A Theory Agenda for Component-Based Design

Joseph Sifakis[1], Saddek Bensalem[2], Simon Bliudze[1], and Marius Bozga[2]

[1] EPFL, Rigorous System Design Laboratory, Station 14, 1015 Lausanne, Switzerland
[2] Université Grenoble Alpes, VERIMAG, 38000 Grenoble, France
CNRS, VERIMAG, 38000 Grenoble, France

Dedicated to Martin Wirsing.

Abstract. The aim of the paper is to present a theory agenda for component-based design based on results that motivated the development of the BIP component framework, to identify open problems and discuss further research directions. The focus is on proposing a semantically sound theoretical and general framework for modelling component-based systems and their properties both behavioural and architectural as well for achieving correctness by using scalable specific techniques.

We discuss the problem of composing components by proposing the concept of glue as a set of stateless composition operators defined by a certain type of operational semantics rules. We provide an overview of results about glue expressiveness and minimality. We show how interactions and associated transfer of data can be described by using connectors and in particular, how dynamic connectors can be defined as an extension of static connectors. We present two approaches for achieving correctness for component-based systems. One is by compositional inference of global properties of a composite component from properties of its constituents and interaction constraints implied by composition operators. The other is by using and composing architectures that enforce specific coordination properties. Finally, we discuss recent results on architecture specification by studying two types of logics: 1) interaction logics for the specification of sets of allowed interactions; 2) configuration logics for the characterisation of architecture styles.

1 Introduction

Component-based design is the process leading from given requirements and a set of predefined components to a system meeting the requirements.

Building systems from components is essential in any engineering discipline. Components are abstract building blocks encapsulating behaviour. They can be composed in order to build composite components. Their composition should be rigorously defined so that it is possible to infer the behaviour of composite components from the behaviour of their constituents as well as global properties from the properties of individual components.

The problem of building systems from components can be defined as follows. Given a set of components $\{C_1, \ldots, C_n\}$ and a property of their product state

R. De Nicola and R. Hennicker (Eds.): Wirsing Festschrift, LNCS 8950, pp. 409–439, 2015.

space Φ find a coordinator Co such that the coordinated behaviour $Co(C_1, \ldots, C_n)$ meets the property Φ.

This problem can be studied as a synthesis problem [31]. The coordinator can be considered as a component that adequately restricts the behaviour of the components so that the resulting behaviour meets Φ. Synthesis techniques suffer from well-known complexity limitations. The coordinator is computed by (semi)-algorithms on the product space of the coordinated components.

System design pursues similar and even broader objectives than synthesis: incremental construction of systems meeting given requirements from a set of components. In contrast to synthesis, design lacks rigorous theoretical foundations. Existing frameworks are mostly informal. Designers use "ready-made" solutions to coordination problems, e.g. architectures, protocols, that have been proven correct practically or theoretically. In contrast to synthesis, design requires a variety of composition operators. It is based on the concept of architecture as a means to enforce specific characteristic properties by application of generic coordination principles. A key idea is to ensure correctness by construction by avoiding computationally expensive techniques implying state explosion.

Endowing component-based design with scientific foundations is a major scientific challenge. This requires:

1. *A general concept of component.* Currently there is no agreement on a single component model. System designers deal with heterogeneous components with different characteristics. One source of heterogeneity is the distinction between synchronous and asynchronous components. Hardware components as well as components in some data flow applications are synchronous. Another source of heterogeneity reflects the difference in programming styles. Thread-based programming allows for components to be accessed by an arbitrary number of threads sharing common data. It does not allow a strict separation between behaviour and coordination mechanisms as the programmer explicitly handles synchronisation primitives to ensure coherency of shared data, e.g. to avoid races. This style is hardly amenable to formalisation and analysis. On the contrary, actor-based programming assumes that each component has its own data transformed by a single local thread. Coordination is external to the atomic components of the application and can be ensured using general mechanisms such as protocols.
2. *Theory for composing components.* We need theory for describing and analysing the coordination between components in terms of tangible, well-founded and organised concepts. The theory should propose a set of composition operators meeting the following requirements:
   - Orthogonality, meaning that composition operators are stateless to respect a clear separation between behaviour and coordination. Many component-based frameworks do not meet this requirement. Some allow arbitrary behaviour in coordination mechanisms. This which precludes rigorous mathematical treatment focusing on coordination. Others allow a limited number of types of behaviour such as buffers or queues to the detriment of mathematical elegance.

- Minimality, meaning that none of the coordination primitives can be expressed as the combination of others without using behaviour.
- Expressiveness, meaning that the considered set of composition operators can be used to express any coordination problem. This requirement is further explained and formalised in the paper.

Notice that most of the existing component composition frameworks fail to satisfy these requirements. Some are formal such as process algebras, e.g. CCS, CSP, π-calculus, and use single composition operators that are not expressive enough. Others are ad hoc such as most frameworks used in software engineering, e.g. architecture description languages [28] which are not rooted in rigorous semantics and are hardly amenable to formalisation.

3. *Theory for ensuring correctness of components.* Being able to check or assert correctness of the built components using scalable techniques is an essential requirement. The idea is to avoid a posteriori verification and establish correctness incrementally by applying easy-to-check rules that follow the system construction.
   A key concept in this approach is that of architectures as well-established coordination schemes enforcing given properties. The problem is then to decompose any component coordination property as the conjunction of predefined characteristic properties enforced by predefined architectures.

The aim of the paper is to propose a theory agenda for rigorous component-based design. The agenda is built on existing results developed for the BIP framework [6]. It identifies work directions addressing open problems and covering a good deal of the needs. The exposition of the results is mainly informal. We provide references to technical papers for the interested reader. One of the objectives is to show mathematical relations between three hierarchically structured domains encompassing the basic concepts:

- The domain of *components* offering the possibility of interaction through their ports p and associated variables X_p through which they make available the data transferred when interactions occur.
- The domain of *connectors*, used to model coordination between components. Each connector is characterised by an interaction between ports and associated computation on the exported data. Interactions are arbitrary sets of ports. Their execution implies the atomic synchronisation of the involved components. Clearly if P is the set of the ports then the set of interactions I is a subset of 2^P.
- The domain of configurations which are sets of connectors characterising architectures. Clearly if I is the set of interactions of an architecture then the set of configurations Γ is a subset of 2^I.

The paper is structured as follows. In Section 2, we discuss the problem of composing components by proposing the concept of glue. Glue is a set of stateless composition operators defined by a certain type of operational semantics rules. We provide an overview of results about expressiveness and minimality that led

to the definition of the BIP component framework. In Section 3, we show how interactions and associated data transfer can be described by using connectors. We show in particular, how dynamic connectors can be defined as an extension of static connectors. Two approaches for achieving correctness for component-based systems are presented in Section 4. One is by compositional inference of global properties of a composite component from properties of its constituents and synchronisation constraints implied by composition operators. The other is by using and composing architectures that enforce specific coordination properties. Section 5 discusses recent results on architecture specification by studying two types of logics: 1) interaction logics for the specification of sets of allowed interactions; 2) configuration logics for the characterisation of architectural styles. The last section concludes and discusses further research directions.

2 Composing Components

2.1 The Concept of Component

A component is a tuple $C = (\Sigma, P, X, \rightarrow)$, where

- Σ is a set of *control locations*;
- P is a set of *ports*;
- X is a set of variables partitioned in two disjoint sets X_L and X_P of, respectively, *local* and *port variables*; the variables in X_P are indexed by ports, that is $X_P = \{X_p\}_{p \in P}$;
- $\rightarrow \subseteq \Sigma \times P \times G(X) \times F(X) \times \Sigma$ is a *transition relation*; transitions between control locations are labeled by triplets (p, g, f) where p is a port, g and f are, respectively, a guard Boolean expression and an update function on the variables in X.

A shorthand notation $\sigma \xrightarrow{p,g,f} \sigma'$ is commonly used to denote $(\sigma, p, g, f, \sigma') \in \rightarrow$.

Intuitively, a component can be considered as an open transition system, that is a system that performs coordination-driven computation. Coordination is defined by the environment of the component and involves two aspects: *interaction* (synchronisation) and *data transfer*. Denoting by $\mathbf{X}$ the set of all valuations of the variables in X, a state of the transition system is a pair $s = (\sigma, v)$ where $\sigma \in \Sigma$ is a control location and $v \in \mathbf{X}$ is a valuation of the component variables. Thus the state space of the transition system is $S = \Sigma \times \mathbf{X}$.

If $\sigma \xrightarrow{p,g,f} \sigma'$ then the transition system has a transition from state $s = (\sigma, v)$ to state $s' = (\sigma', v')$ if $g(v) = \texttt{true}$ and the external environment offers an interaction involving p. The execution of a transition consists in exporting the value $v(X_p)$ of the variable X_p [1] associated with port p and receiving back a new value u_p. The resulting valuation is $v' = f\big(v[u_p/X_p]\big)$ where $v[u_p/X_p]$ is the valuation obtained by replacing, in v, the value of X_p by u_p.

[1] For the sake of simplicity of notations, we consider that ports p have associated exactly one variable X_p. This restriction is, however, irrelevant and we'll consider later examples where any number of variables are associated to ports.

Sometimes, for the sake of simplicity and when data treatment is irrelevant, we will use components without data, i.e. $C = (\Sigma, P, \rightarrow)$ with $\rightarrow \subseteq \Sigma \times P \times \Sigma$. Notice that, since there are no data variables, $X = \mathbf{X} = \emptyset$ and $S = \Sigma$, i.e. the notions of state and control location coincide. Therefore, in the rest of this section, we will use 's' to denote both.

The proposed concept of component does not distinguish between input and output ports. We consider that such a distinction is not specific to ports. It can be inferred from the data-flow relation between ports specified in the coordination mechanisms. Similarly, we do not distinguish between synchronous and asynchronous components. This distinction is also inferred from the context of use.

2.2 Glue Operators

The problem of component-based design can be understood as follows. Given a component framework and a property Φ, build a composite component C which satisfies Φ. A component framework comprises a set of components $\mathcal{C}$, an equivalence relation $\cong$ and a set $\mathcal{G}$ of glue operators on these components. The glue $\mathcal{G}$ includes general composition operators, i.e. behaviour transformers, such as parallel composition.

A general formalisation of the notions of component framework and glue is provided in [13]. Below, for the sake of simplicity, we assume that components are characterised by their behaviours specified directly as Labeled Transition Systems (LTS). In this context, a component framework can be considered as a term algebra equipped with an equivalence relation $\cong$ compatible with strong bisimulation on transition systems. A composite component is any (well-formed) expression built from atomic components.

The meaning of a glue operator $gl : \mathcal{C}^n \rightarrow \mathcal{C}$ can be specified by using a set of *Structural Operational Semantics* (SOS) rules [38], defining the transition relation of the composite component $gl(C_1, \ldots, C_n)$ as a partial function of transition relations of the composed components $C_1, \ldots, C_n$. A formal and general definition of glue operators on LTS components is provided in [15]. Equation (1) shows a typical—although not general—form taken by SOS rules defining glue operators.

$$\frac{\{s_i \xrightarrow{p_i} s'_i\}_{i \in I} \quad \{s_j \overset{p_j}{\not\longrightarrow}\}_{j \in J} \quad \{s_i = s'_i\}_{i \notin I}}{s_1 \ldots s_n \xrightarrow{a} s'_1 \ldots s'_n}. \tag{1}$$

Note 1. In the general case, as opposed to (1), several negative premises can apply to a single component. In any case, at most one positive premise can apply to a component.

The rule (1) has two parts: *premises* (above the line) and *conclusion* (below the line). Sets $I, J \subseteq [1, n]$ (with $I \neq \emptyset$) index two subsets of components, which need *not* be disjoint: components $\{C_i\}_{i \in I}$ contribute *positive* premises $s_i \xrightarrow{p_i} s'_i$, whereas components $\{C_j\}_{j \in J}$ contribute *negative* premises $s_j \overset{p_j}{\not\longrightarrow}$.

The rule (1) is interpreted as follows. The state space Σ of the composite component is the Cartesian product of the state spaces of composed components: $\Sigma = \prod_{i=1}^{n} \Sigma_i$. If 1) for each $i \in I$, component C_i can execute a transition from the state s_i to s_i' (with $s_i, s_i' \in \Sigma_i$) labeled by the port $p_i \in P_i$ and 2) for each $j \in J$, component C_j *cannot* execute *any* transition from the state $s_j \in \Sigma_j$ labeled by the port $p_j \in P_j$, then the composite component $gl(C_1, \ldots, C_n)$ can execute a transition from the state $s = s_1 \ldots s_n$ to $s' = s_1' \ldots s_n'$ (with $s, s' \in \Sigma$) labeled by an *interaction* a, where $s_i' = s_i$, for all components that do not participate, i.e. C_i with $i \notin I$.

Notice that the negative premises play the role of priorities. A transition of the composite component can be executed only if a set of transitions of the constituent components are disabled.

An interaction a, in the conclusion of (1), corresponds to the atomic synchronous execution of transitions in the composed components. Depending on the component framework, the interaction label a is obtained by combining the ports $\{p_i\}_{i \in I}$ in different manners.

Example 1. In CCS [34], ports are actions belonging to a given set $L = A \cup \overline{A} \cup \{\tau\}$, where actions in $\overline{A} = \{\overline{a} \mid a \in A\}$ are complementary to those in A and $\tau \notin A \cup \overline{A}$ is a special "silent" action. The binary parallel composition operator is defined by the following three rules:

$$\frac{s_1 \xrightarrow{p} s_1'}{s_1 s_2 \xrightarrow{p} s_1' s_2}, \qquad \frac{s_2 \xrightarrow{p} s_2'}{s_1 s_2 \xrightarrow{p} s_1 s_2'}, \qquad \text{for all } p \in L\,, \tag{2}$$

$$\frac{s_1 \xrightarrow{p} s_1' \quad s_2 \xrightarrow{\overline{p}} s_2'}{s_1 s_2 \xrightarrow{\tau} s_1' s_2'}, \qquad \text{for all } p \in A \cup \overline{A} \text{ (with } \overline{\overline{p}} \stackrel{def}{=} p). \tag{3}$$

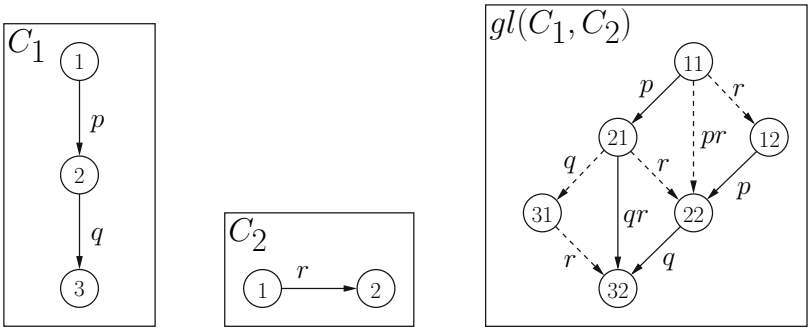
In the conclusion of the rule (3), the resulting interaction is the silent action τ, replacing the combination of two complementary actions p and $\overline{p}$. □

In [25], the authors propose a notion of *label structures*, providing a generic mechanism for defining interaction labels of the composite components. Below, for the sake of simplicity, we consider a in the conclusion of (1) to be the set of ports $\{p_i\}_{i \in I}$ in the positive premises of the rule.

As shown above, positive premises in a rule of form (1) define interactions synchronising transitions of the constituent components. In a given global state of the system, several such interactions could be possible introducing non-determinism in the composed behaviour. Negative premises define *priority rules*, which allow reducing this non-determinism.

Example 2. Consider the two components C_1 and C_2 shown in Figures 1a and 1b. Let gl be a glue operator defined by the following three rules:

$$\frac{s_1 \xrightarrow{p} s_1'}{s_1 s_2 \xrightarrow{p} s_1' s_2}, \qquad \frac{s_1 \xrightarrow{q} s_1' \quad s_2 \xrightarrow{r} s_2'}{s_1 s_2 \xrightarrow{qr} s_1' s_2'}, \qquad \frac{s_1 \xrightarrow{q} s_1' \quad s_2 \not\xrightarrow{r}}{s_1 s_2 \xrightarrow{q} s_1' s_2}. \tag{4}$$



(a) **(b)** **(c)**

Fig. 1. Component behaviours for Example 2

The composed component $gl(C_1, C_2)$ is shown in Figure 1c. The dashed arrows show the transitions of the component obtained by composing C_1 and C_2 with the most liberal parallel composition operator, allowing any combination of transitions of the two components. Solid arrows show the transitions of $gl(C_1, C_2)$.

Among the transitions labeled by q, only the transition $22 \xrightarrow{q} 32$ is enabled and not $21 \xrightarrow{q} 31$ (Figure 1c). Indeed, the negative premise in the third rule of (4) suppresses the interaction when a transition labeled r is possible in the second component. Here, this results in giving $21 \xrightarrow{qr} 32$ "higher priority" over $21 \xrightarrow{q} 31$. Notice that, in the state 22 of $gl(C_1, C_2)$, r is no longer possible, i.e. $2 \not\xrightarrow{r}$ in C_2. Hence, the third rule of (4) applies and we have $22 \xrightarrow{q} 32$. □

Priorities are presented in more detail in Section 2.5, below.

2.3 Properties of Glue

Glue operators must meet the following requirements.

Incrementality. If a composite component is of the form $gl(C_1, C_2, \ldots, C_n)$ for $n \geq 2$, then there exist glue operators gl_1 and gl_2 such that

$$gl(C_1, C_2, \ldots, C_n) \cong gl_1\big(C_1, gl_2(C_2, \ldots, C_n)\big)\,.$$

Incrementality is a kind of generalised associativity[2]. It requires that coordination between n components can be expressed by first coordinating $n-1$ components and then by coordinating the resulting component with the remaining argument.

[2] Notice that, for any permutation $\sigma : [1,n] \to [1,n]$, one can define a glue operator $gl_\sigma(C_1, \ldots, C_2) \stackrel{def}{=} gl\big(C_{\sigma(1)}, \ldots, C_{\sigma(n)}\big)$. Applying incrementality to gl_σ with the permutation $\sigma = (2, 3, \ldots, i, 1, i+1, \ldots, n)$, we conclude that there must exist glue operators gl_1 and gl_2 such that $gl(C_1, \ldots, C_i, \ldots C_n) = gl_\sigma(C_i, C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n) = gl_1\big(C_i, gl_2(C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n)\big)$, for any $i \in [1,n]$.

Flattening. Conversely, $\mathcal{G}$ must be closed under composition, i.e. if a composite component is of the form $gl_1(C_1, gl_2(C_2, \ldots, C_n))$ then there exists an operator gl such that

$$gl_1\big(C_1, gl_2(C_2, \ldots, C_n)\big) \cong gl(C_1, C_2, \ldots, C_n)\,.$$

This property is essential for separating behaviour from glue and treating glue as an independent entity that can be studied and analysed separately. Flattening enables model transformations, e.g. for optimising code generation or component placement on multicore platforms [18,20].

Compositionality. The equivalence relation $\cong$ must be a congruence with respect to the glue operators. For all $gl \in \mathcal{G}$, all $C, C_1, \ldots, C_n \in \mathcal{C}$ and $i \in [1, n]$,

$$C_i \cong C \qquad \text{must imply} \qquad gl(C_1, \ldots, C_i, \ldots, C_n) \cong gl(C_1, \ldots C, \ldots, C_n)\,.$$

Compositionality is fundamental for reasoning about systems. It allows considering properties of components in isolation and separately from the properties of glue operators to infer global properties of the system by construction. Furthermore, compositionality allows component providers to protect their intellectual assets by providing only an abstract specification of a component—any observationally equivalent implementation can then be substituted without affecting the semantics of the system.

It can be shown that glue operators defined by SOS rules, as in (1), are always compositional if the equivalence relation $\cong$ is compatible with strong bisimulation (recall the assumption of Section 2.2).

It should be noted that almost all existing frameworks fail to meet all three requirements. Process algebras are based on two composition operators (some form of parallel composition and hiding) which are orthogonal to behaviour, but fail to meet the flattening requirement as formulated above: in order to flatten a composite component, the operand components might have to be modified or additional components (e.g. context) might need to be introduced. General component frameworks, such as [2,24], adopt more expressive notions of composition by allowing the use of behaviour for coordination between components and thus do not separate behaviour from interaction. Furthermore, most of these frameworks are hardly amenable to formalisation through operational semantics.

2.4 Expressiveness of Glue

Comparison between different formalisms and models is often made disregarding their structure and reducing them to behaviourally equivalent formalisms, such as Turing machine. This leads to a notion of expressiveness which is not adequate for the comparison of high-level languages. All programming languages are deemed equivalent (Turing-complete) disregarding their adequacy for solving problems. For component frameworks separation between behaviour and coordination mechanisms is essential.

A notion of expressiveness for component frameworks characterising their ability to coordinate components is proposed in [15]. It allows the comparison of two component frameworks with glues $\mathcal{G}$ and $\mathcal{G}'$ respectively, the same set of components and equipped with the same congruence relation $\cong$.

We say that $\mathcal{G}'$ is *more expressive* than $\mathcal{G}$—denoted $\mathcal{G} \preccurlyeq \mathcal{G}'$—if, for any composite component $gl(C_1, \ldots, C_n)$ obtained by using $gl \in \mathcal{G}$, there exists $gl' \in \mathcal{G}'$, such that $gl(C_1, \ldots, C_n) \cong gl'(C_1, \ldots, C_n)$. That is, any coordination expressed by using $\mathcal{G}$ can be expressed by using $\mathcal{G}'$.

Example 3. Let P be a set of ports and consider two glues Bin and Ter generated respectively by families of binary and ternary rendezvous operators: $rdv^{(2)}_{a,b}$ and $rdv^{(3)}_{a,b,c}$, defined by the following rules (for all interactions $a, b, c \in 2^P$):

$$rdv^{(2)}_{a,b} : \frac{s_1 \xrightarrow{a} s_1' \quad s_2 \xrightarrow{b} s_2'}{s_1 s_2 \xrightarrow{ab} s_1' s_2'}, \qquad rdv^{(3)}_{a,b,c} : \frac{s_1 \xrightarrow{a} s_1' \quad s_2 \xrightarrow{b} s_2' \quad s_3 \xrightarrow{c} s_3'}{s_1 s_2 s_3 \xrightarrow{abc} s_1' s_2' s_3'}. \tag{5}$$

Clearly, $Ter \preccurlyeq Bin$. Indeed, for any $a, b, c \in 2^P$, and any $C_1, C_2, C_3 \in \mathcal{C}$, we have $rdv^{(3)}_{a,b,c}(C_1, C_2, C_3) \cong rdv^{(2)}_{a,bc}\big(C_1, rdv^{(2)}_{b,c}(C_2, C_3)\big)$. On the contrary, $Bin \not\preccurlyeq Ter$, since any two components at any given state can only perform two actions (one action each), whereas three are needed for a ternary synchronisation. □

We call *universal glue* the set $\mathcal{G}_{univ}$, which contains all glue operators that can be defined by the rules similar to (1) in the general form defined in [15] (see also Note 1). An interesting question is whether the expressiveness of $\mathcal{G}_{univ}$ can be achieved with a minimal set of operators. Results in [15] bring a positive answer to this question. It is shown that the glue of the BIP framework [6] combining two classes of operators, interactions and priorities, is as expressive as $\mathcal{G}_{univ}$. Furthermore, this glue is minimal in the sense that it loses universal expressiveness if either interactions or priorities are removed.

A consequence of these results is that most existing formal frameworks using only interaction such as process algebras are less expressive. This comparison can be strengthened by using the following weaker notion of expressiveness.

Often component frameworks consider certain behaviours, such as, for instance, FIFO buffers, to be part of the coordination primitives. To address such cases, we introduce a weaker form of expressiveness comparison. We say that $\mathcal{G}'$ is *weakly more expressive* than $\mathcal{G}$—denoted $\mathcal{G} \preccurlyeq_W \mathcal{G}'$—if there exists a *finite* set of *coordinating components* $\mathcal{D} \subseteq \mathcal{C}$, such that, for any component $gl(C_1, \ldots, C_n)$ with $gl \in \mathcal{G}$ there exist $gl' \in \mathcal{G}'$ and $D_1, \ldots, D_k \in \mathcal{D}$, such that $gl(C_1, \ldots, C_n) \cong gl'(C_1, \ldots, C_n, D_1, \ldots, D_k)$. That is, to realise the same coordination as gl, additional behaviour is needed. The term "weakly more expressive" is justified by the observation that, taking $\mathcal{D} = \emptyset$, $\mathcal{G} \preccurlyeq \mathcal{G}'$ clearly implies $\mathcal{G} \preccurlyeq_W \mathcal{G}'$.

Example 4. Taking on Example 3, it is clear that $Bin \preccurlyeq_W Ter$. Indeed, let $D = \big(\{*\}, \{\tau\}, \{* \xrightarrow{\tau} *\}\big)$ (with $\tau \notin P$) be the only coordinating component. Considering τ as the "silent" action, it is easy to see that, for all $a, b \in 2^P$ and

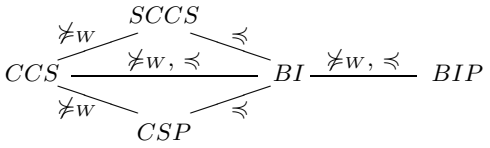


Fig. 2. Summary of relations between glues

$C_1, C_2 \in \mathcal{C}$, we have $rdv^{(2)}_{a,b}(C_1, C_2) \cong rdv^{(3)}_{\tau,a,b}(D, C_1, C_2)$. Therefore, we say that Bin and Ter are *weakly equivalent.* □

It can be shown that glues including only interactions fail to match universal expressiveness even under this definition. Adding new atomic components does not suffice if the behaviour of the composed components is not modified.

Relations between the glues of BIP (see Section 2.5 below) and classical process algebras, namely CCS [34], SCCS [33] and CSP [26], which were obtained in [15], are summarised in Figure 2. BI denotes the BIP glue without priorities.

2.5 The BIP Component Model

In the light of the above results the BIP component model has been defined in [6,14]. BIP uses two types of glue. Given a set of atomic components $C_1, \ldots, C_n$ a composite component is modelled by an expression of the form $\pi\gamma(C_1, \ldots, C_n)$ where γ is a set of interactions and π a priority relation.

Let $C_i = (\Sigma_i, P_i, \rightarrow)$, for $i \in [1, n]$, with disjoint sets of ports, i.e. $P_i \cap P_j = \emptyset$, for $i \neq j$ and denote $P = \bigcup_{i=1}^{n} P_i$. The glue operator corresponding to a set of interactions $\gamma \subseteq 2^P$ is defined by the following set of rules in the format generalising (1) (see Note 1):

$$\frac{\left\{s_i \xrightarrow{a \cap P_i} s'_i\right\}_{i \in I} \quad \left\{s_i = s'_i\right\}_{i \notin I}}{s_1 \ldots s_n \xrightarrow{a} s'_1 \ldots s'_n}, \qquad \text{for all } a \in \gamma\,, \tag{6}$$

where $I = \{i \in [1, n] \mid a \cap P_i \neq \emptyset\}$ is the set indexing the components that participate in the interaction. Notice that (6) has only positive premises.

Priority is a strict partial order relation $\pi \subseteq 2^P \times 2^P$. For two interactions $a, b \in 2^P$, we write $a \prec b$ as a shorthand for $(a, b) \in \pi$. As described in Section 2.2, priority introduces negative premises in the derivation rules. Intuitively, for an interaction a to be executed, it has to be enabled (cf. (6)) and all interactions with higher priority than a must be disabled.

For an interaction $a \in 2^P$, denote by $\pi(a) = \left\{b \in 2^P \,\middle|\, a \prec b\right\}$ the set of interactions having higher priority than a. For an interaction $b \in \pi(a)$ to be disabled, a corresponding transition must be disabled in at least one of the contributing components. To assign such a component to each $b \in \pi(a)$ we use, in the derivation rules (7) below, indexing functions $j : \pi(a) \rightarrow [1, n]$, such that, for all $b \in \pi(a)$, we have $b \cap P_{j(b)} \neq \emptyset$. Thus the glue operator $\pi\gamma$ is defined by the following set of rules:

$$\frac{\left\{s_i \xrightarrow{a\cap P_i} s_i'\right\}_{i\in I} \quad \left\{s_{j(b)} \not\xrightarrow{b\cap P_{j(b)}}\right\}_{b\in\pi(a)} \quad \left\{s_i = s_i'\right\}_{i\notin I}}{s_1 \ldots s_n \xrightarrow{a} s_1' \ldots s_n'},$$

$$\text{for all } a \in \gamma \text{ and } j : \pi(a) \to [1,n] \text{ such that } \forall b \in \pi(a), \left(b \cap P_{j(b)} \neq \emptyset\right), \quad (7)$$

where $I = \{i \in [1,n] \,|\, a \cap P_i \neq \emptyset\}$ is the set indexing the components that participate in the interaction a. In [15], we have shown that any operator of the universal glue $\mathcal{G}_{univ}$ can be obtained in such manner by combining a priority π and a set of interactions γ.

Besides meeting the universal expressiveness property, BIP meets the incrementality, flattening and compositionality requirements discussed in Section 2.3 (see the detailed discussion in [5]). Glue is a first class entity that can be analysed and composed.

The BIP model is implemented by the BIP language and an extensible toolbox. The BIP language can be considered as a general component coordination language. It leverages on C++ style variables and data type declarations, expressions and statements, and provides additional structural syntactic constructs for defining component behaviour, describing connectors and priorities. Moreover, it provides constructs for dealing with parametric and hierarchical descriptions as well as for expressing timing constraints associated with behaviour. The BIP toolbox includes tools for checking correctness, for source-to-source transformations and for code generation. Correctness can be either formally proven using invariants and abstractions, or tested by using simulation. For the latter case, simulation is driven by a specific middleware, the BIP engine, which allows exploration and inspection of traces corresponding to BIP models. Source-to-source transformations allow static optimizations as well as specific transformations towards implementation, i.e. distribution. Finally, code generation targets different platforms and operating systems support (e.g. distributed, multi-threaded, real-time, for single/multi-core platforms).

In the rest of the paper, we focus on modelling interactions by using connectors as well as the formalisation of architectures and their use for achieving correctness by construction.

3 Connectors and Their Properties

In this section we study connectors as a means for expressing coordination constraints between components. Connector descriptions involve a control part describing interactions and a data transfer part describing data transformations of the interacting components. We provide a principle for the hierarchical structuring of connectors and show how hierarchical connectors can be flattened into equivalent simple connectors. Finally, we propose a formalism for describing dynamic connectors that is currently under study.

3.1 Simple Connectors

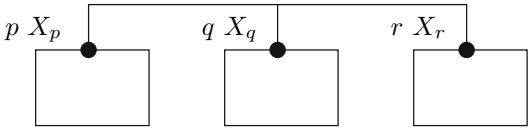
We consider a set of components $\{C_i\}_{i\in I}$ with disjoint sets of ports $\{P_i\}_{i\in I}$. We denote for a set of ports P by X_P the associated set of variables. A connector γ is an expression of the form

$$\gamma \;=\; (a).\big[g(X_a) : X_a := f(X_a)\big]\,,$$

where a is an interaction, that is a set of ports such that $|a \cap P_i| \leq 1$ for all $i \in I$.

The interaction describes the control part of the connector. It is an n-ary atomic strong synchronisation between ports specified by the set of the synchronised ports. The term in brackets consists of a guard on the exported variables followed by an assignment. It describes the data transfer part of the connector. The execution of a connector is possible only if ports involved in the interaction a are enabled in the components and the guard g evaluates to true for the exported values. It consists in modifying the exported variables as specified by the assignment and letting the involved components complete the synchronised transitions.

The figure below depicts a connector between ports p, q and r with associated exported variables X_p, X_q and X_r. An interaction can occur only when at least two of the exported values differ (the guard is true). It is completed by assigning the maximum of their values to the port variables.



$$(pqr).\big[(X_p \neq X_q) \vee (X_p \neq X_r) : X_p, X_q, X_r := \max(X_p, X_q, X_r)\big]$$

Fig. 3. Simple connector

The effect of the application of connectors on a set of components is formally defined in [17].

3.2 Hierarchical Connectors

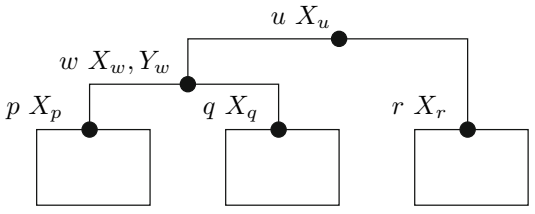
Hierarchical connectors are useful when we want to build systems incrementally. The idea is to equip each connector with a port and an associated variable. The port can be then used further in other connectors, and hence lead to a hierarchical structuring of connectors. Syntactically, a hierarchical connector γ is an expression of the form

$$\gamma = (w \leftarrow a).\big[g(X_a) : (X_w, X_L) := f_{up}(X_a) \,//\, X_a := f_{down}(X_w, X_L)\big]\,.$$

As for simple connectors, the coordination in hierarchical connectors involves two parts. The control part $w \leftarrow a$ defines a dependency relation between the

connector port w and its interaction a. That is, w is enabled if and only if the interaction a is enabled. The data part $\left[g(X_a) : (X_w, X_L) := f_{up}(X_a) \,//\, X_a := f_{down}(X_w, X_L)\right]$ defines the computation realised on local variables X_L and data associated to ports. The interaction is enabled and the computation is performed only if the guard $g(X_a)$ evaluates to true. In this case, computation involves two steps. First, an *up* function f_{up} is used to compute X_w and X_L depending on interaction variables X_a. Second, if an (upper) interaction involving w takes place, the *down* function f_{down} is used to update X_a based on X_w and X_L. Moreover, in an hierarchical connector, execution of up and down steps is coordinated: first, all up steps are performed bottom-up (as long as guards are satisfied), then, if a top-level interaction is executed, all down steps are performed top-down.

As an example, the coordination enforced by the simple connector presented in Figure 3 can be equally obtained by using the hierarchical connector depicted in Figure 4. The ternary connector and its associated data transfer is split in two binary connectors, glued together by the port w.



$(w \leftarrow pq).[\texttt{true} : X_w := \max(X_p, X_q), Y_w := (X_p \neq X_q) \,//\, X_p, X_q := X_w]$
$(u \leftarrow wr).[Y_w \vee (X_w \neq X_r) : X_u := \max(X_w, X_r) \,//\, X_w, X_r := X_u]$

Fig. 4. Hierarchical connector

Hierarchical connectors can be statically flattened, that is, transformed into functionally equivalent simple connectors. For the control part, flattening amounts to substituting the inner connector ports by the associated interactions. For the data part, it reduces to static composition of up and down functions together with propagation of the guards. Flattening has been formally defined as a rewriting system on hierarchical connectors and proven confluent and terminating [17]. As an example, flattening of the connector from Figure 4 transforms it back into the simple connector from Figure 3.

3.3 Dynamic Connectors

How can we reason about architectures whose structure changes dynamically? There exists a variety of paradigms dealing with dynamic change in coordination. One is based on the use of process algebras such as the π-calculus [35]. Nonetheless, there is no clear distinction between behaviour and coordination and thus it is hard to come up with a concept of architecture in this context.

Another considers architectures as graphs and studies their possible configurations by using graph grammars. Technically architecture styles and possible configurations are described by context-free graph grammars [32,36]. This approach implicitly assumes the existence of a global coordinator. Furthermore, the focus is on changing structure and it is not easy to account for data transfer. Other more ad hoc techniques consider that dynamic architectures are just coordinators between components that can modify the architecture connectivity [1]. The approach closest to the one presented below is explored in [21], where *dynamic BI(P)* (BIP without priorities) allows spawning new components and interactions during execution.

We show below how dynamic connectors can be defined as a direct extension of connectors in BIP. We assume that system models are built using arbitrary numbers of typed components. The type T of a component defines its set of ports and associated exported variables. Two kinds of variables can be used in descriptions: 1) component variables c_i with an associated component type T, denoted $c_i{:}T$; 2) variables U_i representing sets of components of the same type T, denoted $U_i{:}T$. We denote by $c.p$ the port p of component c.

A connector description consists of a set of initialisation statements followed by a set of rules. The initialisation statements define initial values of the variables U representing sets of components. The rules define sets of dynamic connectors. The format for the description is the same as for static connectors. The main difference is that the rules may involve guards and computation that modifies the sets of components.

The following example models a ring architecture composed of n elements

$$U := \{c_i : T,\ \underline{for}\ 0 \leq i < n\}\,, \tag{8}$$

$$r_i := (c_i.out, c_{(i+1)\%n}.in).[\texttt{true} :\ X_{c_{(i+1)\%n}.in} := X_{c_i.out}],\ \underline{for}\ 0 \leq i < n\,. \tag{9}$$

Line (8) initialises a variable U with an array of component instances by using the iterator primitive $\underline{for}\ 0 \leq i < n$. Line (9) gives a set of n rules for specifying connectors transferring data from outputs to inputs.

The following example models a set of n components that must strongly synchronise through their port p, with the possibility of disconnecting a component when it detects a failure and the possibility to rejoin the group in case of recovery. The first line creates an array of n instances of components c of type T. The description uses two variables U and U_{act} representing sets of components. The former is used to record the universe of the created components and the latter to record the set of the active components.

The configurations are described by three rules. Rule (10) involves an interaction requiring the synchronisation of all the active components. The corresponding computation consists in assigning to the synchronised port variables the maximum of the exported values. Rule (11) describes disconnection of the i-th component c_i when it detects a failure. Rule (12) describes insertion of a component after recovery.

$$U := \{c_i : T,\ \underline{for}\ 0 \le i < n\}\,, \quad U_{act} := U\,,$$
$$r := (c.p,\ \underline{for}\ c \in U_{act}).[\texttt{true} :\ x_{c.p} := \max\{x_{c.p} \mid c \in U_{act}\},\ \underline{for}\ c \in U_{act}]\,, \tag{10}$$
$$r_c^{fail} := (c.fail).[\texttt{true} :\ U_{act} := U_{act} - c],\ \underline{for}\ c \in U_{act}\,, \tag{11}$$
$$r_c^{join} := (c.join).[\texttt{true} :\ U_{act} := U_{act} + c],\ \underline{for}\ c \in U \setminus U_{act}\,. \tag{12}$$

As a final illustration, consider the Master-Slave example presented in [19]. Systems are constructed from two types of components, respectively masters (M) and slaves (S). Every master m_i requests sequentially two distinct slaves s_j, s_k (rules 13, 14) and then interacts with both of them (rule 15 above). The rules are graphically depicted in Figure 5.

$$U := \{m_i : M,\ s_j : S,\ \ \underline{for}\ 0 \le i < n, 0 \le j < m\}\,,$$
$$req_{ij}^1 := (m_i.req\ s_j.get)[x_{m_i.req} = \emptyset : x_{m_i.req} := x_{m_i.req} \cup s_j], \tag{13}$$
$$\underline{for}\ 0 \le i < n, 0 \le j < m$$
$$req_{ik}^2 := (m_i.req\ s_k.get)[s_k \notin x_{m_i.req} : x_{m_i.req} := x_{m_i.req} \cup s_k], \tag{14}$$
$$\underline{for}\ 0 \le i < n, 0 \le k < m$$
$$comp_{ijk} := (m_i.comp\ s_j.work\ s_k.work)[s_j, s_k \in x_{m_i.req} : x_{m_i.req} := \emptyset], \tag{15}$$
$$\underline{for}\ 0 \le i < n, 0 \le j, k < m$$

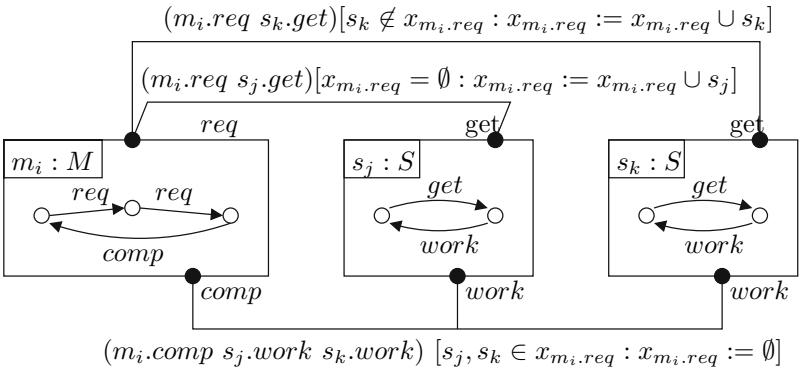


Fig. 5. Dynamic Connectors for the Master-Slave example

4 Achieving Correctness

We present two approaches for achieving correctness for component-based systems. The first is by compositional inference of global properties of a composite component from properties of its constituents and synchronisation constraints implied by composition operators. The second is by using and composing architectures that enforce specific coordination properties.

4.1 Compositional Verification

Compositional verification techniques are used to cope with state explosion in concurrent systems. The idea is to apply divide-and-conquer approaches to infer global properties of complex systems from properties of their components. Separate verification of components limits state explosion. Nonetheless, components mutually interact in a system and their behaviour and properties are inter-related. This is a major difficulty in designing compositional techniques. We developed for BIP a compositional verification method [11,10,9] for safety properties (invariants) based on the following rule:

$$\frac{\left\{C_i \models \Box\Phi_i\right\}_i \quad \Psi \in II\big(\gamma,(C_i)_i\big) \quad \big(\bigwedge_i \Phi_i\big) \wedge \Psi \Rightarrow \Phi}{\gamma((C_i)_i) \models \Box\Phi}. \tag{16}$$

This rule allows one to prove invariance of Φ for systems $\gamma((C_i)_i)$ constructed by using a parallel composition operation parameterised by a set of connectors γ on a set of components $(C_i)_i$. It relies on computing auxiliary invariants as the conjunction of component invariants Φ_i and an interaction invariant Ψ. Component invariants Φ_i are computed locally for components C_i, hence, they satisfy $C_i \models \Box\Phi_i$, for all is. Interaction invariants Ψ expresses constraints on the global state space induced by interactions between components. They are obtained automatically from finite-state abstractions of the system to be verified and without explicitely constructing the product space, that is, denoted by $\Psi \in II\big(\gamma,(C_i)_i\big)$. Finally, if the implication $\big(\bigwedge_i \Phi_i\big) \wedge \Psi \Rightarrow \Phi$ holds, i.e. can be effectively proven by using a SAT/SMT solver, then Φ is an invariant of the composed system.

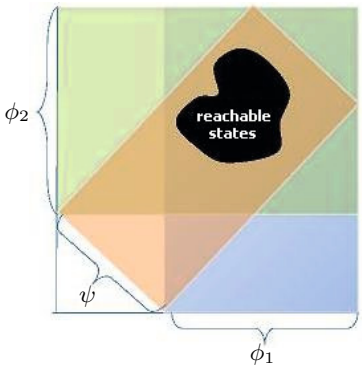


Fig. 6. Rule illustrated

The principle of the rule is graphically illustrated in Figure 6 for two components C_1, C_2 assuming that each dimension corresponds to the state space of each component. Component invariants define restrictions represented as a vertical and a horizontal strip. The intersection of component invariants is a rectangular area including all the states of the Cartesian product of the sets of states meeting each invariant. The restriction induced by interaction invariants is an oblique strip that removes states of the rectangular area that are forbidden by the interactions.

As a concrete illustration, let us consider a simple benchmark example from [11]. The *Temperature Control System* models the control of the coolant temperature in a reactor tank by moving two independent refrigerating rods. The goal is to maintain the coolant between the temperatures $\theta_m = 100°\text{C}$ and $\theta_M = 1000°\text{C}$. When the temperature reaches its maximum value θ_M, the tank must be refrigerated with one of the rods. The temperature rises at a rate

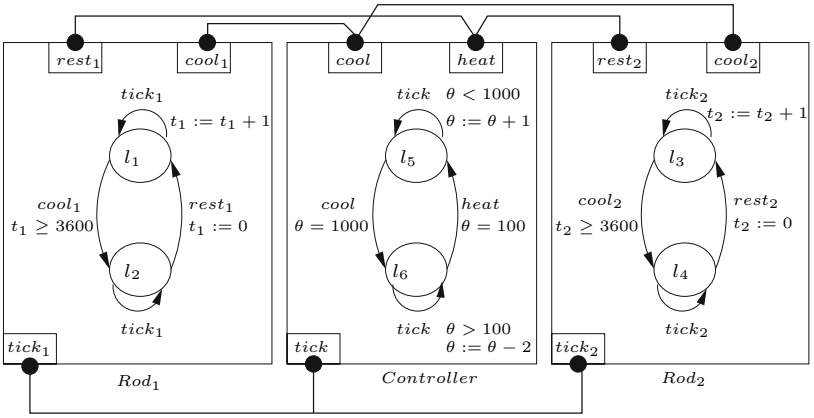


Fig. 7. Temperature Control System in BIP

$v_r = 1°\mathrm{C}/s$ and decreases at rate $v_d = 2°\mathrm{C}/s$. A rod can be moved again only if $T = 3600s$ has elapsed since the end of its previous movement. If the temperature of the coolant cannot decrease because there is no available rod, a complete shutdown is required. A discretised time model of the Temperature Control System in BIP is provided in Figure 7. The model consists of three atomic components, a *Controller* handling the temperature and two components Rod_1, Rod_2 modelling the rods. The variable θ within the *Controller* stores the temperature of the reactor. Its evolution depends on the state respectively, at l_5 (heating) it increases by one every time unit and at l_6 (cooling) it decreases by 2 every time unit. The transitions between states depend on the value of θ, as explained earlier. The $Rod_{1,2}$ components are identical. The $t_{1,2}$ variables are discrete clocks measuring the resting time. They increase by one every time unit. A rod can be used for cooling only when the resting time is greater than 3600. The *Controller* and the *Rods* are interconnected by five connectors ($tick\ tick_1\ tick_2$), ($cool\ cool_1$), ($cool\ cool_2$), ($heat\ rest_1$), ($heat\ rest_2$) modelling respectively, the discrete time progress and the usage/releasing of the rods. In the BIP model, complete shutdown corresponds to a deadlock situation, henceforth, checking for functional correctness amounts to checking deadlock-freedom. The invariants computed on the BIP model are as follows:

$$\begin{aligned}
\Phi_{Controller} &= (at_l_5 \wedge 100 \leq \theta \leq 1000) \vee (at_l_6 \wedge 100 \leq \theta \leq 1000) \\
\Phi_{Rod1} &= (at_l_1 \wedge t_1 \geq 0) \vee (at_l_2 \wedge t_1 \geq 3600) \\
\Phi_{Rod2} &= (at_l_3 \wedge t_2 \geq 0) \vee (at_l_4 \wedge t_2 \geq 3600) \\
\Psi &= (at_l_2 \vee at_l_4 \vee at_l_5) \wedge (at_l_1 \vee at_l_3 \vee at_l_6)
\end{aligned}$$

As explained in [11], deadlock-freedom of BIP models can be characterised as an invariant state property. For our example, potential deadlocks states include, e.g.

$$D_1 = (at\_l_1 \wedge t_1 < 3600) \wedge (at\_l_3 \wedge t_2 < 3600) \wedge (at\_l_6 \wedge \theta = 100)$$
$$D_2 = (at\_l_1 \wedge t_1 < 3600) \wedge (at\_l_3 \wedge t_2 < 3600) \wedge (at\_l_5 \wedge \theta = 1000)$$

Proving deadlock-freedom amounts to checking that no states within D_1 or D_2 are reachable, or equivalently, that both $\Phi_1 = \neg D_1$ and $\Phi_2 = \neg D_2$ are invariants. Using a SAT solver it can be checked that the following assertion holds

$$(\Phi_{Controller} \wedge \Phi_{Rod1} \wedge \Phi_{Rod2} \wedge \Psi) \Rightarrow \Phi_1$$

therefore Φ_1 is a system invariant and all deadlock states within D_1 are unreachable. But, the implication above does not hold when Φ_2 is considered instead of Φ_1. This means that Φ_2 cannot be proven invariant and hence deadlock states in D_2 are potentially reachable. In this case, complementary verification techniques, e.g. backward reachability analysis, can be used to confirm/infirm their reachability in the model.

Table 1 taken from [10] provides an overview of experimental results obtained for several benchmarks. For the columns: n is the number of BIP components in the example, q is the total number of control locations, x is the total number of boolean and integer variables, D provides, when possible, the estimated number of deadlock configurations, D_c (resp. D_{ci}) is the number of deadlock configurations remaining once component respectively interaction invariants are used and t is the total time for computing invariants and checking for satisfiability.

Table 1. Checking deadlock-freedom on classical benchmarks

example	n	q	x	D	D_c	D_{ci}	t
Temperature Control System (2 rods)	3	6	3	8	5	3	3s
Temperature Control System (4 rods)	5	10	5	32	17	15	6s
Readers-Writer (7000 readers)	7002	14006	1	-	-	0	17m27s
Readers-Writer (10000 readers)	10002	20006	1	-	-	0	36m10s
Gas station (100 pumps - 1000 customers)	1101	4302	0	-	-	0	9m14s
Philosophers (2000 Philos)	4000	10000	0	-	-	3	32m14s
Philosophers (3001 Philos)	6001	15005	0	-	-	1	54m34s

The original method from [11] has been extended in several directions. Incremental extensions, where invariants and properties are established along the model construction, have been studied in [8,7]. Moreover, it has been combined with backward reachability analysis and automatic strengthening of invariants for elimination of false positives [12]. More recently, the method has been extended to timed models and timed properties [3].

4.2 Property Enforcement—Architectures

Property enforcement consists in applying architectures to restrict the behaviour of a set of components so that the resulting behaviour meets a given property.

Depending on the expressiveness of the glue operators, it may be necessary to use additional components to achieve a coordination to satisfy the property.

Architectures depict design principles, paradigms that can be understood by all, allow thinking on a higher plane and avoiding low-level mistakes. They are a means for ensuring global properties characterising the coordination between components—correctness for free. Using architectures is key to ensuring trustworthiness and optimisation in networks, OS, middleware, HW devices etc.

System developers extensively use libraries of reference architectures ensuring both functional and non-functional properties, for example fault-tolerant architectures, architectures for resource management and QoS control, time-triggered architectures, security architectures and adaptive architectures. The proposed definition is general and can be applied not only to hardware or software architectures but also to protocols, distributed algorithms, schedulers, etc.

An architecture is a partial operator $A : \mathcal{C}^n \to \mathcal{C}$, imposing a characteristic property Φ and defined by a glue operator gl and a set of coordinating components $\mathcal{D}$, such that:

- A transforms a set of components $C_1, \ldots, C_n$ into a composite component $A[C_1, \ldots, C_n] = gl(C_1, \ldots, C_n, \mathcal{D})$;
- $A[C_1, \ldots, C_n]$ meets the characteristic property Φ.

An architecture is a solution to a coordination problem specified by Φ, using a particular set of interactions specified by gl. It is a partial operator, since the interactions of gl should match actions of the composed components.

Application and platform restrictions entail reduced expressiveness of the glue operator gl that must be compensated by using the additional set of components $\mathcal{D}$ for coordination. For instance, glue operators defined by connectors (cf. Sections 3.1–3.3) are memoryless. Hence, they can only be used to impose state properties. Imposing more complex safety properties requires additional coordination behaviour. Similarly, for distributed architectures, interactions are point-to-point by asynchronous message passing. Synchronisation among the components is achieved by stateful protocols.

The characteristic property assigns a meaning to the architecture that can be informally understood without the need for explicit formalisation (e.g. mutual exclusion, scheduling policy, clock synchronisation).

In addition to imposing the characteristic property, an architecture must preserve essential properties of the composed components. In particular, any invariant of a component C_i must be an invariant of $A[C_1, \ldots, C_n]$. In Section 4.3, we provide results about preservation of safety and liveness properties by architecture composition. Since there exists a unary identity architecture, which does not modify the behaviour of its operand, preservation of properties by architectures follows from that by architecture composition.

Architectures should, in principle, preserve deadlock-freedom: if components C_i are deadlock-free then $A[C_1, \ldots, C_n]$ should be deadlock-free too. However, in general, preservation of deadlock-freedom cannot be guaranteed by construction, since architectures restrict the behaviour of components they are applied to.

Instead, deadlock-freedom has to be verified a posteriori using techniques such as the one presented in Section 4.1.

4.3 Property Composability

In a design process it is often necessary to combine more than one architectural solution on a set of components to achieve a global property. System engineers use libraries of solutions to specific problems and they need methods for combining them without jeopardising their characteristic properties.

For example, a fault-tolerant architecture combines a set of features building protections against trustworthiness violations. These include 1) triple modular redundancy mechanisms ensuring continuous operation in case of single component failure; 2) hardware checks to be sure that programs use data only in their defined regions of memory, so that there is no possibility of interference; 3) default to least privilege (least sharing) to enforce file protection. Is it possible to obtain a single fault-tolerant architecture consistently combining these features? The key issue here is feature interaction in the integrated solution. Non-interaction of features is characterised below as property composability based on our concept of architecture.

Consider two architectures A_1, A_2, enforcing respectively properties Φ_1, Φ_2 on components $C_1, \ldots, C_n$. That is, $A_1[C_1, \ldots, C_n]$ and $A_2[C_1, \ldots, C_n]$ satisfy respectively the properties Φ_1, Φ_2. Is it possible to find an architecture $A[C_1, \ldots, C_n]$ that meets both properties? For instance, if A_1 ensures mutual exclusion and A_2 enforces a scheduling policy is it possible to find architectures on the same set of components that satisfies both properties?

A full, rigorous definition of the notions of architecture and property enforcement is provided in [4] alongside a constructive definition of an associative, commutative and idempotent architecture composition operator $\oplus$. An architecture is defined as a triple $A = (\mathcal{D}, P_A, \gamma)$, where $\mathcal{D}$ is a finite set of coordinating components, P_A is a set of ports and $\gamma \subseteq 2^{P_A}$ is an interaction model over P_A. Noticing that the interaction model γ can be represented by the corresponding characteristic predicate φ_γ on variables in P_A, the composition of two architectures $A_1 = (\mathcal{D}_1, P_{A_1}, \gamma_1)$ and $A_2 = (\mathcal{D}_2, P_{A_2}, \gamma_2)$ is defined by putting $A_1 \oplus A_2 \stackrel{def}{=} (\mathcal{D}_1 \cup \mathcal{D}_2, P_{A_1} \cup P_{A_2}, \gamma)$ where γ is such that $\varphi_\gamma = \varphi_{\gamma_1} \wedge \varphi_{\gamma_2}$. The properties of $\oplus$ are studied and applied for building correct-by-construction components incrementally. In particular $\oplus$ has a neutral element A_{id}, which is the most liberal architecture enforcing no coordination constraints.

When applying an architecture A to enforce a property Φ on components $C_1, \ldots, C_n$, the property Φ is expressed in terms of the states of $C_1, \ldots, C_n$. The states of the coordinating components $\mathcal{D}$ (see Section 4.2) are irrelevant. Therefore, we say that an architecture A enforces a property Φ on components $C_1, \ldots, C_n$ if the projection of every trace of $A[C_1, \ldots, C_n]$ onto the state space of $A_{id}[C_1, \ldots, C_n]$ satisfies Φ. In [4], we show that if two architectures A_1 and A_2 enforce the respective *safety* properties Φ_1 and Φ_2 on components $C_1, \ldots, C_n$, then $A_1 \oplus A_2$ enforces on these components the conjunction $\Phi_1 \wedge \Phi_2$ of the two properties.

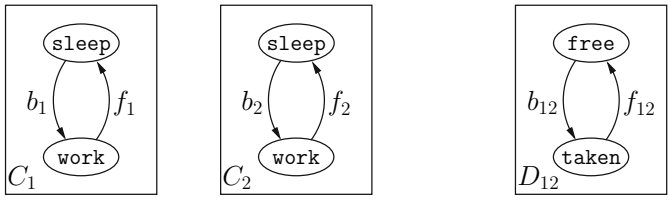


Fig. 8. Components (a) and coordinator (b) for Example 5

Example 5 (Mutual exclusion). Consider the components C_1 and C_2 in Figure 8a. In order to ensure mutual exclusion of their `work` states—$\Phi_{12} = (s_1 \neq \texttt{work} \vee s_2 \neq \texttt{work})$, where s_1 and s_2 are, respectively, state variables of C_1 and C_2—we apply the architecture A_{12} consisting of a coordinating component D_{12}, shown in Figure 8b, and the glue operator defined by the set of interactions and $\gamma_{12} = \{b_1b_{12}, b_2b_{12}, f_1f_{12}, f_2f_{12}\}$ (see Section 2.5).

Assuming that the initial states of C_1 and C_2 are `sleep`, and that of D_{12} is `free`, neither of the two states (`free`, `work`, `work`) and (`taken`, `work`, `work`) is reachable, i.e. the mutual exclusion property Φ_{12} holds in $A_{12}[C_1, C_2]$.

Let C_3 be a third component, similar to C_1 and C_2, with the set of ports $\{b_3, f_3\}$. We define two additional architectures A_{13} and A_{23} similar to A_{12}: they consist, respectively, of coordinating components D_{13} and D_{23}, which, up to the renaming of ports, are the same as D_{12} in Figure 8b, $\gamma_{13} = \{b_1b_{13}, b_3b_{13}, f_1f_{13}, f_3f_{13}\}$ and $\gamma_{23} = \{b_2b_{23}, b_3b_{23}, f_2f_{23}, f_3f_{23}\}$. As above, A_{13} and A_{23} enforce on $A_{13}[C_1, C_3]$ and $A_{23}[C_2, C_3]$, respectively, the mutual exclusion properties $\Phi_{13} = (s_1 \neq \texttt{work} \vee s_3 \neq \texttt{work})$ and $\Phi_{23} = (s_2 \neq \texttt{work} \vee s_3 \neq \texttt{work})$. The composition of the three architectures $A_{12} \oplus A_{13} \oplus A_{23}$, imposing the mutual exclusion property $\Phi_{12} \wedge \Phi_{13} \wedge \Phi_{23} = (s_1 \neq \texttt{work} \wedge s_2 \neq \texttt{work}) \vee (s_2 \neq \texttt{work} \wedge s_3 \neq \texttt{work}) \vee (s_1 \neq \texttt{work} \wedge s_3 \neq \texttt{work})$ on the three components C_1, C_2 and C_3, is given by the set of coordinating components $\{D_{12}, D_{13}, D_{23}\}$ and the set of interactions $\gamma = \{b_1b_{12}b_{13},\ f_1f_{12}f_{13},\ b_2b_{12}b_{23}, f_2f_{12}f_{23},\ b_3b_{13}b_{23},\ f_3f_{13}f_{23}\}$ (see [4] for details). □

One can define a canonical lattice on the set of architectures. The lattice is induced by the partial order relation $<$, defined by putting $A_1 < A_2$ if and only if $A_1 \oplus A_2 \cong A_1$. The neutral architecture A_{id} is the top element of the lattice; the bottom element is the "blocking" architecture, inhibiting all actions of the components, thus leading to a global deadlock.[3] The composition $A_1 \oplus A_2$ is then the greatest lower bound of A_1 and A_2 with respect to $<$. It represents the most liberal architecture enforcing both Φ_1 and Φ_2.

In the above setting, interfering features of a system are translated as contradictory properties. For example, the following two features can be required from an elevator cabin [23,37]:

[3] A deadlocked system trivially satisfies all safety properties.

1. If the elevator is full, it must stop only at floors selected from the cabin and ignore outside calls.
2. Requests from the second floor have priority over all other requests.

Clearly these two requirements are contradictory, since they cannot be jointly satisfied when the elevator is called from the second floor while it is full. Applying the composition of two architectures enforcing respectively these two properties on the components forming the elevator cabin would generate deadlocks.

Thus, although architecture composition $\oplus$ preserves safety properties, it does not preserve deadlock-freedom. Deadlock-freedom can be compositionally verified by techniques such as the one presented in Section 4.1.

The treatment of liveness properties is based on the idea that each coordinator must be "invoked sufficiently often" for the corresponding liveness properties to be imposed on the system as a whole. For each coordinator, one designates the set of its "idle states". It is then required that each coordinator be executed infinitely often, unless, from some point on, it remains forever in an idle state [4]. In [4], it is shown that this notion of liveness is preserved by the composition of architectures, provided that the composed system is deadlock-free and the composed architectures are pairwise non-interfering in the following sense. Architecture A_1 is *non-interfering* w.r.t. architecture A_2 and a set of components $C_1, \ldots, C_n$, if each path in $(A_1 \oplus A_2)[C_1, \ldots, C_n]$, which executes transitions of the coordinators of A_1 infinitely often, either executes transitions of the coordinators of A_2 or visits their idle states infinitely often.[4]

Example 6. Consider the system $(A_{12} \oplus A_{23} \oplus A_{13})[C_1, C_2, C_3]$, as in Example 5. Let each coordinator have a single idle state `free`. Consider the applications of each pair of coordinators, i.e. $(A_{12} \oplus A_{23})[C_1, C_2, C_3]$, $(A_{23} \oplus A_{13})[C_1, C_2, C_3]$ and $(A_{12} \oplus A_{13})[C_1, C_2, C_3]$. For $(A_{12} \oplus A_{23})[C_1, C_2, C_3]$, we observe that along any infinite path, either D_{12} executes infinitely often, or remains forever in its idle state after some point. Hence, A_{23} is non-interfering w.r.t. A_{12} and C_1, C_2, C_3. Likewise for the five other ordered pairs of coordinators. It can be verified that $(A_{12} \oplus A_{23} \oplus A_{13})[C_1, C_2, C_3]$ is deadlock-free. Hence, we conclude that $(A_{12} \oplus A_{23} \oplus A_{13})$ is live. □

Thus, verifying liveness in a composed system is reduced to checking the deadlock-freedom and pairwise non-interference of architectures, both of which can be performed compositionally.

To put the above vision for correctness into practice, we need to develop a repository of reference architectures. The repository should classify existing architectures according to their characteristic properties. There exists a plethora of results on distributed algorithms, protocols, and scheduling algorithms. Most of these results focus on principles of solutions and discard essential operational details. Their correctness is usually established by assume/guarantee reasoning: a characteristic global property is implied from properties of the integrated components. This is enough to validate the principle but does not entail correctness

[4] Notice that the "non-interference w.r.t." relation is not commutative.

of particular implementations. Often, these principles of solutions do not specify concrete coordination mechanisms (e.g. in terms of operational semantics), and ignore physical resources such as time, memory and energy. The reference architectures included in the repository, should be

- described as executable models in the chosen component framework;
- proven correct with respect to their characteristic properties;
- characterised in terms of performance, efficiency and other essential non-functional properties.

For enhanced reuse, reference architectures should be classified according to their characteristic properties. A list of these properties can be established; for instance, architectures for mutual exclusion, time-triggered, security, fault-tolerance, clock synchronisation, adaptive, scheduling, etc. Is it possible to find a taxonomy induced by a hierarchy of characteristic properties? Moreover, is it possible to determine a minimal set of basic properties and corresponding architectural solutions from which more general properties and their corresponding architectures can be obtained?

The example of the decomposition of fault-tolerant architectures into basic features can be applied to other architectures. Time-triggered architectures usually combine a clock synchronisation algorithm and a leader election algorithm. Security architectures integrate a variety of mitigation mechanisms for intrusion detection, intrusion protection, sampling, embedded cryptography, integrity checking, etc. Communication protocols combine sets of algorithms for signalling, authentication and error detection/correction. Is it possible to obtain by incremental composition of features and their characteristic properties, architectural solutions that meet given global properties? This is an open problem whose solution would greatly enhance our capability to develop systems that are correct-by-construction and integrate only the features needed for a target characteristic property.

5 Architecture Specification

So far we have focused on modelling component-based systems and on methods for proving their behavioural correctness. In this section, we study logics for the specification of properties of architectures. Notice that the presented architecture modelling adopts an imperative description style: the coordination between components is given by a set of connectors. No interaction is allowed except the ones specified by connectors. In contrast, logics adopt a declarative style. A logical specification is the conjunction of formulas; its meaning is the set of the models belonging to the intersection of the meanings of the formulas. Consequently, logical specifications characterise not a single model but a set of models that may be empty. In the latter case, the specification is inconsistent.

Typically, an architecture defines a set of interactions between types of components. On the contrary, a class of architectures, what is usually called an architecture style, is represented by a set of congurations. We propose two types

of logics for architectures: 1) Interaction logics to specify a particular architecture as the set of the allowed interactions; 2) Configuration logics to specify families of architectures as the set of the allowed configurations of interactions.

Configurations are defined as follows. Given a set of ports P an interaction a is a subset of P; there exist $2^{|P|}$ interactions on P. A configuration is a set of interactions $a_1, \ldots, a_n$ represented by a term of the form $a_1 + \cdots + a_n$ where $+$ is an associative commutative and idempotent operator. Notice that there exist $2^{2^{|P|}}$ configurations on the alphabet P. For instance, if $P = \{p, q\}$ then the set of non-empty interactions is $\{p, q, pq\}$ and the set of non-empty configurations is $\{p, q, pq, p + q, pq + p, pq + q, pq + p + q\}$.

For example, it is shown in the next subsection that the dynamic Master/Slave architecture presented in Section 3.3 can be specified in Interaction Logic. The class of Master/Slave architectures can be characterized by a formula of the configuration logic that specifies all the allowed configurations of interactions involving some master and slaves.

5.1 Interaction Logics

Let P be an alphabet of ports. The set of the formulas of the propositional interaction logic $PIL(P)$ is defined by the syntax:

$$f ::= \mathtt{true} \,|\, p \in P \,|\, f \wedge f \,|\, \overline{f}\,. \tag{17}$$

The models of the logic are interactions a on P. The semantics defined by the following satisfaction relation $\models^i$.

$$\begin{aligned}
&a \models^i \mathtt{true}, && \text{for any } a,\\
&a \models^i p, && \text{if } p \in a,\\
&a \models^i f_1 \wedge f_2, && \text{if } (a \models^i f_1) \wedge (a \models^i f_2),\\
&a \models^i \overline{f}, && \text{if } (a \models^i f) \text{ does not hold.}
\end{aligned}$$

We use the logical connectives $\vee$ and $\Rightarrow$ with the usual meaning. Notice that the formulas of the logic can be put in the form of the disjunction of monomials $\bigwedge_{p\in I} p \wedge \bigwedge_{p\in J} \overline{p}$, such that $I \cap J = \emptyset$. An interaction a is characterised by the monomial $\bigwedge_{p\in a} p \wedge \bigwedge_{p\notin a} \overline{p}$. Propositional interaction logic has been extensively studied in [16] where it is shown that it can provide a basis for the efficient representation of connectors. For example, the interaction between p_1, p_2 and p_3 is defined by the formula $f_1 = (p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_3) \wedge (p_3 \Rightarrow p_1)$. Broadcast from a sending port s towards receiving ports r_1 and r_2 is defined by the formula $f_2 = (p_1 \Rightarrow s) \wedge (p_2 \Rightarrow s)$. Notice that the non-empty solutions are the interactions s, sp_1, sp_2, sp_1p_2.

In [19], we have shown that $PIL(P)$ can be extended into a first order logic to represent architectures built from arbitrary numbers of components, instantiating a finite number of component types. We present a slightly different version

of this logic. As in [19], we assume that system specifications are built using arbitrary numbers of typed components. The type T of a component defines its set of ports and associated exported variables. Furthermore, we consider a set of component variables c_i with associated component types T. The fact that the component variable c_i is of type T is denoted by $c_i : T$. The syntax of the formulas of the first order interaction logic is defined by:

$$f ::= \texttt{true} \mid c.p \mid c = c' \mid f \wedge f \mid \overline{f} \mid \forall c{:}T.f\,, \tag{18}$$

where c and c' are component variables.

In this definition, T denotes a component type. Each component type represents a set of component instances with identical interface and behaviour. The variables c and c' range over component instances. They are strongly typed and, moreover, they can be tested for equality. The semantics of the logic can be derived from the semantics of the propositional logic as follows.

A formula of the logic defines the set of the interactions of a system built from known instances of typed components. Quantifiers can be eliminated by using the identity: $\forall c{:}T.F(c) \equiv F(t_1) \wedge \cdots \wedge F(t_k)$, where $t_1, \ldots, t_k$ are the instances of components of type T in the model. After quantifier elimination, we get a formula of the propositional logic. This logic can be used to specify dynamic architectures. For instance the formula $\forall c : \texttt{Sender}.\exists c' : \texttt{Receiver}.(c.send \wedge c'.receive)$, means that for any `Sender` there exists a `Receiver` such that their ports *send* and *receive*, respectively, interact. Relevant specification examples using this logic are provided in [19]. Furthermore, it is shown that for a given model the specified interactions can be computed efficiently by using a symbolic representation.

We provide logical specifications for the architecture of the Master-Slave example already seen in Section 3.3. Following the approach in [19], we introduce some additional notations that prove to be very useful for writing specifications:

$$\begin{aligned}
Y.p \ \textit{requires}\ R.q &\equiv \forall c : Y.\ \exists c' : R.\ (c.p \Rightarrow c'.q) \\
&\quad \text{(every } p \text{ port requires a } q \text{ port for interaction)} \\
Y.p \ \textit{accepts}\ R.q &\equiv \forall c : Y. \bigwedge_{(T,r)\neq(R,q)} \forall c' : T.((c.p \neq c'.r) \Rightarrow \overline{c'.r}) \\
&\quad \text{(every } p \text{ port can only interact with } q \text{ ports)} \\
\textit{unique}\ Y.p &\equiv \forall c : Y.\ \forall c' : Y.(c.p \wedge c'.p \Rightarrow c = c') \\
&\quad \text{(no interaction between ports } p\text{)}
\end{aligned}$$

Using the above abbreviations the architecture of the Master-Slave example is described by the following interaction logic formula:

$$\begin{aligned}
&(M.req \ \textit{requires}\ S.get) \wedge (M.req \ \textit{accepts}\ S.get) \wedge (\textit{unique}\ S.get) \\
&(S.get \ \textit{requires}\ M.req) \wedge (S.get \ \textit{accepts}\ M.req) \wedge (\textit{unique}\ M.req) \\
&(M.comp \ \textit{requires}\ S.work) \wedge (M.comp \ \textit{accepts}\ S.work) \\
&(S.work \ \textit{requires}\ M.comp) \wedge (S.work \ \textit{accepts}\ M.comp) \wedge (\textit{unique}\ M.comp)
\end{aligned}$$

Notice the difference in the description styles for the same example. When connectors are used, the style is imperative. The set of the interactions is constructed by enumerating connectors. When formulas are used, the style is declarative. The set of the interactions is in the intersection of the meanings of formulas which express constraints on the interactions required and accepted by each component. It has been shown that the two approaches are equivalent as long as we deal with interactions without data transfer. The association of computation and data transfer with formulas is not as natural as for connectors and raises methodological and technical issues.

The two styles correspond to two different approaches for eliciting architectural knowledge [22]. One is bottom-up and is adopted for building architectural models in various architecture description languages [28]. The other is top-down and is used to capture essential dependencies between features.

5.2 Configuration Logics

Let P be an alphabet on ports. The set of the formulas of the propositional configuration logic $PCL(P)$ is defined by the syntax:

$$f ::= \mathtt{true} \,|\, m \in PIL(P) \,|\, f \wedge f \,|\, \neg f \,|\, f + f \,, \tag{19}$$

where m is a monomial of the interaction logic.

The models of the logic are configurations γ on P, of the form $\gamma = a_1 + \cdots + a_n$ where the a_i's are interactions on P. The semantics is defined by the satisfaction relation $\models$.

$$\begin{aligned} &\gamma = a_1 + \cdots + a_n \models m, && \text{if, for each } a_i,\ a_i \overset{i}{\models} m, \\ &\gamma = a_1 + \cdots + a_n \models f_1 + f_2, && \text{if, for each } a_i,\ (a_i \models f_1) \text{ or } (a_i \models f_2)\,, \end{aligned}$$

where m is a monomial of the interaction logic. For logical constants and connectives we take the standard meaning.

Notice the overloading of the $+$ operator. The meaning of the formula $f_1 + f_2$ is the set of the configurations obtained by combining some configuration satisfying f_1 with some configuration satisfying f_2. In particular, we have the property: $f_1 + (f_2 \vee f_3) = (f_1 + f_2) \vee (f_1 + f_3)$.

A simple example illustrates the expressive power of this logic. Let $P = \{p, q, r, s\}$ be an alphabet of ports. The monomial $p \wedge q \wedge \overline{r}$ specifies, in the interaction logic, the set of interactions pq and pqs. In the configuration logic, it specifies the set of configurations pq, pqs and $pq + pqs$. The formula $p \wedge q \wedge \overline{r} + \mathtt{true}$ characterises all the configurations of the form $\gamma = \gamma_1 + \gamma_2$, where γ_1 satisfies $p \wedge q \wedge \overline{r}$ and γ_2 is an arbitrary configuration. Notice, in particular, that $\mathtt{true}$ is not an absorbing element for $+$. Hence, γ_1 cannot be empty.

In general, a formula of the form $f + \mathtt{true}$ characterises all the configurations comprising the configurations satisfying f. This type of formulas is particularly useful for writing specifications. We write $\sim f = f + \mathtt{true}$ for any formula f of the logic. The operator $\sim$ is idempotent and satisfies the following property: $\sim f \wedge \sim g = \sim(f + g)$ for any formulas f and g.

We extend $PCL(P)$ into a second order logic. We assume that system models are built using arbitrary numbers of typed components. The type T of a component defines its set of ports and associated exported variables. We consider a set of component variables c_i with an associated component type T. The fact that the component variable c_i is of type T is denoted by $c_i : T$. Furthermore, we consider a set of variables U_i ranging over sets of components. This set includes a particular variable $\mathcal{U}$ representing the set of all the components of a model. We also adopt the notation $U_i : T$ to signify that all components in the set U_i are of type T.

The syntax of the second order configuration logic formulas is defined by:

$$f ::= \mathtt{true} \mid m \in PIL(P) \mid c = c' \mid c \in U \mid U \subseteq U' \mid f \wedge f \mid \neg f \mid f + f \mid \forall U : T.f \mid \forall c : T.f\,, \quad (20)$$

where m is a monomial, c, c' are component variables and U, U' are variables over sets of components.

The semantics can be derived from the semantics of the propositional logic. For a given model $\gamma(C_1, \ldots, C_n)$, quantifiers can be eliminated in a formula to obtain a formula of the propositional logic.

The specification of a ring architecture composed of components $c : T$ having ports $c.in$ and $c.out$ is the conjunction of the following formulas:

$$\forall c : T.\exists c' : T \sim(c.out = c'.in) \wedge \forall c'' : T(c' \neq c'').\neg \sim(c.out = c''.in)\,, \quad (21)$$

$$\forall c : T.\exists c' : T \sim(c.in = c'.out) \wedge \forall c'' : T(c' \neq c'').\neg \sim(c.in = c''.out)\,, \quad (22)$$

$$\forall U' : T(U' \neq \mathcal{U}).\exists c \in U', c' \in \mathcal{U} \setminus U'.\ \sim(c.out = c'.in)\,. \quad (23)$$

Formula (21) characterises all the configurations such that each output port $c.out$ of a component c is connected to some input port $c.in$ of some other component c' and explicitly excludes connections of $c.out$ with input ports of components other than c'. Formula (22) requires symmetrically connectivity of each input port to a single output port. The two formulas guarantee cyclical connectivity. Formula (23) requires that there exists a single (maximal) cycle. It says that any subset U' of components of the universal set $\mathcal{U}$ has a component that is connected to some component of its complement.

A comparison between the ring architecture model given in Section 3.3 and the above logical specification shows significant differences in both the style of expression (imperative vs. declarative) and the basic connectivity concepts. The model does not allow other configurations than the ones explicitly specified. Logical specifications characterise configurations that include token ring architectures without excluding other compatible connectivity properties.

6 Conclusion

The paper discusses research issues related to the design of component-based systems by distinguishing three main problems. The first problem is modelling

composite components as the composition of atomic components characterised by their interface and behaviour. We propose a general framework for component composition and study expressiveness of families of operators. For universal expressiveness, it is necessary to combine multiparty interaction and priorities. We propose the concept of connector as a means for structuring interaction between components. So far, static connectors and their properties have been thoroughly studied. We present an extension for the description of dynamic connectors that needs to be further studied and validated through application.

The second problem is achieving correctness of component-based systems by application of scalable techniques. We identify two possible avenues. One relies on compositionality principles and proceeds by analysis of the composed components and their coordination. The other relies on enforcement of specific properties. A key problem in the application of this approach is composability: how to obtain a system meeting a given global property by composing architectures meeting specific properties? Existing results limit both approaches to particular classes of properties, e.g. deadlock-freedom and state invariants. We believe that a significant research effort is needed to overcome these limitations.

The third problem is using logics to characterise architectures and their properties. We show that two types of logics are needed for this purpose. Interaction logics characterise the possible interactions of a system, that is of a particular architecture. These logics have been studied to a large extent and applied in the BIP framework. In contrast, configuration logics can be used to characterise families of architectures, e.g. architecture styles. They are languages used for a feature-oriented analysis of architectures, such as OCL [27]. The relationships between configuration logic and other approaches for the description of architectures styles [1,29,30,32] need to be investigated.

The paper clearly distinguishes between architecture models and two types of logic-based specification formalisms. It also establishes links between the two types of description through satisfaction relations. Table 2 depicts the main characteristics of each formalism and significant differences.

Table 2. Architectures and Architectural Properties

Formalism features	**Architecture Modeling Connectors (Imperative)**	**Architecture Modeling Interaction Logics (Declarative)**	**Architecture Styles Specification - Configuration Logics**
Fixed set of components and connectors	Static connectors $I(P)$ $[g(X_P){:}X_P := f(X_P)]$	Propositional interaction logic, e.g. causality rules	Propositional configuration logic, e.g. connectivity primitives $\approx a$ and $\sim a$
Typed components; variables over components	Generic connectors	First-order interaction logic, e.g. Dy-BIP	First-order configuration logic
Variables over sets of components	Dynamic connectors	Second-order interaction logic	Second-order configuration logic

Interestingly, static models correspond to propositional logics, while dynamic models to higher order logics. Both dynamic models and higher order logics share the same basic concepts, e.g. they are defined on a set of typed components by using variables ranging over components and sets of components. Notice that component variables are needed to describe generic models and properties, while variables over sets of components are needed to describe dynamic creation/deletion and dynamic configurations. These similarities should allow a tight comparison of the three proposed formalisms, that needs to be further investigated.

References

1. Allen, R.B., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: Astesiano, E. (ed.) ETAPS 1998 and FASE 1998. LNCS, vol. 1382, pp. 21–37. Springer, Heidelberg (1998)
2. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
3. Aştefănoaei, L., Ben Rayana, S., Bensalem, S., Bozga, M., Combaz, J.: Compositional invariant generation for timed systems. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 263–278. Springer, Heidelberg (2014)
4. Attie, P., Baranov, E., Bliudze, S., Jaber, M., Sifakis, J.: A general framework for architecture composability. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 128–143. Springer, Heidelberg (2014)
5. Baranov, E., Bliudze, S.: Offer semantics: Achieving compositionality, flattening and full expressiveness for the glue operators in BIP. Technical Report EPFL-REPORT-203507, EPFL IC IIF RiSD (November 2014), `http://infoscience.epfl.ch/record/203507`.
6. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM), pp. 3–12. IEEE Computer Society (2006)
7. Bensalem, S., Bozga, M., Boyer, B., Legay, A.: Incremental generation of linear invariants for component-based systems. In: 13th International Conference on Application of Concurrency to System Design (ACSD), pp. 80–89. IEEE (2013)
8. Bensalem, S., Bozga, M., Legay, A., Nguyen, T.-H., Sifakis, J., Yan, R.: Incremental component-based construction and verification using invariants. In: Bloem, R., Sharygina, N. (eds.) 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 257–256. IEEE (2010)
9. Bensalem, S., Bozga, M., Nguyen, T.-H., Sifakis, J.: D-finder: A tool for compositional deadlock detection and verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 614–619. Springer, Heidelberg (2009)
10. Bensalem, S., Bozga, M., Nguyen, T.-H., Sifakis, J.: Compositional verification for component-based systems and application. IET Software 4(3), 181–193 (2010)
11. Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.-H.: Compositional verification for component-based systems and application. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 64–79. Springer, Heidelberg (2008)

12. Bensalem, S., Griesmayer, A., Legay, A., Nguyen, T.-H., Peled, D.: Efficient deadlock detection for concurrent systems. In: Singh, S., Jobstmann, B., Kishinevsky, M., Brandt, J. (eds.) 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 119–129. IEEE (2011)
13. Bliudze, S.: Towards a theory of glue. In: Carbone, M., Lanese, I., Silva, A., Sokolova, A. (eds.) 5th International Conference on Interaction and Concurrency Experience (ICE). EPTCS, vol. 104, pp. 48–66 (2012)
14. Bliudze, S., Sifakis, J.: The algebra of connectors — Structuring interaction in BIP. In: 7th ACM & IEEE International Conference on Embedded Software (EMSOFT), pp. 11–20. ACM SigBED (2007)
15. Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 508–522. Springer, Heidelberg (2008)
16. Bliudze, S., Sifakis, J.: Causal semantics for the algebra of connectors. Formal Methods in System Design 36(2), 167–194 (2010)
17. Bliudze, S., Sifakis, J., Bozga, M., Jaber, M.: Architecture internalisation in BIP. In: Proceedings of The 17th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE), pp. 169–178. ACM (July 2014)
18. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: From high-level component-based models to distributed implementations. In: 10th ACM International Conference on Embedded Software (EMSOFT), pp. 209–218. ACM, New York (2010)
19. Bozga, M., Jaber, M., Maris, N., Sifakis, J.: Modeling dynamic architectures using dy-BIP. In: Gschwind, T., De Paoli, F., Gruhn, V., Book, M. (eds.) SC 2012. LNCS, vol. 7306, pp. 1–16. Springer, Heidelberg (2012)
20. Bozga, M., Jaber, M., Sifakis, J.: Source-to-source architecture transformation for performance optimization in BIP. In: IEEE International Symposium on Industrial Embedded Systems (SIES), pp. 152–160 (July 2009)
21. Bruni, R., Melgratti, H., Montanari, U.: Behaviour, interaction and dynamics. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. LNCS, vol. 8373, pp. 382–401. Springer, Heidelberg (2014)
22. Dhungana, D., Rabiser, R., Grünbacher, P., Prähofer, H., Federspiel, C., Lehner, K.: Architectural knowledge in product line engineering: An industrial case study. In: 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA), pp. 186–197. IEEE (2006)
23. D'Souza, D., Gopinathan, M.: Conflict-tolerant features. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 227–239. Springer, Heidelberg (2008)
24. Eker, J., Janneck, J., Lee, E., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—The Ptolemy approach. Proceedings of the IEEE 91(1), 127–144 (2003)
25. Fares, E., Bodeveix, J.-P., Filali, M.: Event algebra for transition systems composition - application to timed automata. In: Sánchez, C., Venable, K.B., Zimányi, E. (eds.) 20th International Symposium on Temporal Representation and Reasoning (TIME), pp. 125–132 (September 2013)
26. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science. Prentice Hall (April 1985)
27. ISO/IEC. Information technology – Object Management Group – Object Constraint Language (OCL). Technical Report ISO/IEC 19507, ISO, Object Management Group (2012)
28. ISO/IEC/IEEE. Systems and software engineering – Architecture description. Technical Report ISO/IEC/IEEE 42010, ISO (2011)

29. Koehler, C., Lazovik, A., Arbab, F.: Connector rewriting with high-level replacement systems. Electr. Notes Theor. Comput. Sci. 194(4), 77–92 (2008)
30. Kumar, A.: Software architecture styles a survey. International Journal of Computer Applications 87(9) (2014)
31. Lustig, Y., Vardi, M.: Synthesis from component libraries. International Journal on Software Tools for Technology Transfer 15(5-6), 603–618 (2013)
32. Metayer, D.L.: Describing software architecture styles using graph grammars. IEEE Trans. Software Eng. 24(7), 521–533 (1998)
33. Milner, R.: Calculi for synchrony and asynchrony. Theoretical Computer Science 25(3), 267–310 (1983)
34. Milner, R.: Communication and Concurrency. Prentice Hall International Series in Computer Science. Prentice-Hall (1989)
35. Milner, R.: Communicating and Mobile Systems: The π-calculus. Cambridge University Press (1999)
36. Papadopoulos, G.A., Arbab, F.: Configuration and dynamic reconfiguration of components using the coordination paradigm. Future Generation Computer Systems 17, 1023–1038 (2001)
37. Plath, M., Ryan, M.: Feature integration using a feature construct. Science of Computer Programming 41(1), 53–84 (2001)
38. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)