networks of Processes. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'15)*. IEEE, 2015.
29. L. Henrio, E. Madelaine, and M. Zhang. A theory for the composition of concurrent processes. In E. Albert and I. Lanese, editors, *11th Int. Fed. Conf. on Distributed Computing Techniques (FORTE)*, LNCS. IFIP, Springer, 2016. Heraklion, Greece.
30. H. Lin. Symbolic transition graph with assignment. In U. Montanari and V. Sassone, editors, *CONCUR '96*, Pisa, Italy, 26–29 Aug. 1996. LNCS 1119.
31. H. Lin. Model checking value-passing processes. In *8th Asia-Pacific Software Engineering Conference (APSEC'2001)*, Macau, december 2001.
32. D. Marmsoler. Towards a theory of architectural styles. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 823–825, New York, NY, USA, 2014. ACM.
33. D. Marmsoler. Hierarchical specification and verification of architectural design patterns. In A. Russo and A. Schürr, editors, *Fundamental Approaches to Software Engineering, FASE 2018*, volume 10802 of *LNCS*, pages 149–168. Springer, 2018.
34. A. Mavridou, E. Stachtiari, S. Bliudze, A. Ivanov, P. Katsaros, and J. Sifakis. Architecture-based design: A satellite on-board software case study. In *13th Int. Conf. on Formal Aspects of Component Software (FACS 2016)*, 2016.
35. R. Milner. Calculi for synchrony and asynchrony. *TCS*, 25(3):267–310, 1983.

36. X. Qin, S. Bliudze, E. Madelaine, and M. Zhang. Using SMT engine to generate symbolic automata. In *18th International Workshop on Automated Verification of Critical Systems (AVOCS 2018)*. Electronic Communications of the EASST, 2018.
37. Q. Wang and S. Bliudze. Verification of component-based systems via predicate abstraction and simultaneous set reduction. In P. Ganty and M. Loreti, editors, *Trustworthy Global Computing*, volume 9533 of *LNCS, pp 147–162*. Springer, 2015.