Contributions to the Rigorous Design of Concurrent Component-Based Software and Systems Using BIP

Simon Bliudze

Habilitation à Diriger des Recherches
Université de Lille

École Graduée MADIS-631
Spécialité : Informatique

Simon Bliudze

Université de Lille, CRIStAL &
Centre Inria de l'Université de Lille
Équipe-Projet SPIRALS

Contributions to the Rigorous Design of Concurrent Component-Based Software and Systems Using BIP

Soutenue le 08 mars 2024 devant le jury composé de

Rapporteurs:
Susanne GRAF   -   DR CNRS au Verimag
Christian PEREZ   -   DR Inria au Centre Inria de Lyon
Gianluigi ZAVATTARO   -   Professeur à University of Bologna, Italie

Examinateurs:
Panagiotis KATSAROS   -   Professeur à Aristotle University of Thessaloniki, Grèce
Olga KOUCHNARENKO   -   Professeure à l'Université de Franche-Comté

Garante:
Laurence DUCHIEN   -   Professeure à l'Université de Lille

Abstract

The work presented in the manuscript was carried out in the framework of the Rigorous System Design (RSD) approach. This approach emphasizes multiple levels of separation of concerns. It enforces behavioural properties through semantics-preserving transformations. The RSD flow involves designing an application model, verifying elementary properties, extending the model with platform components, and generating executable code. The approach is applied to various domains, including embedded systems, cyber-physical systems, autonomous systems, and digital twins. The three main chapters summarise the key contributions of the author.

Chapter 2 explores the classical semantics of BIP, a robust component framework central to the RSD methodology. It introduces the theory of architectures, a key element enabling the combination of predefined BIP design patterns to enforce desired behavioural properties. The design process, starting from system requirements and leading to the C++ implementation, is illustrated using the CubETH nano-satellite on-board software case study.

Chapter 3 develops a formal algebraic framework for comparing the expressive power of component-based frameworks. It applies this framework to analyse the expressiveness of the BIP framework. Additionally, an alternative "offer" semantics of BIP is introduced, exploring its relationship with the classical semantics.

Chapter 4 addresses the adaptation of the RSD approach to general-purpose software development. It introduces JavaBIP, a Java implementation of BIP-inspired coordination mechanisms. JavaBIP utilizes Java annotations and reflection mechanisms to define BIP models associated with Java objects. The chapter details the coordination mechanisms, modular architecture, and the dynamic addition/removal of components in JavaBIP. Two ongoing applications in Cloud Computing and Software Variability, leveraging JavaBIP, are highlighted.

Future work directions are outlined, including the need for a unifying modelling framework for self-adaptive systems, exploration of architecture styles, development of model extraction mechanisms to address software evolution, ensuring model-software adequation through runtime monitoring and dynamic approaches, and addressing the challenge of distributed implementation for scalability while maintaining expressiveness in coordination mechanisms.

Résumé

Le travail présenté dans le manuscrit a été réalisé dans le cadre de l'approche de Conception Rigoureuse des Systèmes (RSD). Cette approche met l'accent sur plusieurs niveaux de séparation des préoccupations et impose des propriétés comportementales grâce à des transformations préservant la sémantique. Le flux RSD implique la conception d'un modèle d'application, la vérification des propriétés élémentaires, l'extension du modèle avec des composants de plateforme, et la génération de code exécutable. L'approche est appliquée à divers domaines, notamment les systèmes embarqués, les systèmes cyber-physiques, les systèmes autonomes et les jumeaux numériques. Les trois principaux chapitres résument les principales contributions de l'auteur.

Le chapitre 2 explore la sémantique classique de BIP, un cadre de composants robuste au cœur de la méthodologie RSD. Il introduit la théorie des architectures, un élément clé permettant la combinaison de motifs de conception BIP prédéfinis pour imposer des propriétés comportementales souhaitées. Le processus de conception, partant des exigences du système et aboutissant à l'implémentation C++, est illustré à l'aide de l'étude de cas du logiciel embarqué du nano-satellite CubETH.

Le chapitre 3 développe un cadre algébrique formel pour comparer la puissance expressive des cadres basés sur les composants. Il applique ce cadre pour analyser l'expressivité du cadre BIP. De plus, une sémantique alternative "offre" de BIP est introduite, explorant sa relation avec la sémantique classique.

Le chapitre 4 aborde l'adaptation de l'approche RSD au génie logiciel généraliste. Il introduit JavaBIP, une implémentation Java de mécanismes de coordination inspirés de BIP. JavaBIP utilise des annotations Java et des mécanismes de réflexion pour définir des modèles BIP associés à des objets Java. Le chapitre détaille les mécanismes de coordination, l'architecture modulaire, et l'ajout/suppression dynamique de composants dans JavaBIP. Deux applications en cours de développement dans les domaines du Cloud Computing et de la Variabilité logicielle, exploitant JavaBIP, sont mises en avant.

Les orientations futures sont évoquées, notamment la nécessité d'un cadre de modélisation unificateur pour les systèmes auto-adaptatifs, l'exploration des styles d'architecture, le développement de mécanismes d'extraction de modèles pour traiter l'évolution des logiciels, l'assurance de l'adéquation modèle-logiciel par le biais de la surveillance en temps réel et des approches d'analyse dynamique, ainsi que le défi d'une implémentation distribuée permettant le passage à l'échelle tout en maintenant l'expressivité des mécanismes de coordination.

Acknowledgments

I am very grateful to the members of my HDR jury for accepting to serve on the committee and for the stimulating questions and comments during the defence. Particularly, I would like to thank Susanne Graf, Christian Perez and Gianluigi Zavattaro for their time and patience when reviewing the thesis.

Many thanks are due to the examiners and colleagues, Panagiotis Katsaros and Olga Kouchnarenko. A significant part of Chapter 2 presents the results obtained within the ESA-funded project "Catalogue of System and Software Properties" prepared and lead by Panagiotis, which set up the framework for numerous opportunities for fruitful interactions and, in-between, the pleasant time spent, particularly, in Thessaloniki. Even though the results of our recent collaboration with Olga did not make it into the manuscript, I would like to thank her for her comments on the draft versions of parts of the manuscript. Separate thanks are due to Olga for presiding over the defence.

I am deeply grateful to my advisor, Laurence Duchien, for her gentle but resolute guidance. Having the patience to wait out while I was taking my time to work on the manuscript, Laurence offered that invaluable kind of advice that always comes at the right moment and is so much on the spot *once received* that one is almost ashamed of having had to ask.

Many thanks to Lionel Seinturier, the entire SPIRALS team and the colleagues from the Inria Centre at the University of Lille and from CRIStAL for having created and maintained the friendly working environment that was indispensable for the work on the manuscript.

A separate thank you to Hélène Touzet for her support and the final nudge to push me over the crest when my motivation was flailing.

No thanks would be enough to express my gratitude to my long-time mentor, Joseph Sifakis. Most of the work presented in Chapter 2 has been carried out under his guidance. The expressiveness study (Chapter 3) and the JavaBIP project (Chapter 4) were heavily inspired by his thinking.

Of course, this work would not have been possible without the precious collaborations with my co-authors and students. Mentioning all of them here would be impossible but I would particularly like to thank Panagiotis Katsaros, Paul Attie, Mohammad Jaber and my former PhD students Anastasia Mavridou, Eduard Baranov, Alina Zolotukhina, Trình Lê Khánh and Salman Farhat.

Last but not least, I would not be writing these lines, neither the manuscript, nor the papers that underly it, if it were not for the love and unflailing support from Katia, Tom and my parents to whom I am deeply and forever gratefully in debt.

Contents

List of Figures

x

List of Tables

Introduction

Modern software systems are inherently concurrent. They consist of components running simultaneously and sharing access to resources provided by the execution platform. For instance, embedded control software in various domains, ranging from household robotics through operation of smart power-grids to on-board satellite software, commonly comprises, in addition to components responsible for taking the control decisions, a set of components driving the operation of sensing and actuation devices. These components interact through buses, shared memories and message buffers, leading to resource contention and potential deadlocks compromising mission- and safety-critical operations. Similar problems are observed in various kinds of software, including system, work-flow management, integration software, web services etc. Essentially, any software entity that goes beyond simply computing a certain function, necessarily has to interact and share resources with other such entities.

The intrinsic concurrent nature of such interactions is the root cause of the sheer complexity of the resulting software. Indeed, in order to analyse the behaviour of such a software system, one has to consider all possible interleavings of the operations executed by its components. Thus, the complexity of software systems is exponential in the number of their components, making a posteriori verification of their correctness practically infeasible. An alternative approach consists in ensuring correctness by construction, through the application of well-defined design principles [BCK12; Gam+94], imposing behavioural contracts on individual components [Ben+12] or by applying automatic transformations to obtain executable code from formally defined high-level models [Sif12].

1.1 The Rigorous System Design approach

The Rigorous System Design (RSD) [Sif12] approach enforces multiple levels of separation of concerns. It relies on a sequence of semantics-preserving transformations to obtain an implementation of the system from a high-level model, while preserving all the properties established along the way.

Figure 1.1 illustrates a simplified instantiation of the RSD flow. One starts by designing the *application model*. The application model is verified to prove the elementary properties that are not assured by construction, such as absence of local deadlock, and satisfaction of basic requirements. These elementary properties, serve as a basis for the proof of global properties, obtained by construction.

The application model is then extended with additional components modelling the target platform to obtain the *system model*, which is used to perform platform specific analyses and the optimisation of performance through the exploration of the design space (memories, buses, mapping of software components to hardware elements etc.).

Finally, the model is enriched with platform specific information (e.g. communication primitives) and, after removing components modelling hardware elements, executable code is automatically generated.

Proving that the assumptions made at the modelling level to justify the separation of concerns hold, indeed, at the platform level, guarantees that all the properties established throughout the design process also hold for the generated code.
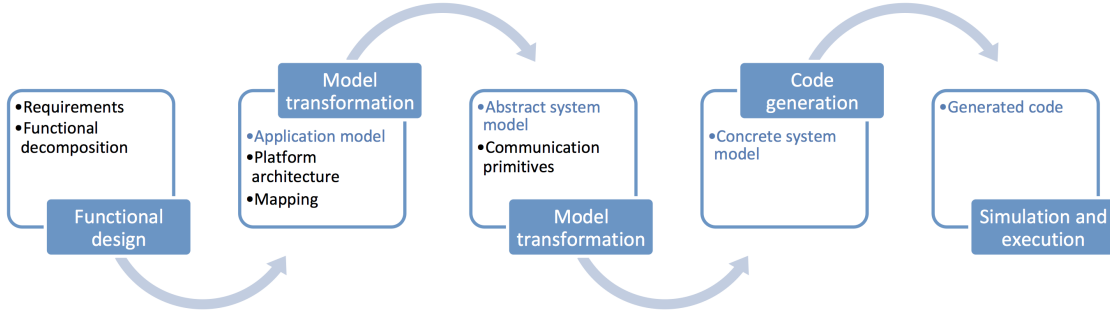
Figure 1.1: A simplified example of the RSD flow instantiation (the blue items are the result of the previous stage; the black ones are provided as new input at the current stage)

Thus, the RSD approach applies—on the higher abstraction level of the system design process—the same principles as those provided by standard compilers for the generation of machine-executable code from programs written in languages such as Java or C++. It consists in decomposing the argument justifying the correctness of the entire process into several independent arguments justifying the correctness of individual transformations. Furthermore, it provides flexibility w.r.t. the target platforms by postponing design choices as far as possible and allowing for different transformations of the same model to be applied at every design stage. However, in drawing this parallel, it is appropriate to differentiate between commonly used compilers such as `gcc`, where the public trust originates mainly from the extensive usage experience, and verified compilers such as `CompCert` [Ler+16], where preservation of semantics at the various stages is formally verified. Although the second scenario is currently preferable for RSD tool-sets due to lack of usage history comparable to that of compilers such as `gcc`, both can be relevant in practice.

Applications Although the RSD approach as formulated by Sifakis [Sif12] originates from the development of the BIP framework for the Embedded Software design (see, for example, [BBS06; Bas+08]), it is applicable in a much broader variety of domains, whereof I will mention below just a few.

In [Bli+19], we have discussed the key issues of applying the RSD approach to the Cyber-Physical Systems (CPSs), which comprise components with both discrete and continuous underlying dynamics. In this context, we have argued that the "objective of cyber-physical system modeling is two-fold. Firstly, simulation of such models provides means for validating the system design. [...] The second objective of cyber-physical systems design is to provide the means for the generation of executable code for the discrete control sub-system." The key point here is that the two design artefacts, i.e. the simulator and the code of the control sub-system, are obtained through *two branches of the design flow sharing a substantial prefix*. Thus, the generated control sub-system is equivalent[1] to the corresponding components of the simulator *by construction*.

Expanding on the above idea, it is clear that the RSD approach can be of great benefit for system design and operation involving the so-called Digital Twins where simulation becomes "a core functionality of systems by means of seamless assistance along the entire life cycle" [Ros+15].

Autonomous and (self-)adaptive systems constitute another significant domain for the application of the RSD approach. These include, for example, autonomous vehicles, Cloud and IoT applications. Indeed, such systems must react to changing environmental constraints and user requirements. Therefore they are characterised by high dynamicity both of their behaviour and their structure. In particular, this implies that many of the underlying verification problems are

---

[1]The notion of equivalence, here, implicitly refers to the same semantic equivalence of the underlying modelling framework that is preserved by the constituent model transformations of the RSD flow.

undecidable [EN95; Blo+15] emphasizing the need for by-construction correctness provided by the RSD approach.

The main theme of my current research project aims for adapting the RSD approach to the broad spectrum of general-purpose software development. This target application domain imposes constraints that are fundamentally different from those of the embedded systems design, which had motivated the design choices behind the original BIP framework.

1.2 Structure of the manuscript

This manuscript is structured into three chapters:

1. BIP-based Rigorous System Design flow

2. A formal study of expressiveness

3. The presentation of JavaBIP—a Java flavour of BIP

The first chapter presents and illustrates with the CubETH nanosatellite on-board software case study the RSD approach based on the classical BIP. In the second chapter, I present an algebraic framework for the comparison of expressiveness of component-based systems and its use for a study of the expressiveness of BIP. This informs the choice of the coordination mechanisms for subsequent work. The third chapter presents the design choices and the implementation of JavaBIP.

The three chapters are mostly independent and can be read individually or out of order. The only notable exception is that the fundamental principles of BIP are presented only in Chapter 2 (Section 2.1).

BIP-based Rigorous System Design flow

Based on the observation that a posteriori verification is not feasible for most realistic concurrent systems, the RSD approach aims at enforcing desired behavioural properties by construction. This is achieved, essentially, by applying a sequence of semantics-preserving model transformations progressively realising design choices appropriate at each level. This has an additional benefit of postponing design choices as much as possible and, thereby, ensures that optimal solutions are not discarded arbitrarily.

Thus, the RSD approach relies on two fundamental elements:

- a unifying component-based framework with formally defined operation semantics is necessary to define and reason about semantics-preserving model transformations, and

- a method for designing correct high-level models based on informal requirements to initiate the process

In this chapter, I present 1) the classical semantics of BIP—the component framework underlying the approach, 2) the theory of architectures—the key ingredient of the design approach, which allows combining predefined BIP design patterns to enforce desired behavioural properties on the resulting system, and 3) the resulting design process going from a list of system requirements to the C++ implementation.

A formal study of expressiveness

In order to understand the applicability limits of the proposed design approach, one has to study the expressive power of the underlying component-based framework. However, for such a study to be possible a proper comparison framework has to be developed. Indeed, most expressiveness

studies focus on two questions: 1) what can be computed? and 2) how concise is the program? The first question is typically answered by a comparison to the computing power of Turing machines. The answer to the second question can be summarised—admittedly in a somewhat simplistic manner—by saying that parts of a language represent syntactic sugar w.r.t. another language. None of these two approaches captures the essence of component-based design, where given composition operators are applied to a set of components to build a system *without changing the components themselves.*

In this chapter, I present 1) a formal algebraic framework that allows the comparison of the expressive power of component-based frameworks, 2) its application to a study of the expressiveness of the BIP framework, and 3) an alternative, "offer" semantics of BIP and its relation with the classical one.

JavaBIP

This chapter presents JavaBIP—a Java implementation of BIP-inspired coordination mechanisms aimed at general purpose software engineering rather than the design of embedded systems.

The main challenge comes from the fact that, in the context of general purpose software engineering, one cannot expect developers to take a disciplined essentially top-down approach relying exclusively on high-level models and semantics-preserving transformations. The key reasons are

1) the complexity of the software stack and, in particular, the broad use of existing libraries and frameworks, and 2) rapid code evolution due, for example, to the application of agile development methodologies.

We have addressed this challenge by moving away from the code generation paradigm, relying instead on Java annotations and reflection mechanisms to define BIP models associated to Java objects.

In this chapter I present 1) the coordination mechanisms adopted in JavaBIP, 2) the modular architecture used for the JavaBIP implementation, 3) the mechanism allowing to add and remove components from the system dynamically, and 4) two applications—currently under development—in the domains of Cloud Computing and Software Variability, respectively.

I or We?

Most of the material presented in this manuscript results from collaborations that I had with many people. I have, therefore decided to use "we" when presenting that material and "I" in the parts specific to this manuscript.

BIP-based Rigorous System Design flow

The RSD approach relies on two fundamental elements. Firstly, a *unifying component-based framework with formally defined operational semantics* is necessary to define and reason about semantics-preserving model transformations. On the one hand, such a unifying framework must be expressive enough to allow modelling of a broad spectrum of systems. On the other hand, it must be simple enough to facilitate the formulation and verification of proofs. Secondly, methods and tools for the design of correct-by-construction *high-level models* are necessary to initiate the process. Although it is difficult to imagine a unique approach that would fit all the various application domains, it seems reasonable to expect that these approaches will share a common core, comprising, at the very least, some form of 1) requirement elicitation and formalisation and 2) (semi-)automatic synthesis of parts of the models in order to discharge these requirements.

In this chapter, I present

- an overview of the Behaviour-Interaction-Priority (BIP) component-based modelling framework that underlies the rest of the work discussed in this manuscript,

- an overview of the theory of architectures—the key ingredient of the design approach, which allows combining predefined BIP design patterns to enforce desired behavioural properties on the resulting system

- an ontology-based design process, developed in the *Catalogue of System and Software Properties* (CSSP) project, going from a list of system requirements to the C++ implementation.

2.1 The BIP component framework

In this section, I will provide brief overviews, first, of the classical operational semantics of BIP as it was initially published in [BS07], then, of connectors that are used to assemble systems from components [BS07; BS08b; BS10].

2.1.1 The basic model

We consider an *algebra of components*,[1] i.e. an algebraic structure

$$\mathbf{A} ::= C \mid o\langle C_1, \ldots, C_n \rangle, \qquad C \in \mathbf{C}, \ C_1, \ldots, C_n \in \mathbf{A} \text{ and } o \in \mathbf{G},$$

generated by a set of composition (glue) operators $\mathbf{G}$ from a set of atomic components $\mathbf{C}$. Semantics of components is defined by a partial mapping $\sigma : \mathbf{A} \rightharpoonup \mathbf{B}$, where $\mathbf{B}$ is a behaviour type.[2]

---

[1]A more detailed formalisation will be provided in Chapter 3.

[2]The mapping is partial to account for the possibility that some syntactically correct terms might not be correct semantically.

Component model

The behaviour type of BIP is the set of *Labelled Transition Systems* (LTS).

Definition 2.1.1. A *labelled transition system* (LTS) is a quadruple $(Q, P, \to, q^0)$, where Q is a set of *states*, P is a set of *ports*, $q^0 \in Q$ is the initial state and $\to\, \subseteq Q \times 2^P \times Q$ is a set of *transitions* labelled by sets of ports, such that only self-loops can be labelled by the empty set of ports, i.e. $(q, \varnothing, q') \in\, \to$ implies $q = q'$. For $q, q' \in Q$ and $a \in 2^P$, we write $q \xrightarrow{a} q'$ for $(q, a, q') \in\, \to$. A label $a \in 2^P$ is *active* in a state $q \in Q$ (denoted $q \xrightarrow{a}$), if there exists $q' \in Q$ such that $q \xrightarrow{a} q'$. We abbreviate $q \xrightarrow{a}{\not{}} \stackrel{def}{=} \neg(q \xrightarrow{a})$.

By abuse of notation, we define the binary relation $\to\, \subseteq Q \times Q$ as $q \to q' \stackrel{def}{\Leftrightarrow} \exists a \subseteq P : q \xrightarrow{a} q'$ and denote by $\xrightarrow{*}$ its reflexive transitive closure. A state $q \in Q$ is *reachable* in the LTS if $q^0 \xrightarrow{*} q$.

Intuitively, transitions labelled by $\varnothing$ represent idling: a component that remains idle should not change state, hence the restriction to self-loops. Notice that we distinguish idling from unobservable internal transitions, which we do not model explicitly. To model unobservable transitions, one can use a reserved label, e.g. τ or ε, and restrict the ways it can be synchronised with other transitions. This is the approach traditionally taken in the literature [Mil89; Hoa85].

Atomic BIP components are those defined directly as LTSs: $\mathbf{C} \stackrel{def}{=} \{(P, B) \mid B = (Q, P, \to, q^0)\}$. The semantics of atomic components is given by their behaviour: $\sigma(P, B) \stackrel{def}{=} B$. The set of ports P is the *interface* of a component (P, B).

Note 2.1.2. When speaking of a set of LTSs $B_i = (Q_i, P_i, \to_i, q_i^0)$, for $i \in [1, n]$, it is common to assume that all Q_i and P_i are pairwise disjoint, i.e. $i \neq j$ implies $Q_i \cap Q_j = P_i \cap P_j = \varnothing$. When the indices are clear from the context, we drop them on transition relations and simply write $\to$ as, for example in $q_i \xrightarrow{a} q_i'$.

> Furthermore, for given pair-wise disjoint sets of ports P_i, with $i \in [1, n]$, we will always denote $P \stackrel{def}{=} \bigcup_{i=1}^n P_i$.

Glue operators

BIP glues consist of two layers. *Interaction models* define the sets of allowed *interactions*, i.e. synchronisations between the transitions of their operand components. *Priority models* define the conflict resolution policies, reducing non-determinism when several synchronisations allowed by the interaction model are enabled simultaneously.

Interaction models For a set of ports P, an *interaction model* is a set of interactions $\gamma \subseteq 2^P$. A compound component $\gamma\langle C_1, \ldots, C_n \rangle$ is obtained by applying an interaction model γ to a set of (not necessarily atomic) components $C_1, \ldots, C_n$.

Intuitively, an interaction a allowed by the interaction model γ can be fired when all the components involved in a are ready to fire the corresponding transitions. All the components that are not involved in a remain in their current states.

Assuming $\sigma(C_i) = (Q_i, P_i, \to_i, q_i^0)$, for $i \in [1, n]$, the semantics of the application of an interaction model γ to components $C_1, \ldots, C_n$ is defined by putting $\sigma(\gamma\langle C_1, \ldots, C_n \rangle) \stackrel{def}{=} (Q, P, \to_\gamma, q^0)$, with $Q = \prod_{i=1}^n Q_i$, $q^0 = (q_1^0, \ldots, q_n^0)$ and the smallest transition relation $\to_\gamma$ satisfying the rule

$$\frac{a \in \gamma \quad \left\{ q_i \xrightarrow{a \cap P_i} q_i' \,\middle|\, i \in I \right\} \quad \left\{ q_i = q_i' \,\middle|\, i \notin I \right\}}{q_1 \ldots q_n \xrightarrow{a}_\gamma q_1' \ldots q_n'}, \tag{2.1}$$

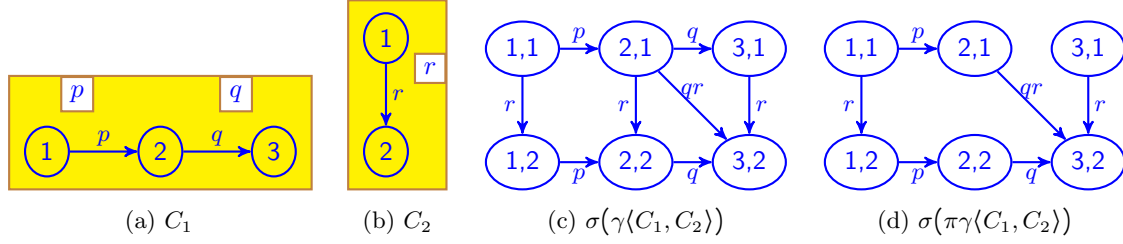where $I = \{ i \in [1, n] \mid a \cap P_i \neq \varnothing \}$.

Figure 2.1: Components and behaviour for Examples 2.1.5 and 2.1.6 ($\gamma = \{p, q, r, qr\}$, $\pi = \{q \prec qr, r \prec qr\}$)

Note 2.1.3 (How to read the rule above?)**.** Here and below, we use Structural Operational Semantics (SOS) rules [Plo81] to define the semantics of composition operators.

Each SOS rule consists of a set of whitespace-separated *premises* written above the line and one *conclusion* written below the line. For the sake of clarity, we group some premises in sets. For instance, in the rule (2.1), the second premise is, in fact, itself a set of premises $q_i \xrightarrow{a \cap P_i} q_i'$, one for each $i \in I$.

In our context, SOS rules should be read as follows. If *all* premises of the rule are satisfied for a given valuation of the free variables—variables a, $q_1, \dots, q_n$ and $q_1', \dots, q_n'$ for (2.1)—then the conclusion must be satisfied as well. Thus, the rule (2.1) can be rewritten as the first order formula

$$\forall a \subseteq 2^P, \forall q_1, q_1' \in Q_1, \dots, \forall q_n, q_n' \in Q_n,$$
$$\left(a \in \gamma \land \exists I = \{i \in [1, n] \mid a \cap P_i \neq \varnothing\} : \left(\bigwedge_{i \in I} q_i \xrightarrow{a \cap P_i} q_i' \right) \land \left(\bigwedge_{i \notin I} q_i = q_i' \right) \right)$$
$$\implies q_1 \dots q_n \xrightarrow{a}_\gamma q_1' \dots q_n'.$$

Notice that we refer to the *smallest* transition relation that satisfies this rule. This means that the relation $\rightarrow_\gamma$ contains all the transitions that are necessary to satisfy the rule, *and only those transitions.*

Note 2.1.4. When the interaction model allows idling, i.e. $\varnothing \in \gamma$, the compound component has a self-loop labelled by $\varnothing$ in every state. The fact that components can have idling self-loops does not introduce any ambiguity in the interpretation of (2.1), since, by Definition 2.1.1, $q \xrightarrow{\varnothing} q'$ implies $q = q'$.

Example 2.1.5. Consider the two components C_1 and C_2 shown in Figures 2.1a and 2.1b, respectively.[3] We have $P_1 = \{p, q\}$ and $P_2 = \{r\}$. Consider the interaction model $\gamma = \{p, q, r, qr\}$.[4] The semantics of the glue operator defined by the interaction model γ is given by the following four rules, obtained by instantiating the rule (2.1), removing premises whereof satisfaction does not depend on the state of the operand behaviours—e.g. when the rule is instantiated with an interaction $a \in \gamma$, the corresponding premise is satisfied in all states—and replacing the primed state variables in the conclusions with the unprimed ones whenever the premise of the form $q_i = q_i'$ should have been used:

$$\frac{q_1 \xrightarrow{p} q_1'}{q_1 q_2 \xrightarrow{p} q_1' q_2}, \quad \frac{q_1 \xrightarrow{q} q_1'}{q_1 q_2 \xrightarrow{q} q_1' q_2}, \quad \frac{q_2 \xrightarrow{r} q_2'}{q_1 q_2 \xrightarrow{r} q_1 q_2'}, \quad \frac{q_1 \xrightarrow{q} q_1' \quad q_2 \xrightarrow{r} q_2'}{q_1 q_2 \xrightarrow{qr} q_1' q_2'}. \tag{2.2}$$

---

[3]We omit the initial states, which are irrelevant for this example.

[4]For the sake of clarity, we use the juxtaposition of ports $\gamma = \{p, q, r, qr\}$ instead of the set notation $\gamma = \{\{p\}, \{q\}, \{r\}, \{q, r\}\}$ for interactions.

The behaviour of the compound component $\gamma\langle C_1, C_2\rangle$ is shown in Figure 2.1c. $\diamond$

Priority models   A *priority model* on an interaction model γ is a strict[5] partial order $\pi \subseteq \gamma \times \gamma$ (we write $a \prec b$ as a shorthand for $(a, b) \in \pi$).

Intuitively, an interaction can be fired only if no higher-priority interaction is enabled.

The semantics of the application of a priority model π to a composite component $C = \gamma\langle C_1, \ldots, C_n\rangle$, such that $\sigma(C) = (Q, P, \to, q^0)$, is defined by putting $\sigma(\pi\langle C\rangle) \stackrel{def}{=} (Q, P, \to_\pi, q^0)$, with the smallest transition relation $\to_\pi$ satisfying the rule

$$\frac{q \stackrel{a}{\to} q' \quad \left\{ q \stackrel{b}{\not\to} \;\middle|\; a \prec b \right\}}{q \stackrel{a}{\to}_\pi q'}. \tag{2.3}$$

Example 2.1.6. Consider again the two components C_1 and C_2 shown in Figures 2.1a and 2.1b, respectively. In addition to the interaction model $\gamma = \{p, q, r, qr\}$ from Example 2.1.5, consider $\pi = \{q \prec qr, r \prec qr\}$.[6]   The semantics of the glue operator defined by the combination of the interaction model γ and the priority model π is given by the following four rules, obtained by composing the rules (2.2) with rules of the form (2.3), replacing $q_1 q_2 \stackrel{qr}{\not\to}$ by $q_1 \stackrel{q}{\not\to}$ or $q_2 \stackrel{r}{\not\to}$ as appropriate:

$$\frac{q_1 \stackrel{p}{\to} q_1'}{q_1 q_2 \stackrel{p}{\to} q_1' q_2}, \quad \frac{q_1 \stackrel{q}{\to} q_1' \quad q_2 \stackrel{r}{\not\to}}{q_1 q_2 \stackrel{q}{\to} q_1' q_2}, \quad \frac{q_1 \stackrel{q}{\not\to} \quad q_2 \stackrel{r}{\to} q_2'}{q_1 q_2 \stackrel{r}{\to} q_1 q_2'}, \quad \frac{q_1 \stackrel{q}{\to} q_1' \quad q_2 \stackrel{r}{\to} q_2'}{q_1 q_2 \stackrel{qr}{\to} q_1' q_2'} \tag{2.4}$$

(the differences with (2.2) are highlighted in red).

The behaviour of the compound component $\pi\gamma\langle C_1, C_2\rangle$ is shown in Figure 2.1d. Comparing Figures 2.1c and 2.1d it is easy to see that, among the transitions labelled by q, only the transition $(2,2) \stackrel{q}{\to} (3,2)$ is enabled and not $(2,1) \stackrel{q}{\to} (3,1)$. Indeed, the negative premise in the second rule of (2.4), generated by the priority $q \prec qr$, suppresses the interaction q when the interaction qr is enabled. The same holds for the transition $(2,1) \stackrel{r}{\to} (2,2)$ labelled by r. $\diamond$

The priority model in Example 2.1.6 is an instance of the so-called *maximal progress* priority defined by $\pi \stackrel{def}{=} \{a \prec b \mid a, b \in \gamma, \; a \subset b\}$. Maximal progress priorities are very common in practical examples and, in particular, are used by default in the standard Verimag implementations of BIP.

Definition 2.1.7. An *n*-ary *BIP glue operator* is a triple $\big((P_i)_{i=1}^n, \gamma, \pi\big)$, where $(P_i)_{i=1}^n$ are pairwise disjoint sets of ports and, denoting $P \stackrel{def}{=} \bigcup_{i=1}^n P_i$, the remaining two elements $\gamma \subseteq 2^P$ and $\pi \subseteq \gamma \times \gamma$ are, respectively, interaction and priority models on P.

We omit the sets of ports $(P_i)_{i=1}^n$ when they are clear from the context.

To simplify the notation, we denote the component obtained by applying the glue operator $\big((P_i)_{i=1}^n, \gamma, \pi\big)$ to sub-components $C_1, \ldots, C_n$, by $\pi\gamma\langle C_1, \ldots, C_n\rangle$ instead of $\big((P_i)_{i=1}^n, \gamma, \pi\big)\langle C_1, \ldots, C_n\rangle$. Furthermore, when $\pi = \varnothing$, we write $\gamma\langle C_1, \ldots, C_n\rangle$, omitting π.

Priorities do not introduce deadlock

Notice that only interactions belonging to the interaction model of a BIP glue operator can be used in the priority model. This gives the BIP glue operators an important property, which was originally shown in [GS03]: application of a priority model does not introduce deadlocks.

---

[5]As opposed to a (non-strict) partial order, which is a reflexive, antisymmetric and transitive relation, a *strict* partial order is an irreflexive and transitive (hence also antisymmetric) one.

[6]Here again, for the sake of clarity, we write $\pi = \{q \prec r\}$ instead of $\pi = \{(q, r)\}$

(a) Port attributes   (b) Rendezvous

$p \quad q \quad r$
$\{pqr\}$

(c) Broadcast

$p \quad q \quad r$
$\{p, pq, pr, pqr\}$

$p \quad q \quad r$
$\{p, q, pq, pr, qr, pqr\}$

Rendezvous
$\{pqr\}$

Atomic broadcast
$\{p, pqr\}$

Coordinated broadcast
$\{qr, pqr\}$

Causality chain
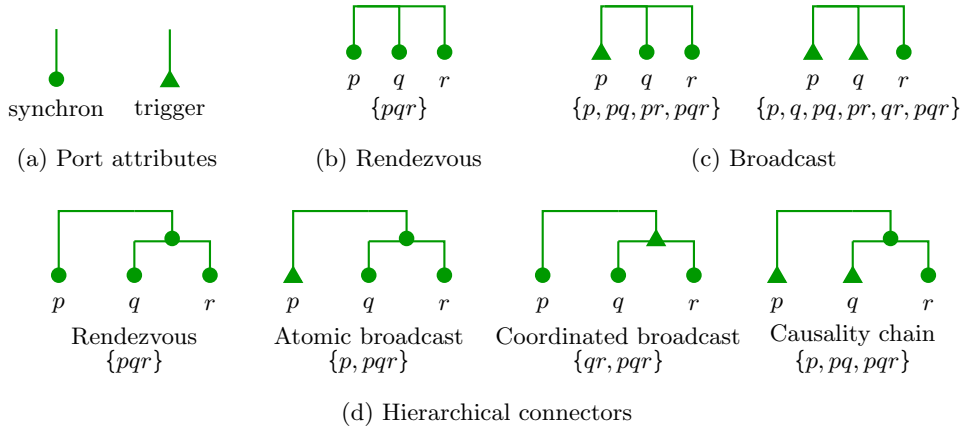$\{p, pq, pqr\}$

(d) Hierarchical connectors

Figure 2.2: BIP connectors (below each connector, we show the set of interactions it defines)

Definition 2.1.8. Let $B = (Q, P, \rightarrow, q^0)$ be a behaviour. A state $q \in Q$ is a *deadlock* if holds the statement $\forall a \subseteq P, q \overset{a}{\nrightarrow}$.

Lemma 2.1.9 ([GS03]). *Let C_i, such that $\sigma(C_i) = (Q_i, P_i, \rightarrow, q_i^0)$, for $i \in [1, n]$, be a set of components, γ and π be respectively interaction and priority models on $P = \bigcup_{i=1}^{n} P_i$. A state $q \in \prod_{i=1}^{n} Q_i$ is a deadlock in $\sigma(\pi\gamma\langle C_1, \ldots, C_n \rangle)$ if and only if it is a deadlock in $\sigma(\gamma\langle C_1, \ldots, C_n \rangle)$.*

Proof. The "if" implication is trivial. To prove the "only if" implication, assume that, for some $a \in \gamma$, we have $q \overset{a}{\rightarrow}_\gamma$. Let $b \subseteq P$ be an interaction, maximal w.r.t. π, such that $b \in \gamma$, $a \prec b$ and $q \overset{b}{\rightarrow}_\gamma$. If such b exists, holds $q \overset{b}{\rightarrow}_\pi$. Otherwise holds $q \overset{a}{\rightarrow}_\pi$. In both cases, q is not a deadlock in $\sigma(\pi\gamma\langle C_1, \ldots, C_n \rangle)$. □

Notice that this proof does not rely on π being a strict partial order. The lemma can be generalised to any *acyclic* relation $\pi \subseteq \gamma \times \gamma$.
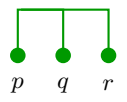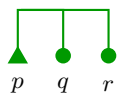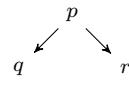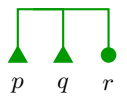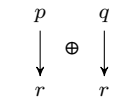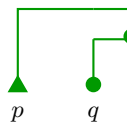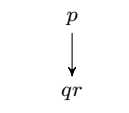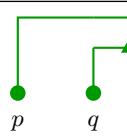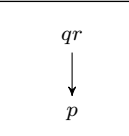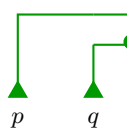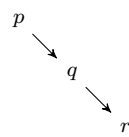
2.1.2   Connectors and algebras for structuring interaction

Specifying interaction models as sets of sets of ports is not practical due to their potentially exponential size. An algebra of connectors was introduced in [BS08b] in order to structure interactions in BIP models. *Connectors* are hierarchical, tree-like structures with component ports at the leaves (see Figure 2.2). Connectors define sets of interactions based on the synchronisation attributes of the connected ports, which may be either *synchron* or *trigger* (Figure 2.2a):

- if all connected ports are synchrons, then synchronisation is by *rendezvous*, i.e. the connector defines exactly one interaction, which comprises all its ports (Figure 2.2b);

- if the connector has at least one trigger, the synchronisation is by *broadcast*,[7] i.e. the connector defines the set of interactions comprised by all non-empty subsets of the connected ports containing at least one of the trigger ports (Figure 2.2c).

---

[7] Notice that our use of the terms "broadcast" and "atomic broadcast" (Figure 2.2d) is very different from the meaning commonly used in distributed computing where messages are broadcast to a number of recipients through a network. Here, we use the terms "rendezvous", "broadcast" and "atomic broadcast" for the sake of homogeneity with previous work on BIP (e.g. [BS07]) to denote different kinds of connectors. The use of the term "broadcast" in [BS07] was inspired by previous work on Statecharts [Har87].

Table 2.1: Algebraic representations of the connectors in Figure 2.2: graphical representation (triggers are shown as triangles, synchrons as bullets); interaction models (sets of interactions allowed by each connector); terms of the Algebra of Connectors (triggers are shown by ticks, square brackets indicate sub-connectors); Causal Interaction Trees terms (arrows indicate dependencies, $\oplus$ indicates independence); Systems of Causal Rules (a valuation of all port variables corresponds to an interaction, variables with value *true* are exactly those participating in it)

Connectors	Interaction Models	Algebra of Connectors	Causal Interaction Trees	Systems of Causal Rules
p q r	$\{pqr\}$	pqr	pqr	$p \Rightarrow qr$ $q \Rightarrow pr$ $r \Rightarrow pq$
p q r	$\{p, pq, pr, pqr\}$	$p'qr$	p ↙ q ↘ r	$p \Rightarrow true$ $q \Rightarrow p$ $r \Rightarrow p$
p q r	$\{p, q, pq, pr, qr,$ $pqr\}$	$p'q'r$	$p \to r \ \oplus\ q \to r$	$p \Rightarrow true$ $q \Rightarrow true$ $r \Rightarrow p \vee q$
p q r	$\{p, pqr\}$	$p'[qr]$	$p \to qr$	$p \Rightarrow true$ $q \Rightarrow pr$ $r \Rightarrow pq$
p q r	$\{qr, pqr\}$	$p[qr]'$	$qr \to p$	$p \Rightarrow qr$ $q \Rightarrow r$ $r \Rightarrow q$
p q r	$\{p, pq, pqr\}$	$p'[q'r]$	$p \to q \to r$	$p \Rightarrow true$ $q \Rightarrow p$ $r \Rightarrow q$

The same principle is recursively extended to hierarchical connectors, where one interaction from each subconnector is used to form an allowed interaction according to the synchron/trigger labeling of the connector nodes. For instance the causal chain connector in Figure 2.2d has the port p labeled as a trigger, whereas the binary broadcast subconnector $q{\blacktriangleright}\!\!-\!\!{\bullet}r$ is labeled as a synchron. Thus the causal chain connector allows the singleton interaction p and any interaction that combines p with some interaction of the subconnector $q{\blacktriangleright}\!\!-\!\!{\bullet}r$. Since the latter allows interactions q and qr, this results in three interactions allowed by the hierarchical connector: p, pq and pqr.

The interaction model is defined as the set of all interactions allowed by at least one of the connectors.

In previous work [BS07; BS10], we have defined and studied several algebraic structures used to represent BIP interaction models, in particular (cf. Table 2.1)

- the *Algebra of Connectors*—provides a textual notation for connectors equivalent to the graphical notation presented above;

- *Causal Interaction Trees*—exhibit the causal dependencies among sub-interactions of the interactions allowed by a connector;

- and *Systems of Causal Rules*—present the causal dependencies as Boolean implications.

Notice that some connectors are *equivalent*, denoted by the symbol '≃', meaning that they define the same sets of interactions. For example, the two rendezvous connectors shown in Figure 2.2 —using the Algebra of Connectors notation, $pqr \simeq p[qr]$—both allow one same interaction pqr. A slightly more complex example is provided by the two connectors $p[qr]' \simeq [pq']r$ (the former is the Coordinated broadcast connector from Figure 2.2) both defining the set of two interactions $\{qr, pqr\}$. This notion of equivalence is used for studying connector transformations [BS07; BS08b].[8]

Since interaction models are sets of sets of ports, one can consider their *characteristic predicates*, associating to an interaction model $\gamma \subseteq 2^P$ the corresponding predicate $\varphi_\gamma \in \mathbb{B}[P]$[9] on the port variables. The predicate φ_γ is *true* precisely on such valuations $v : P \to \mathbb{B}$ of the port variables that $\{p \in P \mid v(p) = true\} \in \gamma$. Characteristic predicates of interaction models turn out to be useful for reasoning about architecture composition as presented in Section 2.2 below. Furthermore, the Boolean encoding is also useful for representing interaction models as synchronisation constraints.

Characteristic predicates can be trivially defined in the Disjunctive Normal Form (DNF) as a disjunction of monomials, each representing an interaction. For instance, the characteristic predicate of the interaction model defined by one single connector $p'[q'r]$ (causality chain in Figure 2.2d) is $\varphi_{\{p'[q'r]\}} = p\,\overline{q}\,\overline{r} \vee p\,q\,\overline{r} \vee p\,q\,r$. However, it turns out to be more insightful to specify characteristic predicates as conjunctions of causal rules:

$$\varphi_{p'[q'r]} \equiv (p \vee q \vee r) \wedge (q \Rightarrow p) \wedge (r \Rightarrow q).$$

Here, the meaning of the implications $r \Rightarrow q$ and $q \Rightarrow p$ is that the participation of the port r in any interaction *requires* the participation of the port q, which, in turn, requires that of the port p. These two implications expose the causal dependencies visible in the connector structure and, even more so, in the corresponding causal interaction tree (see the last row in Table 2.1). The first conjunct simply states that at least one port must participate in any interaction.

In [BS10], we have shown that the set of interactions defined by any connector can be characterised by a Boolean formula as above, where the implications in each conjunct take the form

$$p \Rightarrow a_1 \vee \cdots \vee a_n,$$

with p being a port variable and each a_i being a conjunction of any number of port variables. We call p the *effect*, whereof $a_1, \ldots, a_n$ are the *causes*. Indeed, for p to participate in an interaction, all the ports belonging to at least one of $a_1, \ldots, a_n$ must participate. Thus, we can say that the participation of a_i, for some $i \in [1, n]$, in an interaction *is the reason why p can participate*.

2.2 Architectures: Design patterns for BIP

In this section, I will provide an overview of the theory of architectures—formal design patterns that compositionally enforce properties on BIP models—and the RSD flow instantiation based on that theory developed in the *Catalogue of System and Software Properties* (CSSP) project. The objective of the CSSP instantiation of the RSD flow is to provide means for early prototyping of on-board nanosatellite software based on a set of system requirements.

"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

---

[8]Omitted in this manuscript for the sake of brevity.

[9]I denote $\mathbb{B} \stackrel{def}{=} \{true, false\}$ and $\mathbb{B}[P]$ the Boolean algebra generated by the set P.

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." [Att+14; Att+16]

Given an algebra of components $\mathbf{A}$ (see Section 2.1.1) and some formalism $L_{\mathbf{A}}$ for the specification of properties (typically, a logic with the corresponding satisfaction relation), *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.

Thus, an architecture is an operator $A : \mathbf{A}^n \to \mathbf{A}$, imposing a characteristic property $\Phi \in L_{\mathbf{A}}$. It is defined by a glue operator gl and a finite set of coordinating components $\mathcal{D} \subset \mathbf{A}$,[10] such that:

- A transforms a set of components $C_1, \ldots, C_n$ into a composite component $A(C_1, \ldots, C_n) \stackrel{def}{=} gl\langle C_1, \ldots, C_n, \mathcal{D}\rangle$;

- if $A(C_1, \ldots, C_n)$ is semantically valid, it meets the characteristic property Φ (recall, Section 2.1.1, that the semantic mapping σ is partial).

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 BIP connectors (cf. Section 2.1) are memoryless. Hence, they can only be used to impose state properties. Imposing more complex safety properties requires additional coordinating behaviour. Similarly, for distributed architectures, interactions are point-to-point by asynchronous message passing. Synchronisation among the components is achieved by stateful protocols.

An architecture-based design process providing correctness by construction relies on libraries of architectures, each proven to enforce its corresponding characteristic property. Ideally,[11] the design of a system then consists in 1) formalising the system specification as a set of properties; 2) identifying the atomic components that provide the business-specific functionalities; 3) for each property from the system specification, identifying an architecture that enforces that or a stronger property; 4) applying the combination of the identified architectures to the atomic components. Thus, a key fundamental question is *how to combine several architectures while preserving all their corresponding characteristic properties?* 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?

In the remainder of this chapter, I will present the instanciation of the notion of architectures for the BIP component framework.

2.2.1 The basic model

As discussed above, an architecture can be seen as an operator that transforms a set of components into a new composite component. In the context of BIP, it generalises interaction models, by introducing stateful coordinating components. The interface of an architecture is a set of ports that comprises both the ports of the coordinating components and additional dangling ports that must belong to operand components, to which the architecture is applied.

Definition 2.2.1 (Architecture). An *architecture* is a tuple $A = (\mathcal{D}, P_A, \gamma)$, where $\mathcal{D}$ is a finite set of *coordinating components* with pairwise disjoint sets of ports, P_A is a set of ports, such

---

[10] In all examples that we have encountered, coordinating components were atomic, i.e. $\mathcal{D} \subset \mathbf{C}$ (see Section 2.1.1). However, there is no reason to impose this restriction in general.

[11] We discuss the deviations from this ideal and their impact on the design process later in the section.
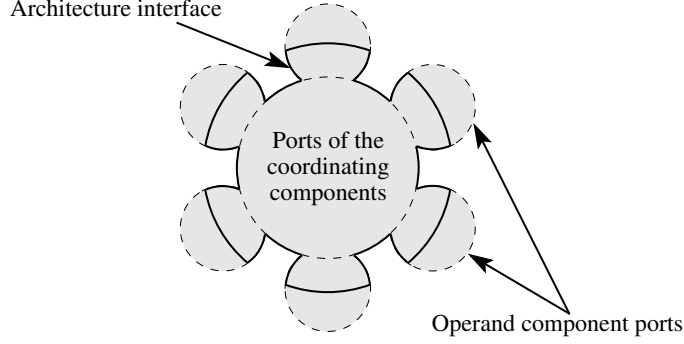
Figure 2.3: A diagram illustrating the relation between ports of an architecture and of its operand components: the inner circle represents the ports of the coordinating components, the "ears" represent the ports of operand components, the representation of the architecture interface is delimited by the solid line

that $\bigcup_{D \in \mathcal{D}} P_D \subseteq P_A$, and $\gamma \subseteq 2^{P_A}$ is an interaction model over P_A. The ports belonging to $P_A \setminus \bigcup_{D \in \mathcal{D}} P_D$ are the *dangling ports* of the architecture.

Definition 2.2.2 (Application of an architecture). Let $A = (\mathcal{D}, P_A, \gamma)$ be an architecture and let $\mathcal{C}$ be a set of components, such that $\bigcup_{C \in \mathcal{C}} P_C \cap \bigcup_{D \in \mathcal{D}} P_D = \varnothing$ and $P_A \subseteq P \stackrel{def}{=} \bigcup_{C \in \mathcal{C} \cup \mathcal{D}} P_C$.

The *application of the architecture A to the set of components $\mathcal{C}$* is the component $A(\mathcal{C}) \stackrel{def}{=} (\gamma \ltimes P)(\mathcal{C} \cup \mathcal{D})$, where $\gamma \ltimes P \stackrel{def}{=} \{a \subseteq P \mid a \cap P_A \in \gamma\}$ is the *extension* of the interaction model γ to the set of ports P.

Intuitively, an architecture enforces coordination constraints on the components in $\mathcal{C}$. The interface P_A of an architecture A contains all ports of the coordinating components $\mathcal{D}$ and some additional ports, which must belong to the components in $\mathcal{C}$ as illustrated in Figure 2.3. In the application $A(\mathcal{C})$, the ports belonging to P_A can only participate in the interactions defined by the interaction model γ of A. Ports that do not belong to P_A are not restricted and can participate in any interaction. In particular, they can join the interactions in γ.

Proposition 2.2.3. *For an architecture $A = (\mathcal{D}, P_A, \gamma)$ and a set of components $\mathcal{C} \subset \mathbf{A}$ such that $P_A \subseteq \bigcup_{D \in \mathcal{D}} P_D \cup \bigcup_{C \in \mathcal{C}} P_C$, the application of A to $\mathcal{C}$ is valid, i.e. $\sigma(A(\mathcal{C}))$ is defined.*

Clearly, if the interface of the architecture covers all ports of the system, i.e. $P = P_A$, the only interactions allowed in $A(\mathcal{C})$ are those belonging to γ.

Finally, the definition of $\gamma \ltimes P$ requires that an interaction from γ be involved in every interaction belonging to $\gamma \ltimes P$. To allow the ports from $P \setminus P_A$ to be fired independently in $A(\mathcal{C})$, one must have $\varnothing \in \gamma$.

Note 2.2.4 (Data). In order to improve their syntactic expressiveness [Fel90], all the formal notions presented so far—namely, BIP components, glue operators and architectures—have been extended to allow the use of local variables in atomic components as well as data transfer among them [Bli+14; BHM19]. However, I will not present these extensions here formally for the sake of conciseness.

The Failure Monitor architecture example In [BHM19], we have introduced the following refined version of the Failure Monitor architecture from [Mav+16].

We assume given a set of operand components realising a certain business functionality that can *fail* and—following a failure—*resume* operation. When a failure occurs, the Failure Monitor architecture initialises a timer and waits for a duration comprised between Min and Max, which
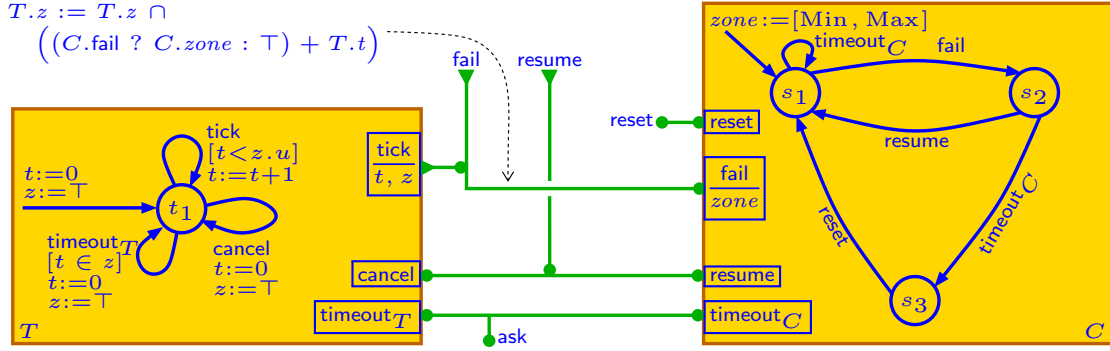
Figure 2.4: The BIP specification of the Failure Monitor architecture

Table 2.2: Interfaces of the coordinating components of the Failure Monitor architecture

Component	Local variables	Ports	Export function
Timer	$\{t, z\}$	$\{\mathsf{tick}, \mathsf{cancel}, \mathsf{timeout}_T\}$	$\{\mathsf{tick} \mapsto \{t, z\}\}$
Control	$\{zone\}$	$\{\mathsf{reset}, \mathsf{fail}, \mathsf{resume}, \mathsf{timeout}_C\}$	$\{\mathsf{fail} \mapsto \{zone\}\}$

are the parameters of the architecture. If by that time the functionality has not been resumed, the architecture issues a request (ask) for the system $reset$ to be triggered externally.[12] These four actions correspond to the four dangling ports of the architecture.

Figure 2.4 shows the coordinating components T (Timer) and C (Control) with the interfaces described in Table 2.2. The export function specifies the local variables that are accessible through each port: variables t and z of the Timer component are accessible only through the port tick. Similarly, variable $zone$ of the Control component is accessible only through the port fail.

Variable t is implicitly assumed to be of type Integer. Variables $z \overset{def}{=} [z.l, z.u]$ and $zone \overset{def}{=} [zone.l, zone.u]$ are of type Integer Interval.

The initial states t_1 and s_1, and valuations $\sigma_T^0 = \{t \mapsto 0, z \mapsto \top\}$, $\sigma_C^0 = \{zone \mapsto [\mathrm{Min}, \mathrm{Max}]\}$ are shown by the incoming arrows $\xrightarrow{t:=0, z:=\top} t_1$ and $\xrightarrow{zone:=[\mathrm{Min,Max}]} s_1$ with $\top = (-\infty, +\infty)$.

Transitions are labelled with ports of the corresponding components, Boolean guards and update assignments on local variables. For instance, the loop transition $t_1 \xrightarrow{\mathsf{tick}, [t<z.u], t:=t+1} t_1$ is labelled by the port tick. It is enabled only when the guard $[t<z.u]$ is satisfied by the local variables of the component. Upon firing, this transition increments the value of the local variable t by 1. The guards and update assignments of the transitions of C are omitted. By default, an omitted guard is $true$ and an omitted assignment is empty $\varnothing$.

The connector $T.\mathsf{tick}\blacktriangleright\!\!-\!\!\bullet(\mathsf{fail}\blacktriangleright\!\!-\!\!\bullet C.\mathsf{fail})$ of Figure 2.4 defines three interactions (cf. bottom row of Table 2.1), each involving a guard ($true$) and a transfer of data between the two coordinating components (see the "$T.z := \dots$" expression in Figure 2.4). The meaning of the expression $C.\mathsf{fail}\ ?\ C.zone : \top$ is the choice between $C.zone$ and $\top$ depending on whether the port $C.\mathsf{fail}$ participates in the interaction or not. Observe that $\top + T.t = \top$ and $T.z \cap \top = T.z$. Thus, the three interactions can be written as follows, simplifying the update assignment of $T.z$ for each

---

[12]This example originates from the CubETH nanosatellite on-board software case study, where the system reset is performed by the battery sub-system.

interaction separately:

$$\big(\{T.\mathsf{tick}\}, true, \varnothing\big),$$
$$\big(\{\mathsf{fail}, T.\mathsf{tick}\}, true, \varnothing\big),$$
$$\big(\{C.\mathsf{fail}, \mathsf{fail}, T.\mathsf{tick}\}, true, T.z := T.z \cap (C.zone + T.t)\big),$$

where $\varnothing$ denotes the absence of assignment (the relation between variables and assignment expressions is empty).

In this example—as in all practical implementations—we implicitly assume the application of the *maximal progress* priority μ, where $(a, g, u) \prec_\mu (b, h, w)$ if $a \subset b$ and $a \neq b$. For instance, the port $T.\mathsf{tick}$ will never fire alone if the port fail is also enabled.

2.2.2 Preservation of properties by architecture composition

As discussed in the openning of this section, we need a way of combining several architectures while preserving their respective characteristic properties. Below, we proceed by 1) defining the composition operator, 2) defining the properties, and 3) stating the desired preservation result.

Definition 2.2.5 (Composition of architectures). Let $A_i = (\mathcal{D}_i, P_{A_i}, \gamma_i)$, for $i = 1, 2$, be two architectures. The *composition* of A_1 and A_2 is the architecture $A_1 \oplus A_2 \stackrel{def}{=} (\mathcal{D}_1 \cup \mathcal{D}_2, P_{A_1} \cup P_{A_2}, \gamma)$, where

$$\gamma = \big\{a \subseteq P_{A_1} \cup P_{A_2} \,\big|\, a \cap P_{A_i} \in \gamma_i, \text{ for } i = 1, 2\big\}. \tag{2.5}$$

Every interaction allowed by $A_1 \oplus A_2$ must comprise both an interaction allowed by A_1 and an interaction allowed by A_2. To allow architecture A_1 to progress independently from A_2, one must have $\varnothing \in \gamma_2$ and vice-versa.

Proposition 2.2.6 (Properties of $\oplus$). *Architecture composition $\oplus$ is commutative and associative; it is idempotent if all coordinating components are deterministic; $A_{id} = \big(\varnothing, \varnothing, \{\varnothing\}\big)$ is its neutral element, i.e. for any architecture A, we have $A \oplus A_{id} = A$.[13] Furthermore, for any component C, we have $A_{id}(C) = C$.*

Note 2.2.7 (n-ary composition of architectures). Notice that while $\oplus$ is defined in Definition 2.2.5 as a binary operator, its associativity allows us to speak of the n-ary composition of architectures.

Definition 2.2.8 (Properties). Let C be a component with $\sigma(C) = (Q, P, \rightarrow, q^0)$. A *property* on C is a state predicate $\Phi : Q \rightarrow \mathbb{B}$. A state $q \in Q$ *satisfies* Φ, written $q \vDash \Phi$, if $\Phi(q) = true$. A property Φ is *initial* if $q^0 \vDash \Phi$. An initial property is an *invariant* if it is satisfied by all reachable states, i.e. holds the formula $\forall q, \big(q^0 \stackrel{*}{\rightarrow} q \implies q \vDash \Phi\big)$.

The main idea of our approach is that an architecture enforces its characteristic property on the set of its operand components. From this point of view, the set of coordinating components is not relevant, neither are their states. Thus, to talk about properties enforced by architectures, we consider properties on the unrestricted composition of the operand components as formalized by the following definition.

Definition 2.2.9 (Enforcing properties). Let $A = (\mathcal{D}, P_A, \gamma)$ be an architecture, let $\mathcal{C}$ be a set of components and let Φ be a property of their parallel composition $A_{id}(\mathcal{C})$ (see Proposition 2.2.6). The *lifting* of Φ to $A(\mathcal{C})$ is defined as $A(\Phi) \stackrel{def}{=} \big\{(q_c, q_d) \,\big|\, q_c \in \Phi, q_d \in \prod_{D \in \mathcal{D}} Q_{\sigma(D)}\big\}$.

We say that A *enforces* Φ *on* $\mathcal{C}$ if, $A(\Phi)$ is an invariant of $A(\mathcal{C})$.

---

[13]Here—and below in similar contexts—we assume given an equivalence relation on the behaviour type **B**, which is then canonically extended to components and architectures. For the sake of simplicity, we write '=' implying equality up to this *semantic* equivalence. Explicit formalisation will be provided in Chapter 3.
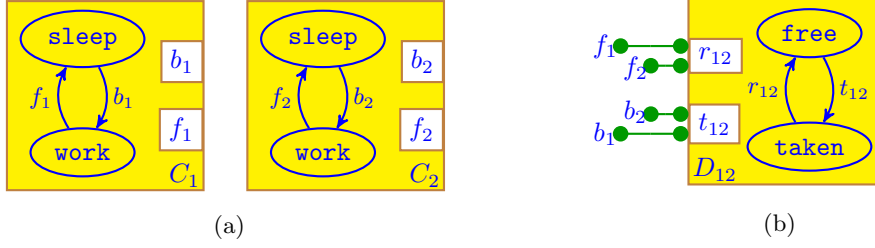
Figure 2.5: Components (*a*) and architecture (*b*) for the Mutual Exclusion architecture example

According to the above definition, when we say that an architecture enforces some property Φ, it is implicitly required that Φ be initial on the coordinated components. Below, we omit mentioning this explicitly.

Theorem 2.2.10 (Preserving enforced properties). *Let $\mathcal{C}$ be a set of components; let $A_i = (\mathcal{D}_i, P_{A_i}, \gamma_i)$, for $i = 1, 2$, be two architectures enforcing on $\mathcal{C}$ the properties Φ_1 and Φ_2 respectively. The composition $A_1 \oplus A_2$ enforces on $\mathcal{C}$ the property $\Phi_1 \wedge \Phi_2$.*

> **Safety or invariants?**
>
> For the sake of clarity I follow here the original papers [Att+14; Att+16], in speaking of invariants rather than of safety properties in full generality. However, the proof of this theorem [Att+16, Theorem 1] is also valid for linear-time safety properties. Furthermore, it can be straightforwardly generalised to branching-time safety.
>
> As a consequence, our definition of properties does not limit the formalisms that can be used for their specification. For instance, in our case studies, we use Computation Tree Logic (CTL).

Consider the special case, where the architecture $A = (\mathcal{D}, P_A, \gamma)$ is expected to be applied to a set of components implementing certain interfaces. Formally, this means that the set of the dangling ports $P_A \setminus \bigcup_{D \in \mathcal{D}} P_D$ is partitioned into sets of ports $P_1, \ldots P_n$ belonging to different operand components. It is common for the characteristic property of A to be specified in some logic as an implication of the form $\bigwedge_{i=1}^{n} \Phi_n \Rightarrow \Psi$, where each of Φ_i is a formula over P_i.[14] In such case, the properties Φ_i are called the *assumptions* of the architecture, whereas Ψ is the *guarantee* it provides. This separation is useful, since it allows decomposing the design process into two phases: 1) the assumptions Φ_i of the architecture are asserted on the operand components—either they are enforced by previous application of architectures or verified by model checking, which, in such case, only concerns a small sub-system of the entire system; 2) the guarantee Ψ is enforced by the application of the architecture.

The Mutual Exclusion architecture example In order to illustrate property enforcement and architecture composition, we take a simpler example of an architecture enforcing the mutual exclusion of critical sections of two processes.

Consider the components C_1 and C_2 in Figure 2.5a. 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}, shown in Figure 2.5b. This architecture comprises a coordinating component D_{12}, and the interaction model $\gamma_{12} = \{b_1 t_{12}, b_2 t_{12}, f_1 r_{12}, f_2 r_{12}\}$.

---

[14]More precisely, one should speak of the occurrence of events modelled by the ports in P_i or the fact of their being enabled or not.
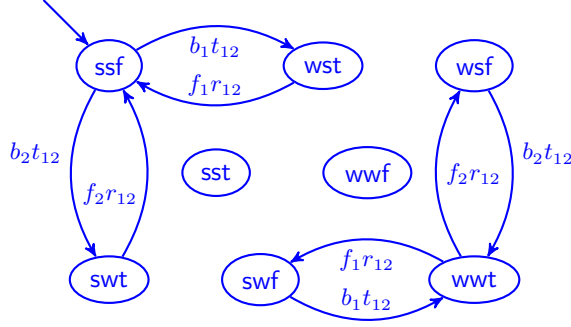
Figure 2.6: Compound behaviour $\sigma\big(A_{12}(C_1, C_2)\big)$ (we abbreviate `sleep`, `work`, `free` and `taken` to `s`, `w`, `f` and `t` respectively)

The compound behaviour $\sigma\big(A_{12}(C_1, C_2)\big)$ is shown in Figure 2.6. 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 (`work`, `work`, `free`) and (`work`, `work`, `taken`) is reachable, i.e. the mutual exclusion property Φ_{12} holds in $A_{12}(C_1, C_2)$.

Notice that, as discussed above, this claim *assumes* that, in both operand components, the critical section—modelled by the state `work`—is "delimited" by the events b_i and f_i. Thus, the characteristic property of the architecture A_{12} can be written in the general case as follows, using CTL:

$$\left((s_1 = \texttt{sleep} \wedge s_2 = \texttt{sleep}) \wedge \bigwedge_{i=1}^{2} \texttt{AG } (f_i \Rightarrow \texttt{A}\big[s_i \neq \texttt{work W } b_i\big]) \right) \Rightarrow \texttt{AG } (s_1 \neq \texttt{work} \vee s_2 \neq \texttt{work}).$$

This allows one to abstract from the precise behaviour of the operand components (e.g. as shown in Figure 2.5a). The first conjunct $(s_1 = \texttt{sleep} \wedge s_2 = \texttt{sleep})$ formalises the global assumption that both components are sleeping in the initial state.

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 2.5b, $\gamma_{13} = \{b_1 t_{13}, b_3 t_{13}, f_1 r_{13}, f_3 r_{13}\}$ and $\gamma_{23} = \{b_2 t_{23}, b_3 t_{23}, f_2 r_{23}, f_3 r_{23}\}$. 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 $\Phi_{23} = (s_2 \neq \texttt{work} \vee s_3 \neq \texttt{work})$.

Consider the application of architectures A_{12} and A_{23} to the three components C_1, C_2 and C_3. The former enforces the property $\Phi_{12} = (s_1 \neq \texttt{work}) \vee (s_2 \neq \texttt{work})$ (the projection of reachable part of the compound behaviour $\sigma\big(A_{12}(C_1, C_2, C_3)\big)$ onto the state-space of the atomic components is shown in Figure 2.7a), whereas the latter enforces $\Phi_{23} = (s_2 \neq \texttt{work}) \vee (s_3 \neq \texttt{work})$ (the projections of reachable states of $A_{23}(C_1, C_2, C_3)$ onto the state-space of the atomic components are shown in Figure 2.7b). By Theorem 2.2.10, the composition $A_{12} \oplus A_{23}$ enforces $\Phi_{12} \wedge \Phi_{23} = (s_2 \neq \texttt{work}) \vee \big((s_1 \neq \texttt{work}) \wedge (s_3 \neq \texttt{work})\big)$, i.e. mutual exclusion between, on the one hand, the `work` state of C_2 and, on the other hand, the `work` states of C_1 and C_3 (see Figure 2.7c). Mutual exclusion between the `work` states of C_1 and C_3 is not enforced. Furthermore, it is easy to check that $A_{12} \oplus A_{23} \oplus A_{13}$ enforces mutual exclusion between the `work` states of C_1, C_2 and C_3 as $\Phi_{12} \wedge \Phi_{13} \wedge \Phi_{23} = \big((s_1 \neq \texttt{work}) \wedge (s_2 \neq \texttt{work})\big) \vee \big((s_1 \neq \texttt{work}) \wedge (s_3 \neq \texttt{work})\big) \vee \big((s_1 \neq \texttt{work}) \wedge (s_3 \neq \texttt{work})\big)$.

2.2.3 Additional verifications

Contradictory requirements   A common problem in practical system design is that of contradictory requirements. For example, the following two properties can be required from an elevator cabin [DG08; PR01]:

(a) $A_{12}(C_1, C_2, C_3)$

(b) $A_{23}(C_1, C_2, C_3)$

(c) $(A_{12} \oplus A_{23})(C_1, C_2, C_3)$

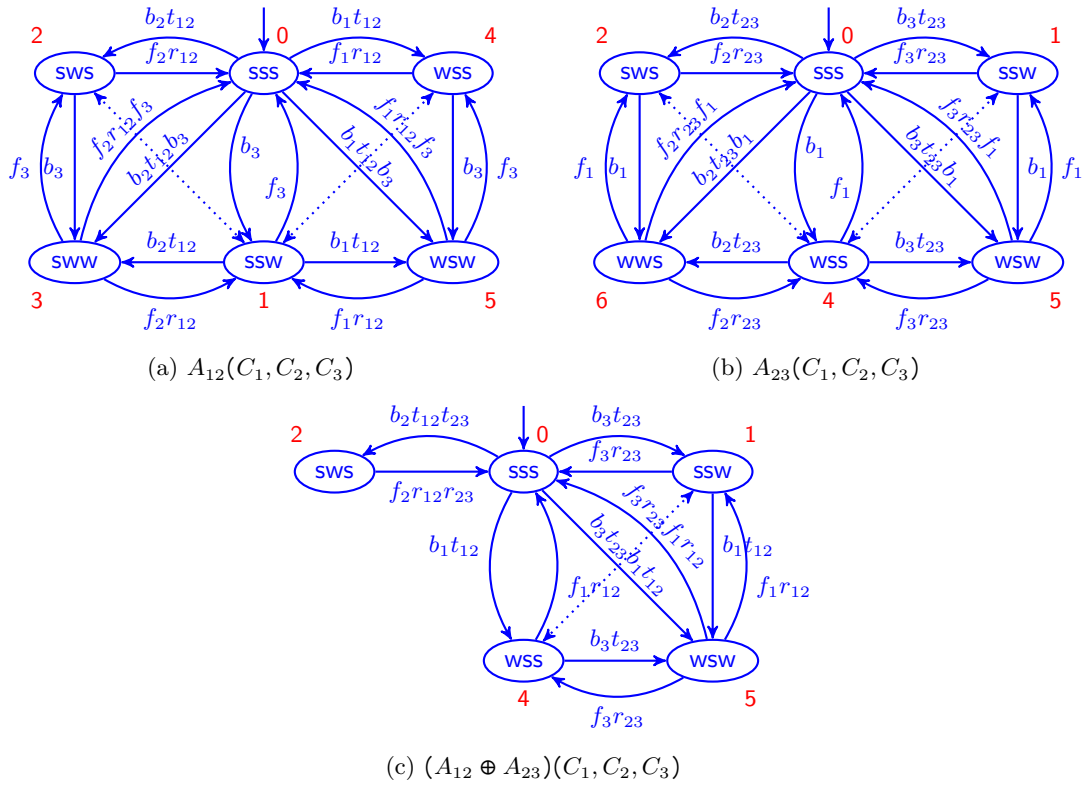Figure 2.7: Projections of reachable states of the compound behaviours onto $\sigma\big(A_{id}(C_1, C_2, C_3)\big)$ (for ease of reading, I omit the transitions indicated by dotted blue arrows; furthermore, for ease of comparison between sub-figures, I additionally label each state with a red number, whereof the main label is the binary representation with $\mathtt{s} = 0$ and $\mathtt{w} = 1$)

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 ⊕ preserves safety properties, it does not preserve deadlock-freedom. However, deadlock-freedom of BIP models can be verified compositionally [Ben+11].

Liveness properties   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 [Att+14]. In [Att+14], 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* with architecture A_2 w.r.t. 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.[15] Notice that the non-interference relation is not commutative.

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.

2.3   From specifications to a system model

In this section, we briefly discuss the design process and tool developed in the *Catalogue of System and Software Properties* (CSSP) project funded by the European Space Agency (ESA). The CSSP design process is a specification and verification process that aims for the early establishment of behavioural correctness throughout the different specification abstraction levels and across the phases of the lifecycle of the satellite on-board software system development. It relies on an ontology-based catalogue of boilerplate requirements and property patterns [Sta+18], as well as on the architecture-based design approach implemented in the BIP framework (Section 2.2). Figure 2.8 provides a high-level illustration of the CSSP process that we have defined in the project; a detailed view is provided in Figure 2.9.

The CSSP *tool* is a GUI front-end for the specification of requirements and properties based on the *Catalogue of Requirement Categories* implemented on top of the *CSSP Ontology*. It is integrated with the Architecture Manipulation Library (AML) used to support the application and composition of formal architectures for property enforcement. From this perspective, the CSSP tool provides also a means for the manipulation of BIP design models.

The CSSP Ontology provides the semantic model for the definition of boilerplates and property patterns. This semantic model includes a representation of the various concepts and entities of the system's domain, which supports the semantic search and reasoning over the repository of requirement and property specifications. Requirement categories may be associated with selected boilerplates on the basis of empirical evidence.

To effectively use the CSSP Ontology and the Catalogue of Requirement Categories, we foresee specific engineering roles (Section 2.3.2) associated with concrete responsibilities and interactions.

---

[15]I refer the reader to [Att+16] for details and examples of architecture (non-)interference.
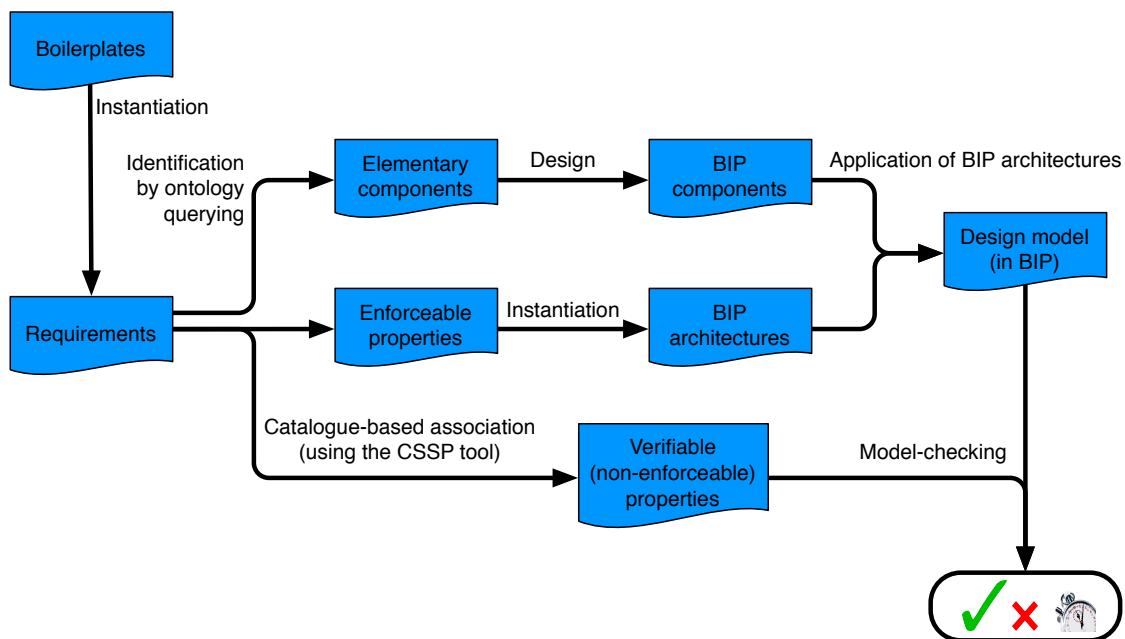
Figure 2.8: High-level illustration of the CSSP design process

2.3.1   The design flow of the CSSP process

The complete CSSP process is illustrated in Figure 2.9. It is built around the CSSP tool, which allows the engineers to 1) query the CSSP Ontology and Catalogue during the requirements and properties specification, 2) progressively define the design model, used for verification of properties, simulation and potential generation of C++ code for (parts of) software. The system model is defined in the BIP language with the assistance of the Architecture Manipulation library.

The definition of ontologies and population of the catalogue activities (in the top-left "Requirement Engineering Domain" box) has been performed as part of the CSSP project and can be later repeated by Requirement Engineers, based on more extensive case studies. The CSSP Ontology and Catalogue of Requirement Categories can be enriched at any time, when new categories, boilerplates and Domain-Specific Ontology (DSO) classes or instances are identified. Instantiation of requirements from boilerplates is a cross-domain activity (Requirement, System and Verification Engineering), which can be performed at any abstraction level, followed by the instantiation of elementary (atomic) components.

All the other activities of the two blocks in the right-hand side of the diagram ("System Engineering Domain" and "Verification Engineering Domain") are performed iteratively, with the exception of Code generation, which may be only performed in Phase C of a space project. Finally, notice that formal architecture application (both in top-down and bottom-up fashion) takes as input system components and desired properties and generates both new components and new properties. Hence, the double arrows in the diagram.

The activities comprising the CSSP process can be roughly grouped into the following main steps, which are performed iteratively, refining the model until all the derived properties are satisfied and all the requirements are discharged:

1. **Requirements are instantiated from boilerplates** available in the CSSP Ontology and Catalogue.

   (*Requirement instantiation* and *Requirement refinement* activities in Figure 2.9, using the integrated CSSP tool.)
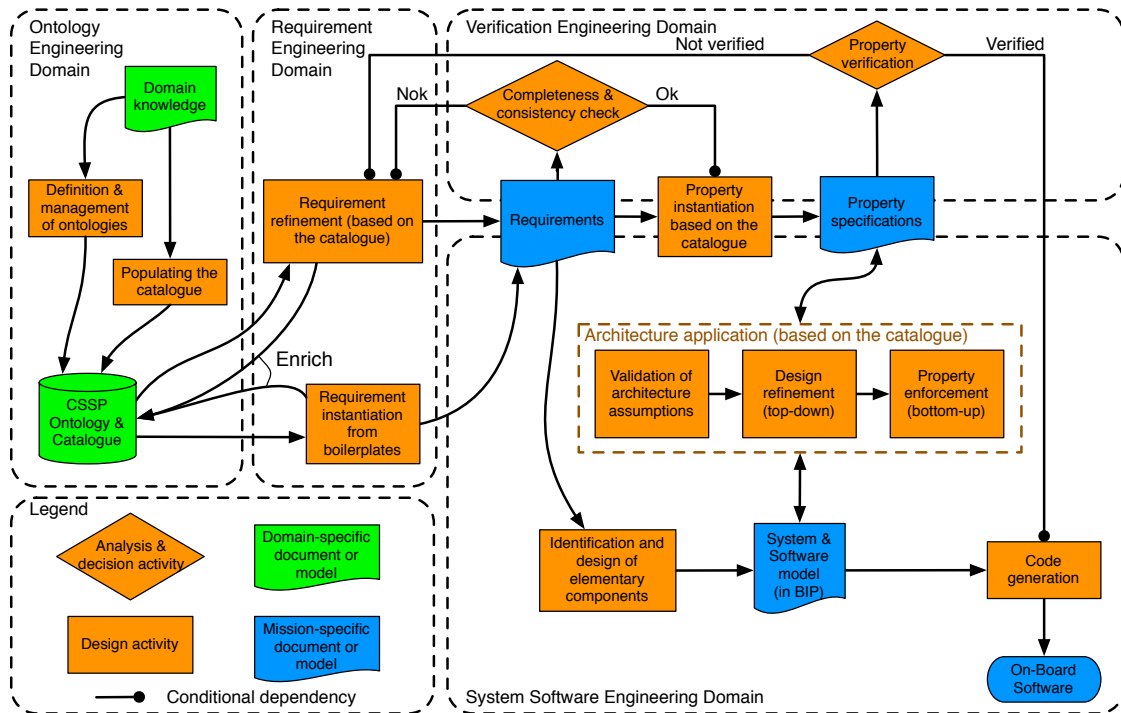
Figure 2.9: The CSSP process

2. **Completeness and consistency of the instantiated requirements is validated** by semantic queries to the CSSP Ontology.

   (*Completeness & consistency check* activity in Figure 2.9, using the integrated CSSP tool.)

3. **Elementary components are identified** based on the specified requirements and the CSSP Ontology; they are used as atomic BIP components in the initial BIP design model.

   (*Identification and design of elementary components* activity in Figure 2.9, using the integrated CSSP tool and a text editor for the BIP design model.)

4. **Properties are instantiated** from the available property patterns

   (*Property instantiation* activity in Figure 2.9, using the integrated CSSP tool.)

5. **Architecture application step:** Formal architectures are applied to elementary components, in order to enforce properties, which are derived from requirements and marked as *enforceable*. This step comprises the following sub-steps:

   (a) **Validation of architecture assumptions:** Some of the formal architectures have associated properties that must be satisfied by operand components, in order for the architecture to be applicable. In such case, model checking is applied to the individual operand components (not the entire model) to verify whether these properties are satisfied.

      (*Validation of architecture assumptions* part of the *Architecture application* activity in Figure 2.9, using the nuXmv[16] model checker.)

   (b) **Design refinement (top-down):** If the above validation (Step 5a) shows that a property is not satisfied, the operand component must be refined. This can involve

---
[16]https://nuxmv.fbk.eu/

- behaviour refinement, i.e. introducing new ports, states and transitions in the finite state machine specifying the component behaviour;

- structural refinement, i.e. replacing the operand component by a combination of two or more smaller components (and any necessary connectors among these).

(*Design refinement* part of the *Architecture application* activity in Figure 2.9, using a text editor.)

(c) **Property enforcement (bottom-up):** Once the operand components satisfy the assumptions of the selected architecture, this latter is applied by adding the coordinating components to the model and modifying the connectors appropriately.

(*Property enforcement* part of the *Architecture application* activity in Figure 2.9, using the AM library within the integrated CSSP tool.)

6. **Additional verification is performed** to ensure that:

(a) the composed system is free from deadlocks;

(b) the applied architectures are mutually non-interfering.

Together, these two analyses ensure that all the enforceable properties associated to the requirements are, indeed, satisfied. More specifically, deadlock-freedom must always be verified, when composing several architectures. Non-interference is only relevant for architectures with associated liveness properties.

(*Property verification* activity in Figure 2.9, using the D-Finder tool.)

7. The BIP model is transformed into the input format of the nuXmv model checker, which is used to check the satisfaction of the properties that cannot be enforced by architectures

(*Property verification* activity in Figure 2.9, using the BIP-to-NuSMV[17] and the nuXmv tools.)

It should be noted that only safety properties were encountered in the case studies realized in the project. Furthermore, there are reasons to believe that the use of liveness properties may not be appropriate in the Space Software and Systems domain. In particular, instead of requiring that an event X shall happen eventually—after an unspecified delay—it is preferable to explicitly define the actions that shall lead to the occurrence of X, or another bounding event Y, such that X must happen before Y. In such case, the property becomes a safety one. For the above reasons, we did not implement the mutual non-interference check of Step 6b above, leaving such an implementation for future work.

2.3.2 Engineering roles involved in the CSSP process

The CSSP process involves well-defined roles that are shown in Figure 2.10. *Ontology Engineers* define the CSSP Ontology and Catalogue, as well as the used boilerplates and property patterns. *Requirements Engineers* define new requirements based on the predefined boilerplates from the CSSP Ontology and Catalogue. *Specification Verification Engineers* define new properties based on property patterns and obtain representations of the properties of interest in the BIP language. *System Software Engineers* develop a BIP design model.

Tables 2.3 and 2.4 list the responsibilities associated to the roles involved in the CSSP process. The responsibilities pertaining to the initial preparation of the CSSP Ontology and Catalogue, adaptation to new mission types and occasional maintenance (e.g. enriching the catalogue in exceptional cases when existing patterns are not sufficient) are listed in Table 2.3. The CSSP approach relies, fundamentally, on the assumption that, at use-time, the CSSP Ontology and

---

[17]https://archiveweb.epfl.ch/risd.epfl.ch/bip2nusmv.html
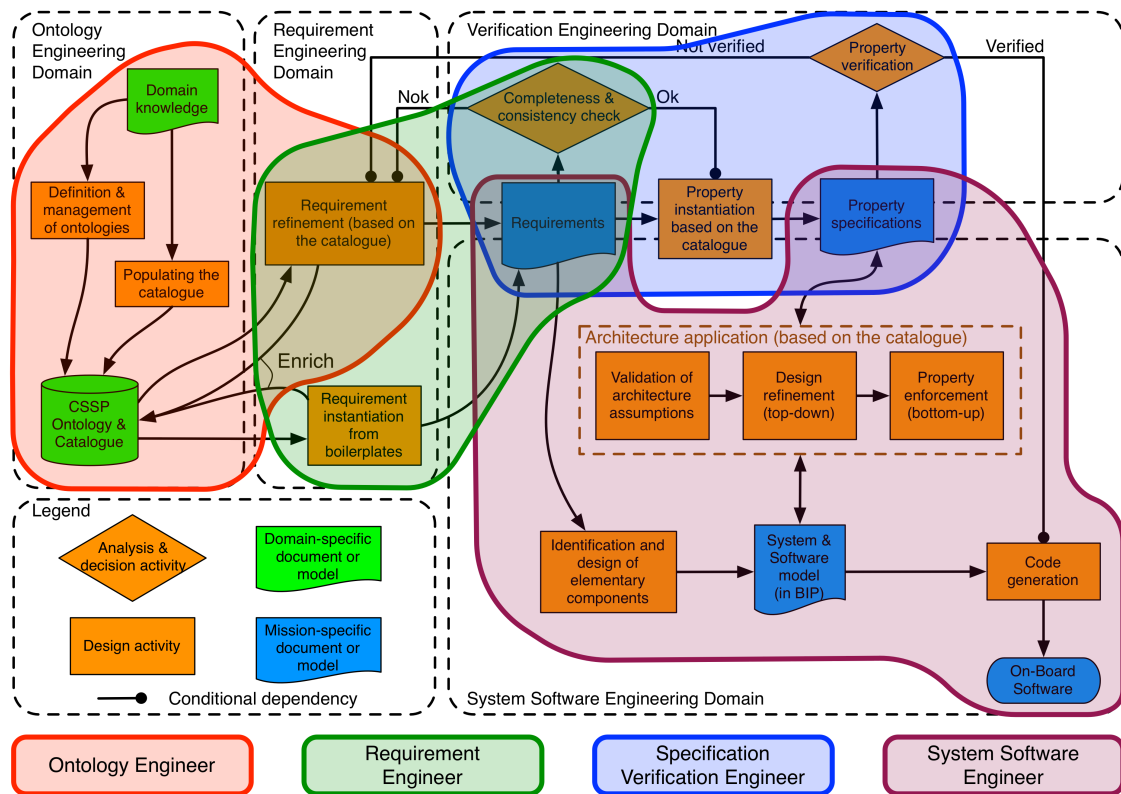
Figure 2.10: Graphical illustration of the engineering roles involved in the CSSP approach

Table 2.3: Preparation and maintenance responsibilities associated with the CSSP process roles

Role	Responsibilities
Ontology Engineer	- Create a CSSP Ontology and Catalogue of Requirement Categories. - Manage the CSSP Ontology by adding new boilerplates and/or property patterns, in particular to adapt to new types of missions. - Extend the CSSP Ontology by defining new domain-specific ontologies for different application domains. (*Only Ontology Engineers can edit the CSSP Ontology and Catalogue.*)
Requirements & Specification Verification Engineers	- Specify the set of property patterns, from which the properties, needed to cover a requirement, will be derived. - Obtain a formal representation of the property patterns in BIP and/or in CTL.

Catalogue have already been populated by experts contributing non-trivial information that cannot—within reasonable costs—be obtained by automated analysis. When the CSSP Ontology and Catalogue has to be extended, similar expertise is unavoidably required. Therefore, responsibilities in Table 2.3 correspond to exceptional activities that are to be carried out as seldom as possible since the completion of the CSSP project.

Table 2.4 lists the responsibilities associated to the various CSSP process roles during the space system project.

2.3.3 Refinement process

Two types of refinement are involved in the process: *refinement of requirements* and *refinement of the design*.

Table 2.4: Design-time responsibilities associated to the CSSP process roles

Role	Responsibilities
Requirements Engineer	- Choose from a list of predefined boilerplates for a category of an abstraction level to create a new requirement. - Fill in the placeholders of the boilerplate with instances of the DSO sub-ontology. - Edit existing requirements based on boilerplates. - Refine requirements based on the input from Completeness & consistency check and Property verification activities. (*Requirements Engineers cannot create new or edit existing boilerplates.*)

Specification Verification Engineer	- Create new properties that will cover a requirement by choosing from the list of property patterns.
	- Replace the placeholders of the property pattern with instances of the DSO sub-ontology.
	- Verify the complete system software model for deadlock-freedom to ensure that the applied formal architectures have not over-constrained the model.
	- When a deadlock is detected in the BIP model, identify a property that can be responsible for the deadlock and attempt replacing it with a weaker alternative, using a different property pattern for the same requirement.
	- Verify the satisfaction of verifiable properties.
	- When a verifiable property is not satisfied,
	
	  (i) notify a System Software Engineer and attempt component refinement (see joint System Software & Specification Verification Engineers responsibilities below);
	  (ii) if component refinement is not possible, replace the property with a weaker alternative, using a different property pattern for the same requirement.
	
	- When behaviour refinement and property relaxation are not sufficient for successful validation, notify a Requirement Engineer and request a refinement of requirements.

System Software & Specification Verification Engineers	- Verify that the operand components, to which formal architectures are applied, satisfy the properties assumed by these architectures. - When an assumed property of a formal architecture is not satisfied by an operand component or—upon request from a Specification Verification Engineer—when a verifiable property is not satisfied by the design model, refine component behaviour to ensure property satisfaction.
System Software Engineer	- Provide a design model by specifying behaviour at the required abstraction level. - In particular, using semantic queries, identify the elementary components that appear in the requirements and provide BIP components implementing their behaviour. - Enforce properties using the BIP correctness-by-construction techniques: instantiate formal BIP architectures and apply them to the design model.

Refinement of requirements is performed by a Requirement Engineer—in collaboration with an Ontology Engineer if the CSSP Ontology and/or Catalogue need be updated (see Figure 2.10)—when a set of requirements is found to be inconsistent. Such inconsistencies can either be detected through the semantic queries to the CSSP Ontology, or manifest themselves through deadlocks, which are detected during property validation. In the former case, the semantic query directly identifies the conflicting requirement. In the latter, an inconsistency is associated to a set of properties. Each property can be traced to the requirement, from which it arose upstream in the process.

An inconsistency can be due either to a mistake during the requirement specification phase (wrong requirement) or to an overly weak assumption on the environment of the involved entities (thus leading to overly strong, hence, inconsistent requirements). In the first case, the wrong requirement is dropped, starting the next design iteration (completeness analysis, additional requirements, property validation etc.). In the second case, where none of the requirements can be identified as outright wrong, additional assumptions must be made on the environment in order
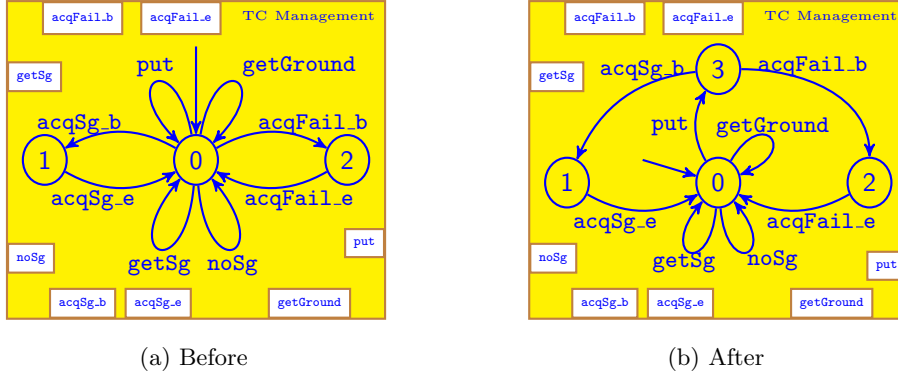
(a) Before   (b) After

Figure 2.11: Behavior refinement to ensure the validity of architecture assumptions

to weaken the existing requirements. This process can be assisted by exploiting the ontological knowledge about the semantics of the entities relevant to the requirements in question.

Design refinement is performed by a System Software Engineer (see Figure 2.10) in two situations: 1) validation of architecture assumptions during the application of formal architectures and 2) when moving to a more detailed requirement specification level.

For instance, the standard ESA procedure separates requirement formulation into several phases, among which the first Requirement Baseline (RB) specification is refined by the Technical Specification (TS) one.

Validation of architecture assumptions: Before an architecture is applied, the assumptions that it makes on the operand components have to be validated. Such assumptions are formulated as CTL properties referring to the behaviour of the operand components; they are verified by model checking.

When an assumed property of a formal architecture is not verified by a corresponding operand component, the behaviour of this component must be refined. An example of such a refinement is provided in [Bli+16, Section 2.6.1]. An Action Flow with Abort architecture is applied to the `TC Management` component shown in Figure 2.11a. The behaviour of this component violates the following assumed property of the architecture:

$$\mathtt{AG}\,\big(put \to \mathtt{AX}\,\mathtt{A}\,[\neg put\,\mathtt{W}\,(acqSg\_e \lor acqFail\_e)]\,\big).$$

In Figure 2.11b, this property is enforced by introducing the additional state 3.[18]

Moving to a new requirement specification level: When moving to a finer specification level, some actions (e.g. services) are usually specified in more detail than on the previous level. In such cases, an action `a` in the abstract component is replaced by a sequence of several actions, potentially involving branching and loops. Although, in principle, this can be done directly in the finite state machine defining the corresponding component behaviour, it is recommended, instead, to introduce a new component, whereof the behaviour realizes precisely the action `a`. Such a component would have two dedicated ports, say `begin` and `end`, corresponding to the invocation and to the termination of `a`. With this latter approach, the only modification that is necessary in the component under refinement, consists then in replacing each occurrence of `a` with a sequence of two actions `a_b` and `a_e` synchronized, respectively with the actions `begin` and `end` of the new component.

---

[18]Depending on whether the CSSP process is used for early requirement validation or for the design of system software, this refinement could take place during different phases: RB or TS definition, respectively.

To ensure that such refinement does not introduce deadlocks or violate previously validated properties, one has to verify that the newly added component satisfies the following two derived properties:

$$\texttt{AF } end \qquad\qquad\qquad\qquad\qquad \text{— the action always terminates,}$$

$$\texttt{AG}\big(end \rightarrow \texttt{AX A}\big[\neg end \texttt{ W } begin\big]\big) \quad \text{— once the action terminates, nothing happens unless it is invoked again.}$$

In order to guarantee the preservation of already established properties, one must—in addition to the verification of the above assumed or derived properties—check an appropriate refinement relation between the initial and the refined component behaviours. This is achieved by establishing a simulation relation [Mil89] or by checking action refinement [GG01], respectively, for the first and the second case of design refinement above.

> Notice that the CSSP process does not impose any particular techniques for the verification of properties, nor for the refinement checking discussed in this section.

2.4 Key contributions

The work presented in this chapter spans a period over 10 years long. It starts with two foundational papers that we have co-authored with Joseph Sifakis during my post-doc at Verimag—the EMSOFT paper [BS07] and its extended journal version [BS08b]—which formalise the BIP operational semantics and, in particular, connectors used for the structured specification of interactions. The journal paper [BS10] explores the idea of exhibiting implicit interaction causality, whereby participation of some ports depends on that of the others. In particular, the Boolean encoding defined in that paper underlies the notion of **Require** and **Accept** macros used in [Boz+12a] and in JavaBIP presented in Chapter 4.

The notion of architectures, embodying BIP design patterns, was proposed in [Att+14; Att+16] (co-authored, in particular, with my former PhD student Eduard Baranov). It was then applied in the context of two collaborations to set up the architecture-based design flow. In collaboration with the EPFL Space Engineering Center we have realised the first case study [Mav+16]. In a collaborative project with the Aristotle University of Thessaloniki and Thales-Alenia Space (France) funded by the European Space Agency, we have generalised this approach extending it with the ontology-based tooling for the elicitation of requirements and generation of BIP models [Sta+18]. The latter two papers were co-authored, in particular, with my former PhD student Anastasia Mavridou.

The above papers study interaction and architectures purely from the structural perspective, disregarding data transfer among the local variables of the components. A formalisation of data transfer is proposed in [Bli+14], whereas [BHM19] extends the architecture composability results to architectures involving data.

In chronological order

[BS07]   Simon Bliudze and Joseph Sifakis. "The Algebra of Connectors—Structuring Interaction in BIP". In: *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software, EMSOFT 2007*. ACM SigBED. Salzburg, Austria, Oct. 2007, pp. 11–20. DOI: 10.1145/1289927.1289935.

[BS08b]   Simon Bliudze and Joseph Sifakis. "The Algebra of Connectors—Structuring Interaction in BIP". In: *IEEE Transactions on Computers* 57.10 (2008). Pp. 1315–1330. ISSN: 0018-9340. DOI: 10.1109/TC.2008.26.

[BS10]   Simon Bliudze and Joseph Sifakis. "Causal semantics for the algebra of connectors".
            In: *Formal Methods in System Design* 36.2 (June 2010). Pp. 167–194. DOI: 10.1007/
            s10703-010-0091-z.

[Bli+14]   Simon Bliudze, Joseph Sifakis, Marius Dorel Bozga, and Mohamad Jaber. "Archi-
            tecture Internalisation in BIP". In: *Proceedings of the 17th International ACM Sig-
            soft Symposium on Component-based Software Engineering (CBSE '14)*. Marcq-en-
            Barœul, France: ACM, 2014, pp. 169–178. ISBN: 978-1-4503-2577-6. DOI: 10.1145/
            2602458.2602477.

[Att+16]   Paul Attie, Eduard Baranov, Simon Bliudze, Mohamad Jaber, and Joseph Sifakis.
            "A General Framework for Architecture Composability". In: *Formal Aspects of Com-
            puting* 18.2 (Apr. 2016). Open access, pp. 207–231. DOI: 10.1007/s00165-015-0349-8.

[Bas+16]   Nick Bassiliades, Simon Bliudze, Panagiotis Katsaros, and Emmanouela Stachtiari.
            General Concept and Technical Approach. Tech. rep. ITT-AO1-7785-D6. EPFL IC
            IIF RiSD & AUTh, Feb. 2016.

[Mav+16]   Anastasia Mavridou, Emmanouela Stachtiari, Simon Bliudze, Anton Ivanov, Panagi-
            otis Katsaros, and Joseph Sifakis. "Architecture-based Design: A Satellite On-board
            Software Case Study". In: *13th International Conference on Formal Aspects of Com-
            ponent Software (FACS 2016)*. Vol. 10231. Lecture Notes in Computer Science. 2016,
            pp. 260–279. DOI: 10.1007/978-3-319-57666-4_16.

[Sta+18]   Emmanouela Stachtiari, Anastasia Mavridou, Panagiotis Katsaros, Simon Bliudze,
            and Joseph Sifakis. "Early validation of system requirements and design through
            correctness-by-construction". In: *Journal of Systems and Software* 145 (Nov. 2018).
            Pp. 52–78. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2018.07.053.

[BHM19]   Simon Bliudze, Ludovic Henrio, and Eric Madelaine. "Verification of concurrent
            design patterns with data". In: *Proc. of the 21st International Conference on Coor-
            dination Models and Languages (COOORDINATION 2019)*. Ed. by Emilio Tuosto
            and Hanne-Riis Nielsen. Vol. 11533. LNCS. Springer, June 2019, pp. 161–181. DOI:
            10.1007/978-3-030-22397-7_10.

A formal study of expressiveness

In order to understand the applicability limits of a design approach, one has to study the expressive power of the underlying component-based framework. However, for such a study to be possible a proper comparison framework has to be developed. Indeed, most expressiveness studies focus on two questions: 1) *what can be computed?* and 2) *how concise is the program?* The first question is typically answered by a comparison to the computing power of Turing machines. The answer to the second question can be summarised—admittedly in a somewhat simplistic manner—by saying that parts of a language represent *syntactic sugar* w.r.t. another language. None of these two approaches captures the essence of component-based design, where given composition operators are applied to a set of components to build a system but they cannot change the components themselves.

In this chapter, I present

- a formal algebraic framework that allows the comparison of the expressive power of component-based frameworks,

- its application to a study of the expressiveness of the BIP framework,

- an alternative, *offer* semantics of BIP and its relation with the classical one.

3.1 Algebraic formalisation

3.1.1 Basic definitions

I will start by refining the basic model used in Section 2.1.1 of the previous chapter. Essentially, this boils down to equipping the semantic domain of an algebra of components with an equivalence relation.

Indeed, as I have implicitly suggested in the previous chapter, every component-based design framework can be viewed as an algebra of components equipped with a semantic mapping. The algebra of components *syntactically* defines the composite components that can be assembled from a given set of the atomic ones. The semantic mapping associates to each component its corresponding behaviour. The codomain of the semantic mapping, which we call the *semantic domain* consists of a behaviour type—defined in terms of Labelled Transition Systems or a similar formalism—and an associated equivalence relation. This can be formalised as follows:

Definition 3.1.1. A *component-based framework* is a tuple $(\mathbf{G}, \mathbf{C}, \mathbf{B}, \simeq, \sigma)$, where

- $\mathbf{G}$ is a set of *composition (glue) operators*, we denote by $\mathbf{G}^n \subseteq \mathbf{G}$, with $n \in \mathbb{N}$, the subset of n-ary operators,

- $\mathbf{C}$ is a *set of atomic components*,

- $(\mathbf{B}, \simeq)$ is a *semantic domain*, consisting of a *behaviour type* $\mathbf{B}$ and an *equivalence relation* $\simeq \subseteq \mathbf{B} \times \mathbf{B}$,

- $\sigma : \mathbf{A} \rightharpoonup \mathbf{B}$ is a partial *semantic mapping* from the algebraic structure

$$\mathbf{A} ::= C \mid f\langle C_1, \ldots, C_n \rangle, \qquad C \in \mathbf{C}, \ n \in \mathbb{N}, \ C_1, \ldots, C_n \in \mathbf{A} \text{ and } f \in \mathbf{G}^n,$$

generated by $\mathbf{G}$ from $\mathbf{C}$, which we call the *algebra of components* of the framework. (Notice that $\mathbf{A}$ does not appear explicitly in the tuple, since it is fully defined by $\mathbf{G}$ and $\mathbf{C}$.)

We call the elements of $\mathbf{A}$ *components* and the elements of $\mathbf{B}$ *behaviours*. The algebraic structure $\mathbf{A}$ represents the set of all systems constructible within the framework.

The behaviour type $\mathbf{B}$ defines the semantic nature of the components manipulated by the framework. The equivalence relation $\simeq \subseteq \mathbf{B} \times \mathbf{B}$ allows comparing components in terms, for example, of their functionality, observable behaviour or capability of interaction with the environment. It is canonically lifted to $\mathbf{A}$ by putting, for any $C_1, C_2 \in \mathbf{A}$,

$$C_1 \simeq C_2 \overset{def}{\Longleftrightarrow} \begin{cases} \text{both } \sigma(C_1) \text{ and } \sigma(C_2) \text{ are defined,} \\ \qquad \text{and} \\ \sigma(C_1) \simeq \sigma(C_2). \end{cases} \tag{3.1}$$

The semantic mapping $\sigma : \mathbf{A} \rightharpoonup \mathbf{B}$ assigns to each component its meaning in terms of the behaviour type $\mathbf{B}$: for any $C \in \mathbf{A}$, we say that $\sigma(C)$ is the behaviour of C.

Example 3.1.2. Consider the framework $CCS^- = (\mathbf{G}^-, \mathbf{C}, \mathbf{B}, \simeq, \sigma^-)$, taking both $\mathbf{C}$ and $\mathbf{B}$ to be the subset of purely sequential processes in CCS [Mil89]:

$$\mathbf{C} = \mathbf{B} ::= 0 \mid l.P \mid P_1 + P_2, \qquad\qquad l \in \mathcal{L}, \ P, P_1, P_2 \in \mathbf{B},$$

where $\mathcal{L} = \{\tau\} \cup \{a, \bar{a} \mid a \in \mathcal{A}\}$, for some given set of actions $\mathcal{A}$. Although the specific choice of the equivalence relation is irrelevant for the purposes of this example, we can take $\simeq$ to be the branching bisimilarity relation [Bas96]. Take $\mathbf{G} = \{\|, \backslash A\}$, where $\|$ is the classical binary CCS parallel composition (replacing the synchronisation of a and $\bar{a}$, for any $a \in \mathcal{A}$, with τ) and $\backslash A$ is the unary restriction operator, which hides all actions in the set $A \subseteq \mathcal{A}$ by replacing them with τ. The semantic mapping σ is defined trivially for restriction $\backslash A$ and through the expansion lemma [Mil89], for parallel composition $\|$. $\diamond$

Definition 3.1.3. The semantic mapping is called *structural*, if it is defined by associating to each n-ary glue operator $f \in \mathbf{G}^n$ a corresponding partial mapping $\hat{f} : \mathbf{B}^n \rightharpoonup \mathbf{B}$ and putting

$$\sigma\big(f\langle C_1, \ldots, C_n \rangle\big) \overset{def}{=} \hat{f}\big(\sigma(C_1), \ldots, \sigma(C_n)\big),$$

for all $n \in \mathbb{N}, C_1, \ldots, C_n \in \mathbf{A}$.

We call $\{\hat{f} \mid f \in \mathbf{G}\}$ the set of *defining mappings* of σ.

Example 3.1.4. Clearly, the semantic mapping in Example 3.1.2 is structural with the following defining mappings.

We adopt the usual simplifying notation by postulating $\bar{\bar{a}} = a$, for all $a \in \mathcal{A}$. We then take the defining mapping $\hat{\|} : \mathbf{B}^2 \rightharpoonup \mathbf{B}$ for the parallel composition operator to be symmetrical, defined by putting

$$\hat{\|}(B_1, B_2) \overset{def}{=} \begin{cases} B_1, & \text{if } B_2 = 0, \\ \hat{\|}(B_1, P_1) + \hat{\|}(B_1, P_2), & \text{if } B_2 = P_1 + P_2, \\ l_1.\hat{\|}(P_1, B_2) + l_2.\hat{\|}(B_1, P_2), & \text{if } B_1 = l_1.P_1 \wedge B_2 = l_2.P_2, \text{with } l_1 \neq \overline{l_2}, \\ l.\hat{\|}(P_1, B_2) + \bar{l}.\hat{\|}(B_1, P_2) + \tau.\hat{\|}(P_1, P_2), & \text{if } B_1 = l.P_1 \wedge B_2 = \bar{l}.P_2, \text{with } l, \bar{l} \neq \tau. \end{cases}$$

The semantic mapping $\widehat{\backslash A} : \mathbf{B} \to \mathbf{B}$ for the restriction operator is defined by putting

$$\widehat{\backslash A}(B) \stackrel{def}{=} \begin{cases} 0, & \text{if } B = 0, \\ l.\widehat{\backslash A}(P), & \text{if } B = l.P \wedge l \notin A, \\ \tau.\widehat{\backslash A}(P), & \text{if } B = l.P \wedge l \in A, \\ \widehat{\backslash A}(P_1) + \widehat{\backslash A}(P_2), & \text{if } B = P_1 + P_2. \end{cases}$$

Alternatively, any semantic mapping defined using Structural Operational Semantics rules [Plo81] is, indeed, structural. $\diamond$

3.1.2 Properties of component-based frameworks

Ideally, glue operators used to compose systems in a component-based design framework must possess the following properties, most of which have been initially stated in a less formal manner in [Sif05].

Incrementality   This property represents a generalised form of associativity. It requires that it be possible to view the sub-systems of a system in separation:

$$\forall n \in \mathbb{N}, \ \forall i \in [1, n], \ \forall C_1, C_2, \ldots, C_n \in \mathbf{A}, \ \forall f \in \mathbf{G}^n, \ \exists g \in \mathbf{G}^2, h \in \mathbf{G}^{n-1} :$$
$$f\langle C_1, C_2, \ldots, C_n \rangle \simeq g\langle C_i, h\langle C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n \rangle \rangle. \quad (3.2)$$

Flattening   This property is complementary to incrementality. It requires that, for any system obtained by hierarchically applying two glue operators to a finite set of sub-systems, there must exist an equivalent system obtained by applying a single glue operator to the *same* sub-systems:

$$\forall n \in \mathbb{N}, \ \forall i \in [1, n], \ \forall j \in [i, n],$$
$$\forall C_1, C_2, \ldots, C_n \in \mathbf{A}, \ \forall f \in \mathbf{G}^{n-j+i}, \ \forall g \in \mathbf{G}^{j-i+1}, \ \exists h \in \mathbf{G}^n :$$
$$f\langle C_1, \ldots, C_{i-1}, g\langle C_i, \ldots, C_j \rangle, C_{j+1}, \ldots, C_n \rangle \simeq h\langle C_1, \ldots, C_n \rangle. \quad (3.3)$$

In other words, $\mathbf{G}$ must be closed under composition. Flattening enables model transformations, e.g. for optimising code generation or component placement on multicore platforms [Bon+10; BJS09].

Uniform flattening   This property strengthens the previous one by requiring the operator h to be the same, independently of the choice of $C_1, \ldots, C_n$.

$$\forall n \in \mathbb{N}, \ \forall i \in [1, n], \ \forall j \in [i, n],$$
$$\forall f \in \mathbf{G}^{n-j+i}, \ \forall g \in \mathbf{G}^{j-i+1}, \ \exists h \in \mathbf{G}^n, \ \forall C_1, C_2, \ldots, C_n \in \mathbf{A} :$$
$$f\langle C_1, \ldots, C_{i-1}, g\langle C_i, \ldots, C_j \rangle, C_{j+1}, \ldots, C_n \rangle \simeq h\langle C_1, \ldots, C_n \rangle. \quad (3.4)$$

Compositionality   This property requires that glue operators preserve the equivalence of their operands:

$$\forall n \in \mathbb{N}, \ \forall i \in [1, n], \ \forall C_1, \ldots, C_n, C_i' \in \mathbf{A}, \ \forall f \in \mathbf{G}^n,$$
$$C_i \simeq C_i' \implies f\langle C_1, \ldots, C_i, \ldots C_n \rangle \simeq f\langle C_1, \ldots, C_i', \ldots C_n \rangle. \quad (3.5)$$

Relaxed compositionality A weaker version of compositionality requires that glue operators only preserve the equivalence of atomic components:

$$\forall n \in \mathbb{N}, \; \forall i \in [1, n], \; \forall C_1, \ldots, C_n, C_i' \in \mathbf{C}, \; \forall f \in \mathbf{G}^n,$$

$$C_i \simeq C_i' \implies f\langle C_1, \ldots, C_i, \ldots C_n \rangle \simeq f\langle C_1, \ldots, C_i', \ldots C_n \rangle. \quad (3.6)$$

Notice that, combined with flattening, this relaxed notion of compositionality is already quite strong: essentially, compositionality allows replacing sub-systems, whereas relaxed compositionality with flattening allow replacing atomic behaviours.

Proposition 3.1.5. *If the semantic mapping is structural and its defining mappings preserve the equivalence $\simeq$ then the framework has compositionality:*

Proof. Take $n \in \mathbb{N}$ and consider $C_1, \ldots, C_n \in \mathbf{A}$ and $f \in \mathbf{G}^n$. Without loss of generality, let $i = 1$ and take $C_1' \in \mathbf{A}$, such that $C_1 \simeq C_1'$. We then have

$$\sigma\big(f\langle C_1, \ldots, C_n \rangle\big) \; = \; \hat{f}\big(\sigma(C_1), \ldots, \sigma(C_n)\big) \; \simeq \; \hat{f}\big(\sigma(C_1'), \ldots, \sigma(C_n)\big) \; = \; \sigma\big(f\langle C_1', \ldots, C_n \rangle\big).$$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \square$

The central subject of this manuscript is the BIP framework. Glue operators in BIP are n-ary. Hence, we will focus our attention on compositionality and flattening, disregarding incrementality. Indeed, as mentioned above, incrementality can be viewed as generalised associativity, which is mainly useful, in our context, to be able to reason about binary operators and generalise the results to n-ary ones.

3.1.3 Comparing the expressiveness

In order to define the notions necessary for comparing the expressiveness of component-based frameworks, we first introduce the following technical definition.

Definition 3.1.6. Given a framework $F = (\mathbf{G}, \mathbf{C}, \mathbf{B}, \simeq, \sigma)$ and a set of variables $\mathcal{Z}$, we will denote by $\mathbf{G}[\mathcal{Z}]$ the set of *expressions on variables in* $\mathcal{Z}$, defined by the following grammar:

$$\mathbf{G}[\mathcal{Z}] ::= Z \mid f\langle E_1, \ldots, E_n \rangle, \qquad Z \in \mathcal{Z}, \; n \in \mathbb{N}, \; E_1, \ldots, E_n \in \mathbf{G}[\mathcal{Z}] \text{ and } f \in \mathbf{G}^n.$$

Comparing the expressiveness of two component-based frameworks is only possible when their semantic domains coincide. Indeed, two component-based frameworks with distinct semantic domains can be compared by mapping to a common behaviour type and taking an appropriate equivalence relation consistent with those of the frameworks. However, this essentially boils down to a substitution of the semantic domains, i.e. considering a different pair of frameworks.

Below, we define two preorders—strong and weak—that allow us to compare component-based frameworks with the same semantic domain. Intuitively, one framework has *strong* full expressiveness w.r.t. another if any operator of the second has an equivalent one in the first. *Weak* full expressiveness allows an expression, i.e. a composition of several operators, to be used instead of a single equivalent operator in the first framework.

Definition 3.1.7. Given two frameworks $F_i = (\mathbf{G}_i, \mathbf{C}_i, \mathbf{B}, \simeq, \sigma_i)_{i \in \{1,2\}}$ with the same semantic domain, we say that F_1 has *strong full expressiveness w.r.t.* F_2, denoted $F_2 \blacktriangleleft F_1$ if

$$\forall n \in \mathbb{N}, \; \forall f \in \mathbf{G}_2^n, \; \exists \tilde{f} \in \mathbf{G}_1^n : \; \forall C_1^i, \ldots, C_n^i \in \mathbf{A}_i,$$

$$\bigwedge_{k=1}^n \sigma_1(C_k^1) \simeq \sigma_2(C_k^2) \implies \sigma_1\big(\tilde{f}\langle C_1^1, \ldots, C_n^1 \rangle\big) \simeq \sigma_2\big(f\langle C_1^2, \ldots, C_n^2 \rangle\big). \quad (3.7)$$

Table 3.1: Expressiveness comparison relations (three "less expressive" relations indicated in grey are symmetric to the "more expressive" ones; the two relations in the second row are not used in this manuscript—they are only presented for the sake of completeness)

$F_1?F_2$ / $F_2?F_1$	◄	◄ ∧ ⫡̸	⫡̸
◄	F_1 and F_2 have strongly equivalent expressiveness $F_1 \Leftrightarrow F_2$	F_1 is weakly more expressive than F_2 $F_2 \to F_1$	F_1 is strongly more expressive than F_2 $F_2 \Rightarrow F_1$
◄ ∧ ⫡̸	F_1 is weakly *less* expressive than F_2 $F_2 \leftarrow F_1$	F_1 and F_2 have weakly equivalent expressiveness $F_1 \leftrightarrow F_2$	F_1 is slightly more expressive than F_2 $F_2 \dashrightarrow F_1$
⫡̸	F_1 is strongly *less* expressive than F_2 $F_2 \Leftarrow F_1$	F_1 is slightly *less* expressive than F_2 $F_2 \dashleftarrow F_1$	F_1 and F_2 are incomparable $F_1 \leftrightarrow\!\!\!\!\!\rightarrow F_2$

We say that F_1 has *weak full expressiveness w.r.t.* F_2, denoted $F_2 \lhd F_1$ if,

$$\forall n \in \mathbb{N}, \ \forall f \in \mathbf{G}_2^n, \ \exists \tilde{f} \in \mathbf{G}_1[Z_1, \dots, Z_n] : \forall C_1^i, \dots, C_n^i \in \mathbf{A}_i,$$

$$\bigwedge_{k=1}^n \sigma_1(C_k^1) \simeq \sigma_2(C_k^2) \implies \sigma_1(\tilde{f}[C_1^1/Z_1, \dots, C_n^1/Z_n]) \simeq \sigma_2(f\langle C_1^2, \dots, C_n^2\rangle), \quad (3.8)$$

where $\tilde{f}[C_1^1/Z_1, \dots, C_n^1/Z_n] \in \mathbf{A}_1$ is the component obtained by substituting in $\tilde{f}$ the variables Z_k by components C_k^1, for all $k \in [1, n]$.

Example 3.1.8. In addition to CCS^- from Example 3.1.2, consider the framework $CCS = (\mathbf{G}, \mathbf{C}, \mathbf{B}, \simeq, \sigma)$, where $\mathbf{C}$, $\mathbf{B}$ and $\simeq$ are the same as in CCS^-, whereas $\mathbf{G} = \mathbf{G}^- \cup \{+\}$, with the extension from σ^- to σ being trivial: $\sigma(C_1 + C_2) \stackrel{def}{=} \sigma(C_1) + \sigma(C_2)$, i.e. $\hat{+} = +$. We trivially have $CCS^- \lhd CCS$.

It is easy to show that the + operator cannot be encoded by any combination of parallel composition and restriction, essentially because the choice represented by + has to be maintained throughout the subsequent execution of the process, whereas both parallel composition and restriction are "memoryless". Hence, $CCS \not\lhd CCS^-$. ◇

As mentioned above, both strong and weak full expressiveness are preorders, i.e. they are reflexive and transitive. Strong full expressiveness trivially implies the weak one.

Definition 3.1.9. Based on the full expressiveness relations, we introduce six comparison relations, presented in Table 3.1. For instance (second cell of the first row), if F_1 has strong full expressiveness w.r.t. F_2, whereas F_2 has only weak—but not strong—full expressiveness w.r.t. F_1, we say that F_1 *is weakly more expressive than* F_2 (alternatively, F_2 *is weakly less expressive than* F_1) and denote this by $F_2 \to F_1$.

If F_2 does not even have weak full expressiveness w.r.t. F_1 (third cell of the first row), we say that F_1 *is strongly more expressive than* F_2, denoted $F_2 \Rightarrow F_1$, etc.

Example 3.1.10. The full expressiveness relations from Example 3.1.8 mean that CCS is strongly more expressive than CCS^-, i.e. $CCS^- \Rightarrow CCS$. ◇

The intuitive meanings of the three relations in the first row of Table 3.1 are the following: given two frameworks $F_i = (\mathbf{G}_i, \mathbf{C}_i, \mathbf{B}, \simeq, \sigma_i)$, if for any operator in $\mathbf{G}_2$ we can find a corresponding operator in $\mathbf{G}_1$ such that its application to an equivalent set of components would result in

equivalent components, then F_1 has at least equivalent expressiveness or is more expressive than F_2. There are three options for the converse. If there is a corresponding operator in $\mathbf{G}_2$ for any operator in $\mathbf{G}_1$, then their expressiveness are equivalent. If every operator in $\mathbf{G}_1$, which does not have a corresponding one in $\mathbf{G}_2$, can be represented by a composition of operators in $\mathbf{G}_2$, then F_1 is weakly more expressive than F_2. Finally, if there exists an operator in $\mathbf{G}_1$ that cannot be represented by any combination of operators in $\mathbf{G}_2$, then F_1 is strongly more expressive than F_2. If such inexpressible operators exists in both $\mathbf{G}_1$ and $\mathbf{G}_2$ then F_1 and F_2 are incomparable (third row of Table 3.1). The intuition behind the relations in the second row of Table 3.1 is similar.

Notice that the relations shown in Table 3.1 are mutually exclusive. For instance, contrary to the usual intuition behind the use of the symbols '$\Leftrightarrow$' and '$\Rightarrow$' in predicate logics, $F_1 \Leftrightarrow F_2$ implies $F_1 \nRightarrow F_2$. In particular, a framework is never more expressive than itself, i.e. $F \nRightarrow F$ and $F \nrightarrow F$.

Intuitively, one should read the symbols in Table 3.1 as follows: $F_1 \Leftrightarrow F_2$ means that there is a *strong correspondence* between F_1 and F_2, $F_2 \Rightarrow F_1$ means that going from F_2 to F_1 makes a *big difference in expressiveness* etc.

3.1.4 Properties of the comparison relations

Let me now provide some key properties of the relations defined in the previous subsection. Here, I will only summarise the key results necessary for the subsequent sections. Full details are provided in [BB20].

Proposition 3.1.11. *The relations $\Rightarrow$, $\rightarrow$ and $\Leftrightarrow$ are transitive.*

In particular, transitivity of $\Leftrightarrow$ implies that this relation is, indeed, an equivalence. Furthermore, all the relations introduced above are preserved by $\Leftrightarrow$.

Proposition 3.1.12. *For any frameworks F_1, F_2, F_3, such that $F_1 \Leftrightarrow F_2$ and any $\mathcal{R} \in \{\Leftrightarrow, \rightarrow, \Rightarrow, \leftrightarrow, \dashrightarrow, \longleftrightarrow \}$, we have $F_1 \mathcal{R} F_3$ if $F_2 \mathcal{R} F_3$.*

The third proposition is consistent with the intuitive interpretation of the symbols '$\Rightarrow$' and '$\rightarrow$' given at the end of the previous subsection: if going from one of F_1 and F_2 to the next framework (resp. F_2 or F_3) makes a big difference in expressiveness, then there also is a big difference between F_1 and F_3.

Proposition 3.1.13. *For any frameworks F_1, F_2, F_3, hold the following implications:*

1. *$F_1 \Rightarrow F_2 \rightarrow F_3$ implies $F_1 \Rightarrow F_3$,*

2. *$F_1 \rightarrow F_2 \Rightarrow F_3$ implies $F_1 \Rightarrow F_3$.*

Notice that relation combinations other than those in the three propositions above do not provide immediate "shortcut" relations. This is mostly due to the fact that the complement relations $\nrightarrow$ and $\nRightarrow$ are not transitive. For instance, given F_1, F_2 and F_3, such that $F_1 \lhd F_2 \lhd F_3$ but $F_1 \nleftarrow F_2 \nleftarrow F_3$, we can proceed as follows:

1. Take all composition operators from F_1 that do not have corresponding ones in F_2 and add them to F_3 as "syntactic sugar" for the corresponding expressions, which necessarily exist since $F_1 \lhd F_3$.

2. Add two *completely new* operators two F_3 and another completely new operator to F_2 with the semantics defined as the composition of those for the two new operators in F_3.

Denoting the extended frameworks by F_2' and F_3', we have $F_1 \lhd F_2' \lhd F_3'$, $F_1 \nleftarrow F_2' \nleftarrow F_3'$ but $F_1 \lhd F_3'$. Without constituting a formal proof, this manipulation does provide an intuition for the reason why other relation combinations do not have generic shortcuts.

As mentioned above, strong full expressiveness trivially implies weak full expressiveness. The converse holds in presence of uniform flattening.

Proposition 3.1.14. *For two frameworks F_1 and F_2, such that F_1 has uniform flattening, $F_2 \vartriangleleft F_1$ implies $F_2 \blacktriangleleft F_1$.*

Sketch of the proof. Weak full expressiveness guaranties that any operator $f \in \mathbf{G}_2$ is expressible as a composition of operators in $\mathbf{G}_1$. Uniform flattening applied several times to such a compound expression can "flatten" it to a single operator corresponding to f. Thus, the requirement for strong full expressiveness is satisfied. $\qquad\square$

Proposition 3.1.14 simplifies the expressiveness comparison. For instance, if both F_1 and F_2 have uniform flattening, the $\vartriangleleft$ and $\blacktriangleleft$ relations coincide. This eliminates the second column and row in Table 3.1, leaving only three possibilities: either the two frameworks are strongly equivalent, or one is strongly more expressive than the other, or they are incomparable.

According to Definition 3.1.7, in order to establish that one framework has strong (resp. weak) full expressiveness w.r.t. another, we have to prove the existence of the corresponding operator (resp. compound expression) that preserves the semantic equivalence (see (3.7) and (3.8)). Below, we show that, under additional assumptions, it is sufficient to only check the preservation of the *behaviour equality*.

Theorem 3.1.15. *For two frameworks $F_i = (\mathbf{G}_i, \mathbf{C}, \mathbf{B}, \simeq, \sigma_i)_{i \in \{1,2\}}$, whereof F_1 is compositional, with the same atomic components, the same semantic domain, and such that, for any $C \in \mathbf{C}$, holds $\sigma_1(C) = \sigma_2(C)$, we have*

1. $F_2 \blacktriangleleft F_1$ if

$$\forall n \in \mathbb{N}, \ \forall f \in \mathbf{G}_2^n, \ \exists \tilde{f} \in \mathbf{G}_1^n : \ \forall C_1^i, \dots, C_n^i \in \mathbf{A}_i,$$
$$\bigwedge_{k=1}^{n} \sigma_1(C_k^1) = \sigma_2(C_k^2) \implies \sigma_1\big(\tilde{f}\langle C_1^1, \dots, C_n^1\rangle\big) = \sigma_2\big(f\langle C_1^2, \dots, C_n^2\rangle\big), \quad (3.9)$$

2. $F_2 \vartriangleleft F_1$ if

$$\forall n \in \mathbb{N}, \ \forall f \in \mathbf{G}_2^n, \ \exists \tilde{f} \in \mathbf{G}_1[Z_1, \dots, Z_n] : \forall C_1^i, \dots, C_n^i \in \mathbf{A}_i,$$
$$\bigwedge_{k=1}^{n} \sigma_1(C_k^1) = \sigma_2(C_k^2) \implies \sigma_1\big(\tilde{f}[C_1^1/Z_1, \dots, C_n^1/Z_n]\big) = \sigma_2\big(f\langle C_1^2, \dots, C_n^2\rangle\big), \quad (3.10)$$

where all notations are as in Definition 3.1.7.

Proof. We prove the proposition for $F_2 \blacktriangleleft F_1$—the proof for $F_2 \vartriangleleft F_1$ is similar.

Let us denote $\mathbf{A}_i^m$ the set of components $C \in \mathbf{A}_i$, such that the maximal chain of applications of composition operators in the construction of C has the length m. In particular, $\mathbf{A}_i^0 = \mathbf{C}$.

The proof is by induction on the structural depth of the components involved. The induction hypothesis is the following: *with the restriction of the last quantification to $\forall C_1^i, \dots, C_n^i \in \mathbf{A}_i^m$, (3.9) implies (3.7) and*

$$\forall C \in \mathbf{A}_2^m, \exists C' \in \mathbf{A}_1^m : \sigma_1(C') = \sigma_2(C). \quad (3.11)$$

The induction step will consist in proving that if this statement holds for all $m' < m$, then it also holds for m. The base case is $m = 0$, i.e. the quantification is over atomic components only.

Consider a pair of operators f and $\tilde{f}$, which satisfy (3.9) with the last quantification being replaced by $\forall C_1^i, \dots, C_n^i \in \mathbf{C}$, and two sets of atomic components $C_1^i, \dots, C_n^i \in \mathbf{C}$ (for $i = 1, 2$), such that $\sigma_1(C_k^1) \simeq \sigma_2(C_k^2)$, for all $k \in [1, n]$. Since, by the assumption of the proposition, $\sigma_1(C_k^2) = \sigma_2(C_k^2)$, we have, by applying (3.9) to $C_1^2, \dots, C_n^2$ *only* (rather than to $C_1^1, \dots, C_n^1$ and $C_1^2, \dots, C_n^2$),

$$\sigma_1\big(\tilde f\langle C_1^2,\ldots,C_n^2\rangle\big) = \sigma_2\big(f\langle C_1^2,\ldots,C_n^2\rangle\big).$$

Since $\sigma_1(C_k^1) \simeq \sigma_2(C_k^2) = \sigma_1(C_k^2)$, by compositionality of F_1, we have

$$\sigma_1\big(\tilde f\langle C_1^1,\ldots,C_n^1\rangle\big) \simeq \sigma_1\big(\tilde f\langle C_1^2,\ldots,C_n^2\rangle\big)$$

and, combining the two,

$$\sigma_1\big(\tilde f\langle C_1^1,\ldots,C_n^1\rangle\big) \simeq \sigma_2\big(f\langle C_1^2,\ldots,C_n^2\rangle\big).$$

Notice that, for $m = 0$, (3.11) holds trivially by the assumption of the proposition, taking $C' = C$.

Let us now prove the induction step. First of all, consider a component $C \in \mathbf{A}_2^m$. Since $m > 0$, we have $C = f\langle C_1,\ldots,C_l\rangle$, for some $f \in \mathbf{G}_2$ and $C_1,\ldots,C_l \in \mathbf{A}_2^{m-1}$. Hence, by the induction hypothesis, there exist $C_1',\ldots,C_l' \in \mathbf{A}_1^{m-1}$, such that $\sigma_1(C_k') = \sigma_2(C_k)$, for all $k \in [1,l]$. By (3.9), there exists $\tilde f \in \mathbf{G}_1$, such that $\sigma_1\big(\tilde f\langle C_1',\ldots,C_l'\rangle\big) = \sigma_2\big(f\langle C_1,\ldots,C_l\rangle\big) = \sigma_2(C)$. Denoting $C' = \tilde f\langle C_1',\ldots,C_l'\rangle \in \mathbf{A}_1^m$, we obtain the proof of the induction step for (3.11).

Consider now a pair of operators f and $\tilde f$, which satisfy (3.9) with the last quantification replaced by $\forall C_1^i,\ldots,C_n^i \in \mathbf{A}_i^m$. Consider, furthermore, two sets of components $C_1^i,\ldots,C_n^i \in \mathbf{A}_i^m$, such that $\sigma_1(C_k^1) \simeq \sigma_2(C_k^2)$, for all $k \in [1,n]$. By (3.11) (as proven above), there exist $C_1^{2\prime},\ldots,C_n^{2\prime} \in \mathbf{A}_1^m$, such that $\sigma_1(C_k^{2\prime}) = \sigma_2(C_k^2)$, for all $k \in [1,n]$. By (3.9), we have

$$\sigma_1\big(\tilde f\langle C_1^{2\prime},\ldots,C_n^{2\prime}\rangle\big) = \sigma_2\big(f\langle C_1^2,\ldots,C_n^2\rangle\big).$$

Since $\sigma_1(C_k^1) \simeq \sigma_2(C_k^2) = \sigma_1(C_k^{2\prime})$, by compositionality of F_1, we have

$$\sigma_1\big(\tilde f\langle C_1^1,\ldots,C_n^1\rangle\big) \simeq \sigma_1\big(\tilde f\langle C_1^{2\prime},\ldots,C_n^{2\prime}\rangle\big)$$

and, combining the two,

$$\sigma_1\big(\tilde f\langle C_1^1,\ldots,C_n^1\rangle\big) \simeq \sigma_2\big(f\langle C_1^2,\ldots,C_n^2\rangle\big),$$

which proves the induction step for (3.7) and thereby concludes the proof of the proposition. $\qquad\square$

All the frameworks mentioned in this chapter have structural semantics and follow SOS formats that preserve bisimilarity. Since we will consider a bisimilarity-based equivalence relation on the behaviour type, all these frameworks are compositional by Proposition 3.1.5. Furthermore, all these frameworks have the same set of atomic components and, up to a canonical extension (see Section 3.3), the same semantic domain. Hence, they satisfy all the assumptions of Theorem 3.1.15, which means that we can prove the positive results about their relative expressiveness by studying the defining mappings of the matching composition operators and showing that their application preserves the equality of behaviours. Negative results are proven by counterexamples.

3.2 Expressiveness of BIP

In this section, we present key results obtained by studying the expressiveness of the BIP framework and comparing it to some of its variations.

3.2.1 BIP-like SOS

Recall from Chapter 2, Rule (2.1) that, in BIP, the semantics of composition with an interaction model γ is defined by the SOS rule

$$\frac{a \in \gamma \quad \left\{q_i \xrightarrow{a \cap P_i} q_i' \,\middle|\, i \in I\right\} \quad \left\{q_i = q_i' \,\middle|\, i \notin I\right\}}{q_1 \ldots q_n \xrightarrow{a}_\gamma q_1' \ldots q_n'},$$

where $I = \{i \in [1, n] \mid a \cap P_i \neq \varnothing\}$. We can "normalise" this rule, transforming it into *a set of rules* whereof premises refer only to state predicates of the forms $q \xrightarrow{a} q'$ and $q = q'$:

$$\left\{ \left. \frac{\left\{q_i \xrightarrow{a \cap P_i} q'_i \mid i \in I^a\right\} \quad \left\{q_i = q'_i \mid i \notin I^a\right\}}{q_1 \dots q_n \xrightarrow{a}_\gamma q'_1 \dots q'_n} \right| a \in \gamma \right\}, \tag{3.12}$$

with the notation $I^a \overset{def}{=} \{i \in [1, n] \mid a \cap P_i \neq \varnothing\}$.

The premises of the form $q = q'$ are there for notational convenience. The premises of the form $q \xrightarrow{a} q'$, however, are essential for the definition of the composition semantics. They are called *positive* because they specify what the components have to *be able to do* in the current state in order to enable the corresponding action in the composed system.[1]

Trivially, any composition operator that can be defined by a set of rules with positive premises of this form, can also be specified as a BIP interaction model. In this section, we will focus instead on composition operators that can be expressed using *negative* premises of the form $q \xnrightarrow{a} q'$, which specify what the components must *not be able to do*.

Indeed, every BIP glue operator is a combination of a (possibly trivial) interaction model with a (possibly trivial) priority model. After some simplifications (see e.g. [BB20]), merging rules of forms (2.3) and (2.1), the semantics of any given BIP operator can be defined by a set of SOS rules of the form

$$\left\{ \left. \frac{\left\{q_i \xrightarrow{a \cap P_i} q'_i \mid i \in I^a\right\} \quad \left\{q_i = q'_i \mid i \notin I^a\right\} \quad \left\{q_j \xnrightarrow{b} \mid (j, b) \in H\right\}}{q_1 \dots q_n \xrightarrow{a} q'_1 \dots q'_n} \right| \Phi(a, H) \right\}, \tag{3.13}$$

where, $\Phi(a, H)$ is some predicate imposing constraints on $a \subseteq P$ and $H \subseteq [1, n] \times 2^P$ such that $\forall (j, b) \in H,\ b \in 2^{P_j}$. As above $I^a \overset{def}{=} \{i \in [1, n] \mid a \cap P_i \neq \varnothing\}$. For instance, in (3.12), we put $\Phi(a, H) \overset{def}{=} a \in \gamma \land H = \varnothing$.

Notice that several distinct sets H can satisfy $\Phi(a, H)$ with the same a, i.e. there can be several rules in the set (3.13) with the same label in the conclusion.

Notice also that, for each component, there is at most one positive but possibly many negative premises in the same rule. The reason for that is that all positive premises must contribute to the transition in the conclusion of the rule and there is at most one transition that any given component can take. There is, of course, no such restriction on the negative premises.

Below we call the format (3.13) *BIP-like SOS*.

3.2.2 The problem of flattening

Before proceeding to the results about the expressiveness of BIP, let me first present an example explaining that there is an object to study in the first place.

Recall (Section 2.1.1) that application of a priority model does not introduce deadlocks.

Example 3.2.1 ([BB15]). Consider the compound component $f\langle g\langle C_1, C_2\rangle, C_3\rangle$ (Figure 3.1a), with the glue operator g defined by the interaction model $\gamma_1 = \{p, q, r, s\}$ and priority model $\pi_1 = \{p \prec r\}$; f defined by the interaction model $\gamma_2 = \{p, q, s, rt\}$ and the empty priority model. The LTS of the compound behaviour is shown in Figure 3.1b with the transitions, suppressed as the result of applying priority in g, shown as dashed arrows. Composing the rules corresponding

---

[1]Technically speaking, this statement is not precise: these premises are called positive because they refer to the positive form of the predicate $\rightarrow$ and not to its negation $\nrightarrow$. However, in our context, this definition appears to be tautological and not really useful.
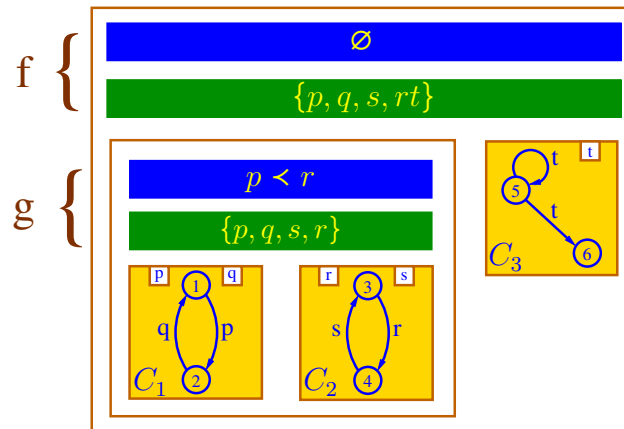
(a) Compound component
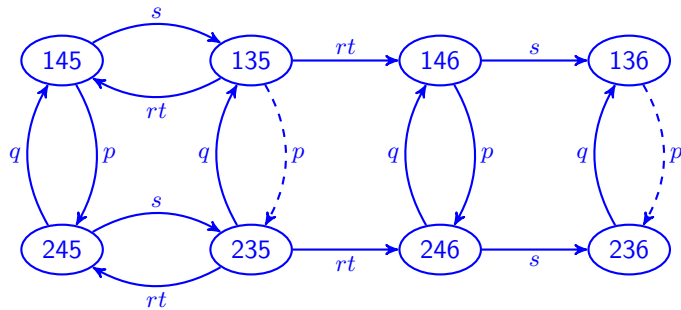
(b) Compound LTS (transitions suppressed by the priority model are shown by dashed arrows)

Figure 3.1: BIP component that cannot be flattened (Example 3.2.1)

to these operators, we obtain the four rules

$$\frac{q_1 \xrightarrow{p} q_1' \quad q_2 \xmapsto{r}}{q_1q_2q_3 \xrightarrow{p} q_1'q_2q_3}, \qquad \frac{q_1 \xrightarrow{q} q_1'}{q_1q_2q_3 \xrightarrow{q} q_1'q_2q_3}, \qquad \frac{q_2 \xrightarrow{s} q_2'}{q_1q_2q_3 \xrightarrow{s} q_1q_2'q_3}, \qquad \frac{q_2 \xrightarrow{r} q_2' \quad q_3 \xrightarrow{t} q_3'}{q_1q_2q_3 \xrightarrow{rt} q_1q_2'q_3'}.$$

$$(3.14)$$

Assume that an interaction model γ and a priority model π are such that $\pi\gamma\langle C_1, C_2, C_3\rangle$ is equivalent to $f\langle g\langle C_1, C_2\rangle, C_3\rangle$. By the first rule in (3.14), the transition $14x \xrightarrow{p} 24x$ is possible in $f\langle g\langle C_1, C_2\rangle, C_3\rangle$, for any $x \in \{5, 6\}$. Hence, $p \in \gamma$. Clearly, 136 is a deadlock state in $f\langle g\langle C_1, C_2\rangle, C_3\rangle$. Hence, 136 must be a deadlock state in $\pi\gamma\langle C_1, C_2, C_3\rangle$ and, by Lemma 2.1.9, also in $\gamma\langle C_1, C_2, C_3\rangle$, which is not possible, since the only premise of the rule

$$\frac{q_1 \xrightarrow{p} q_1'}{q_1q_2q_3 \xrightarrow{p} q_1'q_2q_3},$$

corresponding to p in the semantics (2.1) of γ, is satisfied for $q_1 = 1$ and $q_1' = 2$. $\diamondsuit$

Example 3.2.1 shows that the classical semantics of BIP does not possess flattening. Moreover, since the rules defining the semantics of the composition of the operators f and g can, indeed, be flattened (3.14), classical BIP does not possess strong full expressiveness w.r.t. BIP-like SOS either. Thus, there is difference in the expressiveness of the classical BIP and the framework that would allow any composition operator definable using BIP-like SOS. In particular, this means that the often encountered informal statement: "BIP possesses the expressiveness of the universal glue" (or its equivalent in slightly different formulations) is false. Indeed, it is based on an erroneous proposition in our previous work [BS08a, Proposition 4]. The objective of this expressiveness study was to provide a correction to that erroneous proposition and to characterise the coordination mechanisms lying on the spectrum between the expressiveness of the classical BIP and that of BIP-like SOS.

3.2.3 The semantic domain

We will consider component based frameworks of the form $(\mathbf{G}, \mathbf{C}, \mathbf{B}, \simeq, \sigma)$, with the same semantic domain $(\mathbf{B}, \simeq)$ and set of atomic components $\mathbf{C}$.

We take $\mathbf{B}$ to be the set of LTSs as defined in Chapter 2 (Definition 2.1.1). The equivalence of LTS is defined using the notion of bisimulation [Par81].

Definition 3.2.2. Let $B_1 = (Q_1, P_1, \rightarrow_1, q_1^0)$ and $B_2 = (Q_2, P_2, \rightarrow_2, q_2^0)$ be two LTS, and let $R \subseteq Q_1 \times Q_2$ be a binary relation.

- R is a *simulation* if, for all $q_1 R q_2$, $q_1 \xrightarrow{a}_1 q_1'$ implies $q_2 \xrightarrow{a}_2 q_2'$ for some $q_2' \in Q_2$ such that $q_1' R q_2'$.

- R is a *bisimulation* if both R and R^{-1} are simulations.

We say that B_1 and B_2 are *bisimilar* if there exists a bisimulation relation R relating their initial states, i.e. such that $q_1^0 R q_2^0$.[2]

We focus on bisimilarity due to two of its properties that are particularly important in our context:

---

[2]In previous work, we have mostly refrained from explicitly specifying initial states, implicitly considering that all states are initial. In such case, this requirement boils down to the bisimulation relation being total on both Q_1 and Q_2.

1. Bisimilar behaviours satisfy exactly the same CTL properties (see e.g. [BK08, Theorem 7.20]).

2. Bisimilarity is a congruence for any composition operator defined by SOS rules in a "reasonable" format [Ver95].

With respect to the second item above, I will not delve into the precise meaning of the word "reasonable" as I use it here—this is not the term used by Verhoef [Ver95]—but only say that the key condition is that the SOS format be *stratifiable*. In the case of BIP-like SOS, this stratifiability is obtained trivially, since any transition relation can appear in at most one level of a derivation tree: the conclusion of a rule always refers to a transition relation of a compound component completely encapsulating all of its sub-components.

Definition 3.2.3. The equivalence relation $\simeq \subseteq \mathbf{B} \times \mathbf{B}$ is defined by putting, for two behaviours B_1 and B_2 as in the previous definition, $B_1 \simeq B_2$ if $P_1 = P_2$ and the two LTS are bisimilar.

As in Chapter 2, we take the atomic components to be those defined directly as LTSs:

$$\mathbf{C} \stackrel{def}{=} \left\{ (P, B) \,\middle|\, B = (Q, P, \rightarrow, q^0) \right\}.$$

The semantics of atomic components is given by their behaviour: $\sigma(P, B) \stackrel{def}{=} B$.

All the semantic mappings σ will be compositional, i.e. defined by associating to each composition operator a corresponding defining mapping as discussed in Section 3.1. Thus, we can say that the component-based frameworks considered in the next section will only differ in their sets of glue operators $\mathbf{G}$.

3.2.4   From BIP to BIP-like SOS

The example in Section 3.2.2 shows that there is difference in the expressiveness of the classical BIP and the framework that would allow any composition operator definable using BIP-like SOS. Let us now discuss the details of the expressiveness spectrum that lies in-between.

We denote by $\mathbf{CBIP} = (\mathbf{G_{CBIP}}, \mathbf{C}, \mathbf{B}, \simeq, \sigma)$ (Classical BIP) the framework with $\mathbf{C}$, $\mathbf{B}$ and $\simeq$ defined as in the previous section, and the set $\mathbf{G_{CBIP}}$ comprising all BIP composition operators as defined in Definition 2.1.7 with their corresponding semantics defined by the rules (2.1) and (2.3).

Similarly, we will denote by $\mathbf{BSOS} = (\mathbf{G_{BSOS}}, \mathbf{C}, \mathbf{B}, \simeq, \sigma)$ the framework with $\mathbf{G_{BSOS}}$ comprising all *BIP-like SOS glue operators*: the composition operators defined as $\left((P_i)_{i=1}^n, \mathcal{R} \right)$, where $(P_i)_{i=1}^n$ are pair-wise disjoint sets of ports and $\mathcal{R}$ is a set of BIP-like SOS rules (3.13) inductively defining its semantics.

Besides $\mathbf{CBIP}$ and $\mathbf{BSOS}$, we will define four additional frameworks: $\mathbf{RBIP}$ and $\mathbf{XBIP}$ will extend $\mathbf{CBIP}$, while $\mathbf{AcBSOS}$ and $\mathbf{SiBSOS}$ will restrict $\mathbf{BSOS}$. Figure 3.2 shows the comparison of expressiveness among these six frameworks. This expressiveness hierarchy is extended with additional frameworks in [BB20].

$\mathbf{RBIP}$ (Relaxed BIP) extends $\mathbf{CBIP}$ by allowing priority models to be arbitrary relations on sets of ports, i.e. $\pi \subseteq 2^P \times 2^P$, without requiring that these be strong partial orders nor limiting them to interactions provided by the components.

$\mathbf{XBIP}$ (Complex BIP) is a further extension of $\mathbf{RBIP}$, which allows sets of interactions—as opposed to single interactions in $\mathbf{CBIP}$ and $\mathbf{RBIP}$—to be used as inhibitors in the priority model. For instance, in $\mathbf{RBIP}$, we can have the priorities $a \prec b$ and $a \prec c$—meaning that the interaction a is inhibited whenever *at least one* of the interactions b and c is enabled—or $a \prec bc$—meaning that a is inhibited whenever the *combined* interaction bc is enabled. In addition, in $\mathbf{XBIP}$, we can have a differnt kind of priority, $a \prec \{b, c\}$, meaning that a is inhibited when *both* interactions b and c are enabled. *Depending on the port partition $(P_i)_{i=1}^n$, this may or may not be the same as $a \prec bc$.*
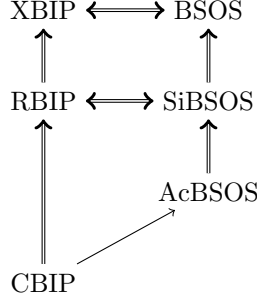
Figure 3.2: Expressiveness relations among the considered frameworks

Table 3.2: Examples of inhibiting relations

Partition of ports	SOS rules	Inhibiting relation
$a \subseteq P_1,$ $b, c \subseteq P_2$	$\dfrac{q_1 \xrightarrow{a} q_1' \quad q_2 \xnrightarrow{b} \quad q_2 \xnrightarrow{c}}{q_1 q_2 \xrightarrow{a} q_1' q_2}$	$\pi = \{(a, \{b\}), (a, \{c\})\}$
$a \subseteq P_1,$ $b, c \subseteq P_2$	$\dfrac{q_1 \xrightarrow{a} q_1' \quad q_2 \xnrightarrow{bc}}{q_1 q_2 \xrightarrow{a} q_1' q_2}$	$\pi = \{(a, \{bc\})\}$
$a \subseteq P_1,$ $b, c \subseteq P_2$	$\dfrac{q_1 \xrightarrow{a} q_1' \quad q_2 \xnrightarrow{b}}{q_1 q_2 \xrightarrow{a} q_1' q_2}, \qquad \dfrac{q_1 \xrightarrow{a} q_1' \quad q_2 \xnrightarrow{c}}{q_1 q_2 \xrightarrow{a} q_1' q_2}$	$\pi = \{(a, \{b, c\})\}$
$a \subseteq P_1, b \subseteq P_2,$ $c \subseteq P_3$	$\dfrac{q_1 \xrightarrow{a} q_1' \quad q_2 \xnrightarrow{b}}{q_1 q_2 q_3 \xrightarrow{a} q_1' q_2 q_3}, \qquad \dfrac{q_1 \xrightarrow{a} q_1' \quad q_3 \xnrightarrow{c}}{q_1 q_2 q_3 \xrightarrow{a} q_1' q_2 q_3}$	$\pi = \{(a, \{bc\})\}$

(Notice that the last premise in the fourth example refers to q_3 as opposed to q_2 in the third example.)

It is relatively straightforward to define the semantics of the priority models in **RBIP** and **XBIP** using BIP-like SOS rules (see [BB20] for details).

In order to determine whether a given set of BIP-like SOS operators can be expressed as a combination of an interaction and a priority model in one of the **CBIP**, **RBIP** and **XBIP** frameworks, we have defined a notion of the *inhibiting relation* $\pi \subseteq 2^P \times 2^{2^P}$ [BB20]. Intuitively, the set $\pi(a)$ comprises all possible sets of interactions, formed by combining negative premises from all the rules with conclusion a, which would inhibit a if enabled simultaneously.

Table 3.2 shows four examples of sets of rules with their corresponding inhibiting relations. In the first example, $\pi(a) = \{\{b\}, \{c\}\}$ contains two singleton sets comprising interactions b and c, respectively. In the second and fourth examples, the set $\pi(a) = \{\{bc\}\}$ contains one singleton set comprising the interaction bc. Finally, in the third example, $\pi(a) = \{\{b, c\}\}$ contains one set comprising two interactions b and c.

Definition 3.2.4. If, for each a in the domain of π, all sets in $\pi(a)$ are singleton, we say that the inhibiting relation π is *simple*.

In Table 3.2, the first, second and fourth examples have simple inhibiting relations, whereas that of the third example is *not* simple.

A *simple BIP-like SOS operator* is a composition operator defined as $((P_i)_{i=1}^n, \mathcal{R})$, where $(P_i)_{i=1}^n$ are disjoint sets of ports and $\mathcal{R}$ is a set of BIP-like SOS rules (3.13) with a simple inhibiting relation. **SiBSOS** (Simple BSOS) is a restriction of **BSOS**, with $\mathbf{G_{SiBSOS}}$ comprising only simple BIP-like SOS operators.

Since all sets in the codomain of a simple inhibiting relation are singleton, they can be replaced

41

by their elements without loss of information, e.g. replacing $(a, \{b\})$ by (a, b). This implies that a simple inhibiting relation $\pi \subseteq 2^P \times 2^{2^P}$ can be equivalently considered as a relation on 2^P (i.e. $\pi \subseteq 2^P \times 2^P$).

Definition 3.2.5. We say that a simple inhibiting relation is *acyclic* if it does not have any cycles when considered as a relation on 2^P.

An *acyclic BIP-like SOS operator* is a composition operator defined as $\left((P_i)_{i=1}^n, \mathcal{R}\right)$, where $(P_i)_{i=1}^n$ are disjoint sets of ports and $\mathcal{R}$ is a set of BIP-like SOS rules (3.13) with a simple and acyclic inhibiting relation. **AcBSOS** (Acyclic BSOS) is a restriction of **SiBSOS**, with $\mathbf{G_{AcBSOS}}$ comprising only acyclic BIP-like SOS operators.

Notice that, for any **CBIP** operator $\left((P_i)_{i=1}^n, \gamma, \pi\right)$, the inhibiting relation of the set of BIP-like SOS rules defining its semantics coincides with π and, therefore, is simple and acyclic. Although the example in Section 3.2.2 shows that there exist **AcBSOS** operators that cannot be expressed as one **CBIP** operator, we have shown [BB16; BB20] that all **AcBSOS** operarators can be expressed as a composition of several **CBIP** ones. Hence the weakly more expressive relation **CBIP → AcBSOS** in Figure 3.2.

3.3 The offer predicate

The example of Section 3.2.2 shows a composition of two operators that cannot be flattened within the constraints of the classical BIP framework. The stated reason for this impossibility result is that, if such a BIP operator were to exist, the corresponding priority model would have to introduce a deadlock, violating the property proven in Lemma 2.1.9.

Let us now consider that same example from a different perspective: *what is it that prevents a priority model from inhibiting the interaction p?* Inspecting the rules (3.14) it is straightforward to conclude that *p should have been inhibited* by r but this latter cannot be part of the interaction model—indeed, it must be synchronised with t to form the interaction rt. In other words, information that can be used by the priority model comprises only the interactions authorised by the underlying interaction model. All the information about transitions enabled in sub-components is lost in the compound component.

3.3.1 Extension of the semantic domain

An alternative approach, consists in extending the definition of the component behaviour to integrate part of this information about the active transitions of its subcomponents. To this end, we extend the notion of behaviour with an additional *offer* predicate [BS11; BB15]. The notable difference with the approach of the previous section is that information is made available about the active transitions of the atomic subcomponents situated at the lowest levels of the hierarchy instead of the one immediately underneath the considered one.

Definition 3.3.1. An *extended behaviour* is a tuple $B = (Q, P, \rightarrow, \uparrow, q^0)$, where $(Q, P, \rightarrow, q^0)$ is an LTS and $\uparrow$ is an *offer* predicate on $Q \times P$, such that $q \uparrow p$ holds (the port $p \in P$ is *offered* in the state $q \in Q$) whenever there is a transition from q containing p, that is holds the following proposition:

$$\forall q \in Q, \ \forall p \in P, \ (\exists a \in 2^P : p \in a \wedge q \xrightarrow{a}) \implies q \uparrow p. \tag{3.15}$$

The offer predicate extends to sets of ports: for $a \in 2^P$, $q \uparrow a \stackrel{\text{def}}{=} \bigwedge_{p \in a} q \uparrow p$. Notice that $q \uparrow \varnothing \equiv true$. We denote $q \not\uparrow a \stackrel{\text{def}}{=} \neg(q \uparrow a) = \bigvee_{p \in a} q \not\uparrow p$.

Behaviour equivalence from Definition 3.2.3 is extended canonically as follows.

Definition 3.3.2. Two extended behaviours $B_i = (Q_i, P_i, \to_i, \uparrow_i, q_i^0)$, with $i = 1, 2$, are *equivalent*—denoted $B_1 \simeq B_2$, by abuse of notation—if $P_1 = P_2$ and there exists a bisimulation relation $R \subseteq Q_1 \times Q_2$, such that $q_1^0 R q_2^0$ and, for all $(q_1, q_2) \in R$ and $p \in P_1$, holds $q_1 \uparrow_1 p \Leftrightarrow q_2 \uparrow_2 p$.

All six frameworks considered in Section 3.2.4 share the same semantic domain and can be extended to corresponding frameworks using the semantics domain with the offer predicate. Let $F = (\mathbf{G}, \mathbf{C}, \mathbf{B}, \simeq, \sigma)$ be one of the six frameworks considered in Section 3.2.4. The corresponding extended version $F' = (\mathbf{G}, \mathbf{C}, \mathbf{B}', \simeq', \sigma')$, has $\mathbf{B}'$ the set of extended behaviours as in Definition 3.3.1, $\simeq'$ the equivalence from Definition 3.3.2 and the semantic mapping σ' is defined by putting $\sigma'(C) \overset{def}{=} (Q, P, \to, \uparrow, q^0)$, with $(Q, P, \to, q^0) = \sigma(C)$ and

$$q \uparrow p \overset{def}{\Longleftrightarrow} \begin{cases} \exists a \in 2^P : p \in a \land q \overset{a}{\to}, & \text{if } C \in \mathbf{C}, \\ \exists i \in [1, n] : q_i \uparrow p, & \text{if } C = f\langle C_1, \dots, C_n \rangle \text{ and } q = (q_1, \dots, q_n). \end{cases} \quad (3.16)$$

Notice that, for atomic components $C \in \mathbf{C}$, we put $q \uparrow p \Leftrightarrow \exists a \in 2^P : p \in a \land q \overset{a}{\to}$, as opposed to (3.15) in Definition 3.3.1, where the implication goes only one way. This distinguishes atomic components from the composite ones. More importantly, the predicate $\uparrow$ is defined by the same rule, for all composition operators.

Note 3.3.3. The above extension is clearly isomorphic w.r.t. component equivalence. In particular, this implies that it does not affect the expressiveness hierarchy in Figure 3.2.

Example 3.3.4. Consider again the compound component of Example 3.2.1. By redefining the semantics of priority models in terms of the offer predicate, the rules (3.14) become:

$$\frac{q_1 \overset{p}{\to} q_1' \quad q_2 \not\uparrow r}{q_1 q_2 q_3 \overset{p}{\to} q_1' q_2 q_3}, \qquad \frac{q_1 \overset{q}{\to} q_1'}{q_1 q_2 q_3 \overset{q}{\to} q_1' q_2 q_3}, \qquad \frac{q_2 \overset{s}{\to} q_2'}{q_1 q_2 q_3 \overset{s}{\to} q_1 q_2' q_3}, \qquad \frac{q_2 \overset{r}{\to} q_2' \quad q_3 \overset{t}{\to} q_3'}{q_1 q_2 q_3 \overset{rt}{\to} q_1 q_2' q_3'},$$

$$(3.17)$$

which, in this modified semantics, represent also the defining mapping of the *flat* composition operator obtained by combining the interaction model $\gamma = \{p, q, s, rt\}$ with the priority model $\pi = \{p \prec r\}$. $\qquad \diamond$

3.3.2 Firing-Negative-Activation SOS and Offer BIP

Generalising (3.17), we consider the following SOS format, proposed intially in [BS11]:

$$\left\{ \frac{\left\{q_i \overset{a \cap P_i}{\longrightarrow} q_i' \mid i \in I^a\right\} \quad \left\{q_i = q_i' \mid i \notin I^a\right\} \quad \left\{q_j \not\uparrow b \mid (j, b) \in H\right\}}{\left\{q_k \uparrow (c \cap P_k) \mid k \in [1, n]\right\}} \middle| \Phi(a, H, c) \right\}, \qquad (3.18)$$

where $\Phi(a, H, c)$ is some predicate imposing constraints on $a, c \subseteq P$ and $H \in [1, n] \times 2^P$, such that, for each $(j, b) \in H$ holds $b \in 2^{P_j}$. As above, $I^a \overset{def}{=} \{i \in [1, n] \mid a \cap P_i \neq \varnothing\}$.

Contrary to the BSOS format (3.13), there are three types of premises in (3.18) other than $q = q'$, respectively called *firing*, *negative* and *activation* premises. Firing and activation premises are collectively called *positive*. The key difference with BSOS (3.13) is that the negative premises are based on the offer predicate. Recall that $q \uparrow \varnothing = true$. Furthermore, $q \uparrow c_1 \land q \uparrow c_2 = q \uparrow (c_1 \cup c_2)$. Hence, one activation premise per component is sufficient to define any inference rule.

We denote by $\mathbf{FNASOS} = (\mathbf{G_{FNASOS}}, \mathbf{C}, \mathbf{B}', \simeq', \sigma')$ the framework with $\mathbf{G_{FNASOS}}$ comprising all *FNASOS glue operators*: the composition operators defined as $\left((P_i)_{i=1}^n, \mathcal{R}\right)$, where $(P_i)_{i=1}^n$ are pair-wise disjoint sets of ports and $\mathcal{R}$ is a set of FNASOS rules (3.18). The remaining elements are $\mathbf{C}$, $\mathbf{B}'$, $\simeq'$ defined as in Section 3.3.1 and σ' is defined inductively by $\mathcal{R}$ and (3.16).

43

Proposition 3.3.5. *FNASOS possesses uniform flattening.*

Sketch of the proof. The proof follows by taking the classical composition of SOS rules. Notice that, for any n-ary composition operator, the definition of the offer predicate can also be written as a set of SOS rules:

$$\left\{ \left. \frac{q_i \uparrow p}{q_1 \ldots q_n \uparrow p} \; \right| \; i \in [1, n] \right\} . \tag{3.19}$$

Thus, we only have to notice that this composition trivially preserves the format (3.18). Indeed, for firing premises, the rules in the same format are substituted directly. For, negative and activation premises, substituted rules are in format (3.19) and, therefore, negative premises are substituted by negative premises, activation premises—by activation premises. $\qquad \square$

Let us now redefine the BIP composition operators in the style of the rule format (3.18). The interaction models are exactly the same as in the classical BIP (see Section 2.1.1).

An *offer priority model* on a set of ports P is a relation $\pi \subseteq 2^P \times \left((2^P \cup \{\bot\}) \times 2^P \right)$.[3] Here, we introduce a fresh symbol $\bot \notin P$ and extend the offer predicate by putting $q \uparrow \bot \stackrel{def}{=} false$. We write $a \prec (b, c)$ as a shorthand for $(a, (b, c)) \in \pi$. The semantics of applying π to a component with the interface P is defined by the following set of rules:

$$\left\{ \left. \frac{q \xrightarrow{a} q' \quad \{q \not\uparrow b \mid a \prec (b, c)\} \quad q \uparrow c}{q \xrightarrow{a} q'} \; \right| \; (a, \_, c) \in \pi \right\} . \tag{3.20}$$

The symbol $\bot$ allows us to add witness premises without any negative ones by putting $a \prec (\bot, c)$. Clearly, to add negative premises without any witness ones, it is sufficient to have $(a, b, \varnothing) \in \pi$.[4]

Definition 3.3.6. An n-ary *Offer BIP glue operator* is a triple $\left((P_i)_{i=1}^n, \gamma, \pi \right)$, where $(P_i)_{i=1}^n$ are pair-wise disjoint sets of ports and, denoting $P \stackrel{def}{=} \bigcup_{i=1}^n P_i$, the remaining two elements $\gamma \subseteq 2^P$ and $\pi \subseteq 2^P \times \left((2^P \cup \{\bot\}) \times 2^P \right)$ are, respectively, interaction and offer priority models on P.

We denote by $\mathbf{OBIP} = (\mathbf{G_{OBIP}}, \mathbf{C}, \mathbf{B}', \simeq', \sigma')$ (Offer BIP) the framework with $\mathbf{C}$, $\mathbf{B}'$ and $\simeq'$ defined as in Section 3.3.1 and the set $\mathbf{G_{OBIP}}$ comprising all Offer BIP glue operators with their corresponding semantics given by the rules (2.1) and (3.20).

3.3.3 Further expressiveness results

For the sake of comparison, we define two additional frameworks.

WBSOS (Witness BIP-like SOS) extends BIP-like SOS glue operators with positive *witness* premises $q \xrightarrow{c} q'$ that do not contribute to the transition defined by the rule, i.e. the enabledness of a transition in one of the components is tested without the transition being fired. Witness premises in **WBSOS** mimic the activation premises in **FNASOS** without using the offer predicate. As **FNASOS**, **WBSOS** possesses uniform flattening [BB20].

ABIP (Activation BIP) is a hybrid framework, which mixes the classical and offer semantics by relying on the usual transition relation for the inhibiting component of the priority model (negative premises) while using the offer predicate for that of the *activation* component (non-firing positive premises).

The expressiveness relations among all the frameworks defined so far are shown in Figure 3.3 [BB20]. Observe that, contrary to the **CBIP** $\Rightarrow$ **BSOS**, we have strong equivalence **OBIP** $\Leftrightarrow$

---

[3]Contrary to the definition in Section 2.1.1 we do not impose additional restrictions on π. In particular, we cannot require it to be a partial order, since its domain and co-domain are not the same.

[4]We cannot use $\varnothing$ instead of $\bot$ because $q \uparrow \varnothing = true$ (see Definition 3.3.1).
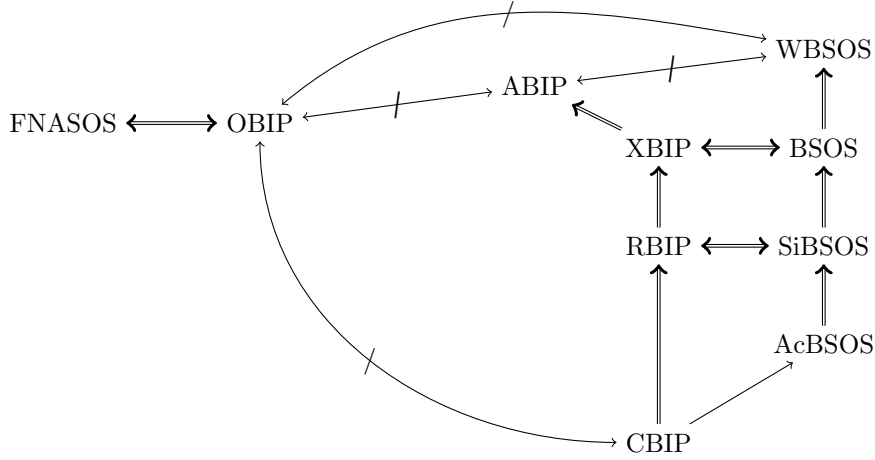
Figure 3.3: Expressiveness relations among all the considered frameworks [BB20]

FNASOS. In particular, a key property that contributes to "collapsing" the expressiveness hierarchy is that $q{\uparrow}b \wedge q{\uparrow}c = q{\uparrow}(b \cup c)$, which does not hold for the transition relation $\rightarrow$. Thus, in particular, there is no need to have multiple inhibiting interactions as in **XBIP**.

One should also notice that, by Propositions 3.1.11–3.1.13, relations shown in Figure 3.3 also imply **OBIP** $\longleftrightarrow\!\!\!\!/$ **XBIP**, **OBIP** $\longleftrightarrow\!\!\!\!/$ **RBIP** etc. (and similarly for **FNASOS**).

It is interesting to observe that, when restricted to flat systems, i.e. those consisting of one operator applied to sets of atomic components, a comparison can be established: every OBIP operator can be expressed as an ABIP operator and every ABIP operator can be expressed as a WBSOS one. The restriction to atomic components is crucial here, with the key property being that the equivalence $q{\uparrow}p \Longleftrightarrow \exists a : q \xrightarrow{a} \wedge p \in a$ holds only on atomic components.

Although the classical and offer BIP are not comparable in general, we have presented in [BB15] a characterisation of the behaviour hierarchy, consisting of three properties, which allows us to draw some commonalities.[5] The first property, when satisfied by all operand components, guarantees that the same glue operators (interaction and priority models) that are used with the classical semantics, can be used with the offer semantics to obtain an equivalent system. In general, the first property is not preserved by composition. However, it is preserved in hierarchical systems, where all priority models are applied after all interaction models. This allows us to conclude that the behaviour of any such system, where atomic components satisfy the first property, is not affected by switching from the classical to the offer semantics.

The second and third properties are more technical. If satisfied by all atomic components in the system, the second property, which is weaker than the first one, guarantees that the combination of glue operators (interaction and priority models) in the classical semantics can be adapted to the offer semantics by some additional modifications to the priority model.

The third property—the weakest of the three—guarantees only that, for any system built in the classical semantics, if the property is satisfied by all atomic components, a glue operator can be found in the offer semantics that would construct an equivalent system with the same atomic components.

---

[5] In [BB15], we use a different but equivalent representation for Offer BIP glue operators.

3.4   Key contributions

A first attempt at the characterisation of the BIP expressiveness was made in [BS08a]. Among the claims made in that paper, the most consequential—although technically minor—can be informally summarised by the statement: "BIP possesses the expressiveness of the universal glue". With my former PhD student Eduard Baranov, we have provided in [BB15] a counter-example showing that the classical semantics of BIP does not possess flattening, which implies that it does not possess strong full expressiveness w.r.t. BIP-like SOS either. Indeed, the above informal statement is wrong. The fundamental reasons for this absence of strong full expressiveness lie in the definition of the priority models driven by the imperative that applying a priority model do not introduce deadlocks in the otherwise deadlock-free system. This property turns out to be one of the key reasons underlying the expressiveness limitations, since deadlocks can be introduced by certain operators respecting the BIP-like SOS format.

This has motivated a further study that we have conducted with Eduard Baranov to characterise the expressiveness of BIP and its variants. That work forms the core of the present chapter. Its initial elements were published in the proceedings of the EXPRESS/SOS 2016 workshop [BB16] and followed by a significantly extended journal version [BB20].

In chronological order

[BS08a]   Simon Bliudze and Joseph Sifakis. "A Notion of Glue Expressiveness for Component-Based Systems". In: *CONCUR 2008*. Ed. by Franck van Breugel and Marsha Chechik. Vol. 5201. LNCS. Springer, 2008, pp. 508–522. DOI: 10.1007/978-3-540-85361-9_39.

[BB15]   Eduard Baranov and Simon Bliudze. "Offer semantics: Achieving compositionality, flattening and full expressiveness for the glue operators in BIP". In: *Science of Computer Programming* 109.0 (2015). Pp. 2–35. ISSN: 0167-6423. DOI: 10.1016/j.scico.2015.05.011.

[BB16]   Eduard Baranov and Simon Bliudze. "A Note on the Expressiveness of BIP". In: *Proceedings Combined 23rd International Workshop on Expressiveness in Concurrency and 13th Workshop on Structural Operational Semantics, EXPRESS/SOS 2016*. Vol. 222. EPTCS. 2016, pp. 1–14. DOI: 10.4204/EPTCS.222.1.

[BB20]   Eduard Baranov and Simon Bliudze. "Expressiveness of component-based frameworks: A study of the expressiveness of BIP". In: *Acta Informatica* 57.6 (Dec. 2020). Pp. 761–800. DOI: 10.1007/s00236-019-00337-7.

JavaBIP

This chapter presents JavaBIP—a Java implementation of BIP coordination mechanisms initially developed in collaboration with Crossing-Tech S.A.—an EPFL Innovation Park company—and two PhD students [Bli+17] and aimed at general purpose software engineering rather than the design of embedded systems.

The main challenge comes from the fact that, in the context of general purpose software engineering, one cannot expect developers to take a disciplined essentially top-down approach relying exclusively on high-level models and semantics-preserving transformations. The key reasons are 1) the complexity of the software stack and, in particular, the broad use of existing libraries and frameworks and 2) rapid code evolution due, for example, to agile development methodologies.

In such context, instrumenting the code to introduce coordination primitives would increase the maintenance costs and strongly impede the work of developers by forcing them to take this instrumentation into account in further evolutions of the software. For this reason, we have decided to move away from the code generation paradigm, instead relying on Java annotations and reflection mechanisms to define BIP models associated to Java objects.

Component specifications required by JavaBIP provide an abstract view of the software under development. Beyond coordination, which is the primary goal of JavaBIP, this abstraction of the software component behaviour can be used for test generation and run-time monitoring.

In this chapter, I briefly present

- the coordination mechanisms adopted in JavaBIP

- the modular architecture used for the JavaBIP implementation

- the mechanism allowing to add and remove components from the system dynamically

- two applications currently under development illustrating the possibilities for obtaining high-level models

4.1 Quick tour of JavaBIP

4.1.1 The key notions

A runnable system in JavaBIP consists of two major parts: the *engine* and several *modules*, one for each component to be coordinated (see Figure 4.1). In a nutshell, a JavaBIP component extends a Java class with a behavioural *BIP specification* (that we shorten to "BIP spec") describing an FSM through Java annotations. Thus, each module is composed of 1) a plain Java object encompassing the functional code of the component, 2) an FSM providing the corresponding *behaviour specification*, and 3) a dedicated *executor*—an object that implements the FSM semantics by keeping track of its current state and of the available transitions.

Transitions of FSMs defining component behaviour specifications can be of three types: *enforceable*, *spontaneous* and *internal*. Enforceable transitions are controlled by the engine. At each execution cycle, executors inform the engine about enforceable transitions offered by the components in their current state. The engine decides which of these should be executed and notifies the executors of its decision. Spontaneous transitions are used to take into account changes in the
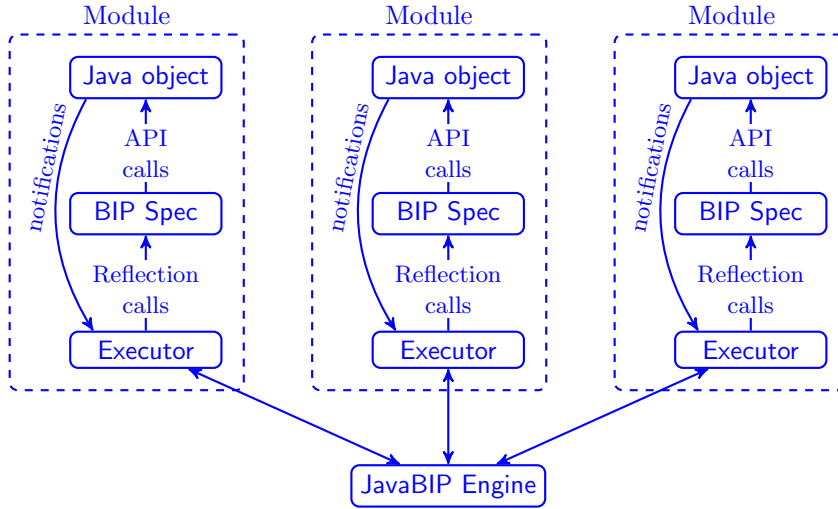
Figure 4.1: High-level view of the JavaBIP runnable system architecture

environment and, therefore, they are not announced to the engine but rather executed after detection of events in the environment of the component. Finally, internal transitions allow behaviour specifications to update their state based on internal information—when enabled, they are executed immediately. Spontaneous and internal transitions cannot be used for synchronisation with other components.

We have developed a generic `Executor` class that is instantiated in conjunction with each BIP spec, i.e. a Java class with BIP annotations. It drives the execution of the corresponding component using Java Reflection API. The `Executor` is also responsible for communicating with the JavaBIP engine to enforce the BIP protocol.

The coordination constraints are specified in terms of *glue* and *data wires*. The glue consists of synchronisation constraints encoding the set of possible interactions as described in the previous chapters. Contrary to the classical BIP, we have decided to use a Boolean encoding based on first-order interaction logic as the underlying representation of glue. This notation imposes synchronisation constraints based on component types rather than on component instances, which allows developers to write glue specifications without knowing the exact number of components in the system. The classical BIP connectors can be implemented on top of this Boolean encoding as syntactic sugar. Similarly, data transfer is specified using a lower level mechanism than in the classical BIP. It is specified as a set of data wires connecting required inputs with provided outputs of the components. However, the following key principle of BIP holds also for JavaBIP: *data transfer can only happen upon an interaction involving ports through which the corresponding data variables are accessible.*

The behaviour specification of each component along with the glue and data wire specifications are provided to the engine. The engine orchestrates the overall execution of the system by 1) deciding which component transitions must be executed at each cycle, 2) transferring the necessary data, and 3) notifying the executors of the selected transitions. The executors then call the appropriate methods of the BIP Spec objects encompassing the functional code of the corresponding components.

4.1.2 The Camel Routes example

To illustrate the basic notions presented in Section 4.1.1, let us consider the Camel routes example, which was our motivating use case during the initial development of JavaBIP. The use case consists in managing the memory usage by a set of Camel routes [TASF]. Camel routes were
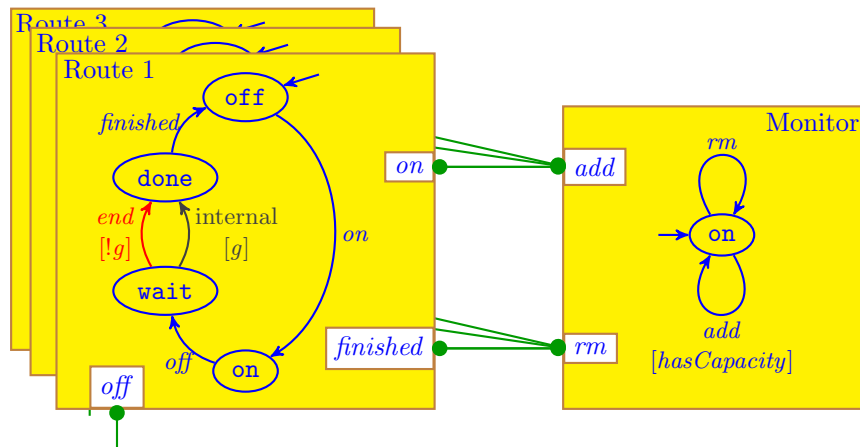
48

Figure 4.2: JavaBIP models of three routes and a monitor

extensively utilized in Connectivity Factory™—the flagship product of our industrial partner Crossing-Tech S.A. A Camel route connects a number of data sources to transfer data among them. The data can be fairly large and may require additional processing. Hence, Camel routes share and compete for memory. Without additional coordination, simultaneous execution of several Camel routes can lead to `OutOfMemory` exceptions, even when each route has been tested and sized appropriately on its own.

The Camel API provides the methods `resume` and `suspend` to control the activation of a route.[1] For simplicity, we assume here that the memory used for data transfer by an active route is known, whereas the memory used by a suspended route is negligible.

Our goal is to limit the number of routes running simultaneously to ensure that the available memory is sufficient for the safe functioning of the system. To achieve this, we introduce an additional monitor component as shown in Figure 4.2 where it is composed with three routes. The behaviour specifications of the Route and Monitor component types are shown in Figures 4.3 and 4.4, respectively.

Behaviour specification

In Figure 4.2, enforceable transitions are shown with blue arrows, the only visible spontaneous and internal transitions are shown with a red and a dark grey arrow, respectively. Boolean expressions in square brackets are the corresponding transition guards.

The Route model, shown in the left-hand side of Figure 4.2, has four states: `off`, `on`, `wait` and `done`. Its initial state is `off` (cf. Figure 4.3 line 7). When the route is at state `off`, it can start working by executing the transition *on*. Respectively, when the route is at state `on`, it can suspend its work by executing the transition *off*. The *on* and *off* transitions are both enforceable (cf. Figure 4.3 lines 3–4) and are associated with the `resume()` and `suspend()` methods of the Camel API (see Figure 4.3 lines 19–27).

Following the call to `suspend()` associated with the transition *off*, the route moves to the state `wait`. If the route has finished processing the previous data batch, it will be suspended immediately—modelled by the internal transition to the state `done` (cf. Figure 4.3 lines 32–33). Otherwise, the internal transition is disabled. Instead, to move to the state `done`, the route has to wait for the processing termination event, associated with the spontaneous transition *end* (cf. Figure 4.3 lines 29–30 and line 2). Neither of these two transitions invokes any code—their role is to update the current state of the BIP Spec FSM. Since their *guards*—evaluated using the

---

```
1   @Ports({
2     @Port(name = "end", type = PortType.spontaneous),
3     @Port(name = "on",  type = PortType.enforceable),
4     @Port(name = "off", type = PortType.enforceable),
5     @Port(name = "finished", type = PortType.enforceable)
6   })
7   @ComponentType(initial = "off", name = "Route")
8   public class Route implements CamelContextAware {
9
10    private CamelContext camelContext;
11    private String routeId;
12    private int deltaMemory = 100; // Dummy value, for the sake of simplicity
13
14    public Route(String routeId, CamelContext camelContext) {
15      this.routeId = routeId;
16      this.camelContext = camelContext;
17    }
18
19    @Transition(name = "on", source = "off", target = "on")
20    public void startRoute() throws Exception {
21      camelContext.resume(routeId);
22    }
23
24    @Transition(name = "off", source = "on", target = "wait")
25    public void stopRoute() throws Exception {
26      camelContext.suspend(routeId);
27    }
28
29    @Transition(name = "end", source = "wait", target = "done", guard = "!g")
30    public void spontaneousEnd() {} // "!g" in the guard above means "not g"
31
32    @Transition(name = "", source = "wait", target = "done", guard = "g")
33    public void internalEnd() {}
34
35    @Transition(name = "finished", source = "done", target = "off")
36    public void finishedTransition() {}
37
38    @Guard(name = "g")
39    public boolean isFinished() {
40      return camelContext.getInflightRepository().
41        size(camelContext.getRoute(routeId).getEndpoint()) == 0;
42    }
43
44    @Data(name = "deltaMemory",
45      accessTypePort = AccessType.allowed, ports = { "on", "finished" })
46    public int deltaMemory() {
47      return deltaMemory;
48    }
49  }
```

Figure 4.3: Annotations for the Route component type

method `isFinished()` (Figure 4.3 lines 38–42)—are disjoint, only one of these two transitions can be enabled at the same time.

Contrary to the internal transition, which is executed immediately when its guard is enabled, the transition labelled by the spontaneous port *end* has to wait for the corresponding notification from the Camel route. In the current version of JavaBIP, the way such notifications are configured depends on the API provided by the Java class of the coordinated object. In the Camel routes example, they are configured using the so-called route policies, which allow defining callbacks on certain events. Since this configuration process is specific to the Camel routes examples and does not illustrate any generic JavaBIP concepts, I do not show it here.[2]

Checking whether the route has finished processing is performed through the guard *g* used in the negative form for the spontaneous transition *end* and in the positive form for the internal transition from the state `wait` (cf. Figure 4.2 and the `guard` attribute of the `@Transition` annotation in Figure 4.3 lines 29 and 32).

---

[2]The corresponding code can be consulted on GitHub.

```
1   @Ports({
2     @Port(name = "add", type = PortType.enforceable),
3     @Port(name = "rm", type = PortType.enforceable)
4   })
5   @ComponentType(initial = "on", name = "MemoryMonitor")
6   public class MemoryMonitor {
7
8     final private int memoryLimit;
9     private int currentCapacity = 0;
10
11    public MemoryMonitor(int memoryLimit) {
12      this.memoryLimit = memoryLimit;
13    }
14
15    @Transition(name = "add", source = "on", target = "on", guard = "hasCapacity")
16    public void addRoute(@Data("memoryUsage") Integer deltaMemory) {
17      currentCapacity += deltaMemory;
18    }
19
20    @Transition(name = "rm", source = "on", target = "on")
21    public void removeRoute(@Data(name="memoryUsage") Integer deltaMemory) {
22      currentCapacity -= deltaMemory;
23    }
24
25    @Guard(name = "hasCapacity")
26    public boolean hasCapacity(@Data("memoryUsage") Integer memoryUsage) {
27      return currentCapacity + memoryUsage < memoryLimit;
28    }
29  }
```

Figure 4.4: Annotations for the Monitor component type

The Monitor model, shown in the right-hand side of Figure 4.2, has only one state and two enforceable transitions: *add* (cf. Figure 4.4 lines 15–18) for adding running routes and *rm* (cf. Figure 4.4 lines 20–23) for removing them. The *add* transition has the guard *hasCapacity* (cf. Figure 4.4 lines 25–28) that checks whether the available memory limit of the system, defined through the constructor of the MemoryMonitor class (see Figure 4.4 lines 11–13), is sufficient for adding more running routes.

Coordination specification

The complete system consists of several routes and one monitor. The Route model is the same for all routes and the monitor is connected to all of them in the same manner, as shown in Figures 4.2 and 4.5.

The port *on* of each route component must synchronise with the port *add* of the monitor. Thus, if the available memory capacity is not sufficient, the *on* transition is blocked. Since the *add* port of the monitor is connected to the *on* ports of several different routes by binary connectors, it must only synchronise with one of them at a time. The *fluid interface* specification synchron(...).to(...) (Figure 4.5 line 5) is an example of syntactic sugar that we have developed on top of the funamental JavaBIP macros. In this example, it specifies a binary synchronisation *on*•—•*add*. Similarly, the transition *finished* of each route must be synchronised with the transition *rm* of the monitor (Figure 4.5 line 6).

To decide whether the memory available in the system is sufficient to add another route, the monitor needs to know how much memory that route is going to use. This value is provided by the route component through an output variable *deltaMemory* declared by the @Data annotation (Figure 4.3 lines 44–45). It is computed by the method deltaMemoryOnTransition() and accessible through ports *on* and *finished* (Figure 4.3 lines 44–48).

On the receiving end, the monitor uses an input variable *memoryUsage* (Figure 4.4 lines 16, 21 and 26) to evaluate the guard *hasCapacity* and to perform the update on the *add* and *rm* transitions.

```
1   BIPGlue bipGlue = new TwoSynchronGlueBuilder() {
2     @Override
3     public void configure() {
4       // Two binary connectors
5       synchron(Route.class, "on").to(MemoryMonitor.class, "add");
6       synchron(Route.class, "finished").to(MemoryMonitor.class, "rm");
7
8       // Singleton connector
9       port(Route.class, "off").acceptsNothing();
10      port(Route.class, "off").requiresNothing();
11
12      // Data wire
13      data(SwitchableRoute.class, "deltaMemory").to(MemoryMonitor.class, "memoryUsage");
14    }
15  }.build();
16
17  BIPEngine engine = engineFactory.create("myEngine", bipGlue);
```

Figure 4.5: Specification of the glue

Input and output ports are connected by directed binary data wires (cf. Figure 4.5 line 13; not shown in Figure 4.2 for the sake of clarity). Data exchange between the routes and the monitor happens if the corresponding synchronisation ($on\bullet\!\!-\!\!\bullet add$ or $finished\bullet\!\!-\!\!\bullet rm$) can be executed, i.e. at least one of the routes is in the state off or done, respectively.

Notice that the access control functionality implemented in this example, can also be implemented in an actor-based framework, through a synchronisation on a future. Indeed, by implementing the monitor as an actor capable of serving the request hasCapacity, it is sufficient to send such a request before resuming a route and store the returned Boolean yes-or-no value in a future, then immediately consulting this future as part of a branching condition. This would block the route activation until the reply from the monitor is available, effectively achieving a synchronisation between the route and the monitor. The advantage of the JavaBIP approach is that—contrary to the solution using a Boolean future—it does not require the synchronisation to be hardcoded in the route specification.

Indeed, following the BIP separation of concerns principle, the specification of the synchronisations $on\bullet\!\!-\!\!\bullet add$ and $rm\bullet\!\!-\!\!\bullet finished$ is not part of the behaviour specification presented in Figures 4.2, 4.3 and 4.4. This ensures the modularity of the system, since, for example, the same BIP spec of a Camel route can be reused with a completely different monitor. As another example, the system shown in Figure 4.6 ensures mutual exclusion among four routes by assembling them in a token ring *instead of* using a monitor. (Notice that the "token" can only be injected into the system once.)

Similarly, if, at a later stage in the project or for the purpose of debugging, the developer needed to introduce a logger component to keep track of route management operations, this could be achieved simply by defining the corresponding BIP specification and replacing binary synchronisations (Figure 4.2) by ternary ones.

4.1.3   Require and Accept macros for glue specification

The primitive mechanism provided in JavaBIP for the specification of glue relies on a macro notation, similar to the one introduced in [Boz+12b]. Consider a port p of a component type T labelling one or more transitions of T. The synchronisation constraint associated to all transitions of T labeled by p is the conjunction of a *causal* and an *acceptance* constraint, defined, respectively, by the **Require** and the **Accept** macros. We now describe the meaning of the two macros through representative examples. The generalization of the above definitions to more complex macros is straightforward, but cumbersome. Therefore we omit it here.
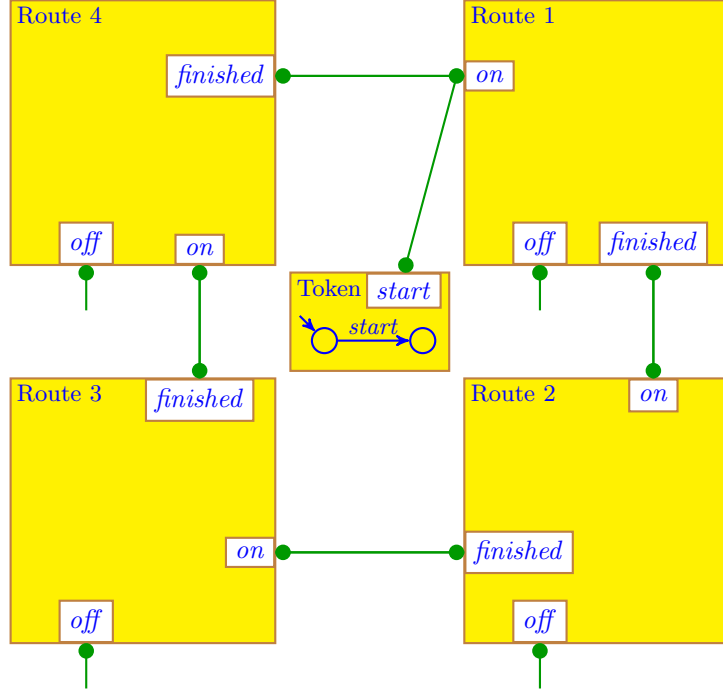
Figure 4.6: Four Camel routes arranged in a token ring (The behaviour of Camel route components is not shown for emphasis on the structure of the system. It is the same as in Figure 4.2.)

The Require macro is used to specify ports required for synchronisation. Let $T^1, T^2 \in \mathcal{T}$ be two component types. The macro

$$T_1.p \;\; \textbf{Require} \;\; T_2.q$$

means that, to participate in an interaction, each of the ports p of component instances of type T_1 requires synchronisation with *precisely one* of the ports q of component instances of type T_2. Notice that the cardinality of required component instances is explicit: should two instances of the same port type be required, this is specified by explicitly putting the required port type twice, e.g.

$$T_1.p \;\; \textbf{Require} \;\; T_2.q \; T_2.q \,,$$

and so on for higher cardinalities. We call *effect* what is specified in the left-hand side of **Require** (e.g. $T_1.p$) and *cause* what is specified in the right-hand side (e.g. $T_2.q \; T_2.q$). A cause consists of a set of *OR-causes*, where each OR-cause is a set of ports. For p to participate in an interaction, all the ports belonging to at least one of the OR-causes must synchronise. For instance,

$$T_1.p \;\; \textbf{Require} \;\; T_2.q \; T_2.q \; ; \; T_2.r$$

means that p requires either the synchronisation of two instances of q or one instance of r. Notice the semicolon that separates the two OR-causes.

The Accept macro defines optional ports for synchronisation, i.e. it defines the boundary of interactions. This is expressed by explicitly excluding from interactions all the ports that are not accepted. Let $T^1, T^2 \in \mathcal{T}$ be two component types. The following:
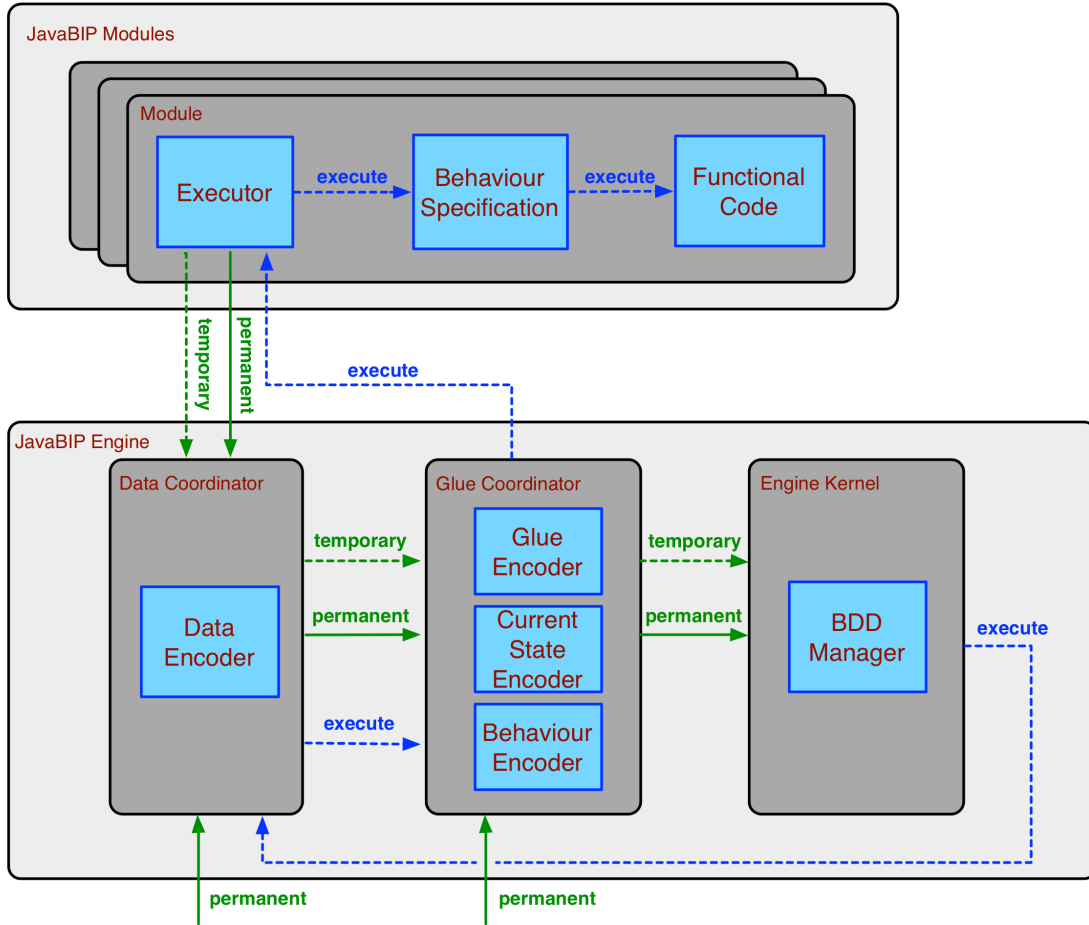
$$T_1.p \;\; \textbf{Accept} \;\; T_2.q$$

53

Figure 4.7: JavaBIP software architecture.

means that p accepts the synchronisation of instances of q but does not accept instances of any other port types.

4.2 Implementation

The software architecture of the *JavaBIP runnable system* is shown in Figure 4.7. As explained in Section 4.1.1, it consists of two main parts: the modules and the engine, shown, in the top and bottom parts of the figure, respectively. Each module sends to the engine the specification of its behaviour. The glue and data-wire specifications are provided to the JavaBIP engine separately.

4.2.1 JavaBIP engine

The implementation of the engine is modular. It consists of a stack of coordinators and the kernel. The coordinators manage the flow of information between the modules and the kernel. Coordinators use dedicated encoders to transform the diverse specifications into permanent and temporary Boolean constraints that are sent to the kernel. The imposed constraints can be of two types:

- *Permanent constraints* are received only once at initialization. They encode information

54

about the behaviour, glue and data wires of the components. In Figure 4.7, the flow of permanent constraints is shown with solid arrows labeled "permanent".

- *Temporary constraints* are received at each execution cycle. They encode information about the enabled transitions of the components. In Figure 4.7, the flow of temporary constraints is shown with dashed arrows labeled "temporary".

We have developed two coordinators that manage different types of constraints: the *Glue coordinator* and the *Data coordinator*. However, for systems without data transfer, it is sufficient to use only the Glue coordinator. Other coordinators can be easily added to manage other types of constraints—the implementation of the coordinator stack renders the architecture extensible.

The kernel solves the combined constraints imposed by the behaviour, glue and data-wire specifications and passes the solution back to the coordinators. Its implementation relies on Binary Decision Diagrams (BDDs)[3] [Ake78], which are efficient data structures to store and manipulate Boolean formulas. Each coordinator interprets the relevant part of the solution and triggers the corresponding action in the executors, where the actual API function calls to the controlled Java objects are made. If the kernel cannot find a solution because the combined constraints are contradictory, a deadlock occurs.

Glue coordinator

The Glue coordinator implements the basic BIP coordination, i.e. it manages the information about the behaviour, glue and current state of the components. It encompasses three dedicated encoders (cf. Figure 4.7): the Behaviour encoder, the Glue encoder and the Current State encoder. The Boolean constraints encoding component behaviour and glue are permanent, hence only computed once at initialization. The Boolean constraints encoding the current states of components are temporary, hence recomputed at each execution cycle.

Each component is registered with the Glue coordinator by providing its behaviour specification. Then, the Glue coordinator forwards to the Behaviour encoder the lists of enforceable ports and states of each registered component. For each enforceable port, a Boolean port variable is created by the BDD manager. Similarly, for each state, a Boolean state variable is created. The behaviour constraints are built, using the port and state variables.

The Behaviour encoder computes the behaviour constraint based on the following rules:

1. Each component can be at one state at a time.

2. Each state has the associated set of enforceable ports that can be enabled when the component is in that state.

3. A component may skip the cycle without executing any of the transitions and remaining at the same state.

The Glue encoder receives from the Glue coordinator the glue specification. The glue BDD is computed using the port variables that have been previously created by the Behaviour encoder.

The Current State encoder is notified each time the Glue Coordinator is informed of a component state. In addition to the current state, such notifications include the list of enforceable ports that are disabled due to evaluation of guards. Indeed, since they cannot be fired at that cycle, the valuation of the corresponding Boolean variables has to be set to *false* in any constraint solution considered by the engine kernel.

The current state BDD of each component is then transferred to the engine kernel, where the conjunction of all current state BDDs is computed at each execution cycle and is further conjuncted all permanent constraints.

---

[3]We have used the JavaBDD package, available at http://javabdd.sourceforge.net

Data coordinator

The Data coordinator is used on top of the Glue coordinator. Using the Data encoder, it encodes as permanent constraints the information about data wires, which connect input and output data provided by the components. In particular, at initialization phase, the Data coordinator receives the data-wire specification of the system (cf. Section 4.1.2). The Data coordinator queries the registered components to determine the ports that require data (input data) and the ports that provide data (output data). Then, the Data coordinator passes to the Data encoder the data wires and the pairs of ports that require and provide data. For each pair of ports, the Data encoder creates a Boolean data variable. It then computes the permanent constraint based on the following rules:

1. Data can only be transferred along a data wire connecting two ports if these ports participate in the interaction.

2. A port that requires input data either for its guard or for the associated method, can only be fired if such data is, indeed, provided.

In addition, at each execution cycle, the Data coordinator produces temporary constraints imposed on component interaction by the guards associated to component inputs. These temporary constraints block the data transfer along the data wires, where the proposed output data values do not satisfy the guards associated to the corresponding input data. To this end, each guard that requires input data is evaluated on all data values proposed along the data wires attached to the corresponding port.

4.2.2 Experimental evaluation

We show experimental results for four examples: 1) the Publish-Subscribe example illustrating mainly the use of spontaneous ports (full details in [Bli+17]); 2)–3) two implementations of the Camel routes example (Section 4.1.2), one with and one without data transfer; and 4) the Trackers-and-Peers example presented below. The JavaBIP models of these examples are available as part of the JavaBIP repository[4] on GitHub.

Trackers and Peers example

Trackers and Peers is a toy example with a complex coordination pattern involving multiparty interaction. It was initially presented in [Boz+12a]. Although it is inspired by a wireless audio protocol for peer-to-peer communication, it should be noted that this example does not implement any kind of message passing (see also footnote 7 on page 9). Here, we provide it purely as an example of a BIP model allowing us to stress-test the JavaBIP engine and evaluate the practical limitations of the current implementation.

The model has two component types: `Tracker` and `Peer`. The protocol allows an arbitrary number of peers to communicate along an arbitrary number of communication channels. Each channel is managed by a unique tracker.

The model for two peers and one tracker is shown in Figure 4.8. Peers are allowed to use at most one channel at a time. Access to channels is subject to the following registration mechanism. Every peer selects the channel it wants to use and registers through the *register* port that is synchronised with the *log* port of the tracker. During this synchronisation, components are exchanging data. In particular, the tracker sends its identity to the peer and the peer stores it. Once registered, peers can either speak to the channel or listen to other registered peers in the channel.

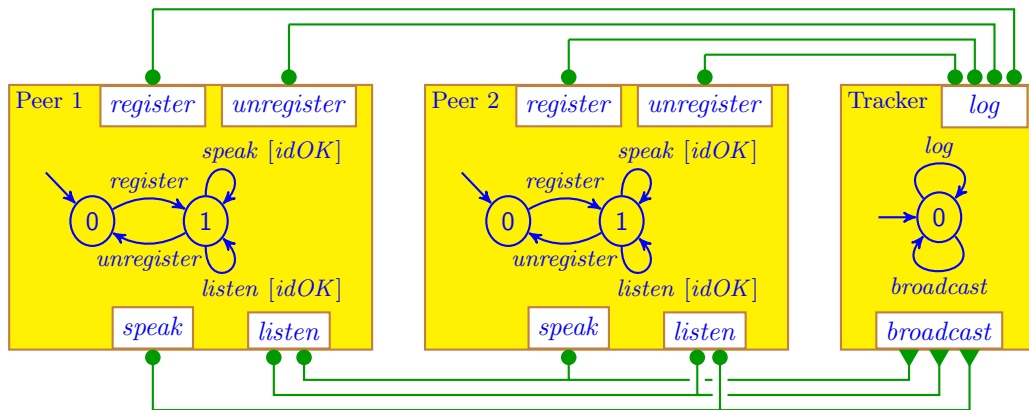
[4]https://github.com/sbliudze/

Figure 4.8: JavaBIP models of one tracker and two peers

To ensure atomicity of each communication, every tracker ensures that at most one registered peer is speaking on its corresponding channel. However, several peers can speak on different channels as part of the same interaction. Thus, every interaction can involve arbitrary subsets of peers and trackers as long as at most one peer speaks on each channel. The interaction structure of this example is, therefore, exponentially complex both in the number of peers and in the number of trackers.

To allow all peers to communicate with all trackers, all corresponding connectors must be present in the system. Data transfer is used to ensure that peers can interact only with the tracker they had previously been registered with: for each interaction, trackers propose their identity as data and peers use the *idOK* guard to decide with which trackers they can synchronise. Thus, all transitions of the system are enforceable and in all possible interactions (except when a tracker is broadcasting without any registered peers) data are exchanged between components.

Evaluation results

To simplify the evaluation and the presentation of the results, we have decided to ensure that the model sizes of all four examples are parametrised by one integer value, which is the total number of components of all types. In particular, in the Camel routes examples (with and without data), all components bar one monitor are routes. In the Trackers-and-Peers example, there are always four times more peers than trackers.

The experiments were run on an Intel Core i7-2640M CPU at $2.80\,\mathrm{GHz}$ x 4 with $8\,\mathrm{GB}$ RAM. Figure 4.9a shows the average execution time of the first 1000 engine cycles for all four examples, with the number of components ranging from 5 to 75. Figure 4.9b shows the peak memory usage of the BDD Manager for each of the three examples. Table 4.1 summarizes all results shown in Figures 4.9a and 4.9b. The two Camel routes examples illustrate the impact of data transfer on the performance of the engine. The behaviour and interaction models of the two Camel route examples are equivalent; in the latter, components also exchange data. Although data transfer causes an increase in the execution time and memory usage of JavaBIP, the overall coordination overhead remains low. The Publish-Subscribe example uses both enforceable and spontaneous transitions. Spontaneous transitions are not controlled by the JavaBIP engine which leads to low coordination overhead as illustrated in Figure 4.9a.

We conclude that the performance of the engine is mostly determined by the complexity of the glue specification. In the Trackers & Peers case-study, the number of possible interactions increases exponentially with the number of components: e.g. the number of possible interactions is in the order of 10^{24} for 75 components. Coordinating a system of such complexity with traditional techniques would be very difficult and error-prone. In JavaBIP, the full glue specification does
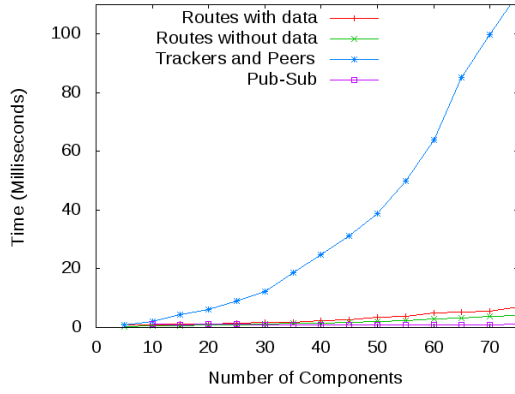
(a) Average engine execution time.
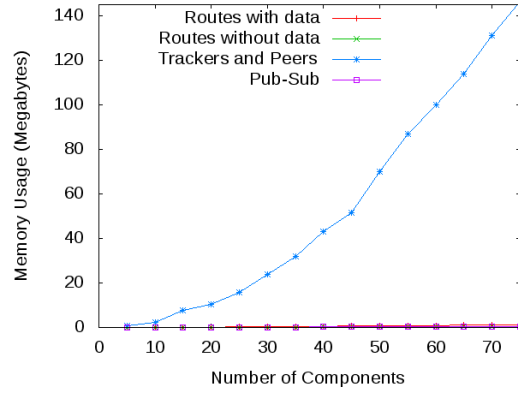
(b) BDD Manager peak memory usage.

Figure 4.9: Performance diagrams

Table 4.1: Engine times and BDD Manager peak memory usages

Nb of com- ponents	Publish-Subscribe		Routes no data		Routes with data		Trackers & Peers	
	Time (ms)	Memory (MB)	Time (ms)	Memory (MB)	Time (ms)	Memory (MB)	Time (ms)	Memory (MB)
5	< 1	0.026	< 1	0.010	< 1	0.015	< 1	0.640
10	< 1	0.048	< 1	0.029	< 1	0.075	2.103	2.278
15	< 1	0.084	< 1	0.047	1.147	0.099	4.264	7.584
20	< 1	0.108	< 1	0.079	1.254	0.180	6.002	10.338
25	< 1	0.130	< 1	0.099	1.585	0.220	8.980	15.932
30	< 1	0.161	1.254	0.120	1.614	0.324	12.329	23.670
35	< 1	0.184	1.328	0.169	1.895	0.456	18.643	31.896
40	< 1	0.233	1.459	0.200	2.393	0.560	24.727	43.045
45	< 1	0.251	1.874	0.238	2.731	0.700	31.187	51.598
50	< 1	0.295	2.167	0.280	3.568	0.780	38.943	69.984
55	< 1	0.315	2.346	0.340	3.796	0.840	49.766	87.097
60	< 1	0.338	2.786	0.387	5.093	0.920	63.766	99.983
65	< 1	0.366	3.286	0.410	5.345	1.028	85.327	113.983
70	1.001	0.394	3.749	0.450	5.548	1.105	99.876	131.237
75	1.125	0.437	4.133	0.488	6.970	1.170	113.657	146.476

not exceed 20 lines of code.

To evaluate the impact of the JavaBIP engine on the system execution time, we have used Camel Routes to transfer files of different size on the same machine that we used to run the above tests. We measured that a Camel Route needs 113 ms to transfer a 3 KB file, while for a 75 MB file a Camel Route needs 890 ms. Notice that this is "an ideal scenario" since only one Camel route is running at each time. In the case where four Camel Routes are running simultaneously, to transfer the 75 MB file we need more than 900 ms. We argue that the overhead induced by the JavaBIP Engine, which is less than 1 ms for four Camel Routes and one Monitor, is negligible when compared to 900 ms or even to 113 ms. Additionally, the memory usage of the BDD Manager remains very low—less than 2 MB for 75 components in the Camel Routes with data example.

4.3 Dynamicity

The JavaBIP version presented in the previous sections is static: to coordinate a system, the full set of components has to be registered before starting the engine. No components can be added on-the-fly and, most importantly, if a failure occurrs in a single component, the engine execution has to stop and the full set of constraints has to be computed anew. On the contrary, modern self-adaptive systems, ranging from cloud platforms and applications to nanosatellite constellations, make use of components that can join and leave during system execution. To allow component instances of known types, i.e. types for which synchronisation constraints exist, to register and deregister at runtime without any additional input from the developer, we have introduced a notion of system validity: *a system is valid if and only if its set of possible interactions is not empty*. The notion of validity allows the engine to be started and stoped automatically at runtime by just checking the status of the system. By stopping the engine if the system is invalid, we eliminate any processing time needed by the engine. To check system validity, we use directed graphs with edge coloring to model component synchronisation dependencies. Notice that the introduced notion of validity is only relevant for the engine: in an invalid system, components can still communicate asynchronously through notification of spontaneous events.

We have extended the interface and implementation of the engine to register, deregister, and pause a component at runtime. The difference between pausing and deregistering a component is as follows. If a component deregisters, then the engine clears all the associated data and references to this component; other components cannot synchronise with the deregistered component unless it registers anew. If a component is paused, other components cannot synchronise with it but the engine keeps all associated data and references to it; the paused component can start synchronising with other components by simply informing the engine that it is back on track. As above, a component is paused only from the perspective of the engine: this does not prevent it from executing internal transitions or interacting with other components through notification of spontaneous events.

The dynamicity extension was carried out by Valentin Rutz as part of his Master thesis work [Rut16], co-supervised by Anastasia Mavridou and me. It has not yet been merged into the JavaBIP main branch.

4.4 Applications

In this section, I briefly present two applications of JavaBIP currently under development, which both illustrate—from slightly different perspectives—the issue of obtaining high-level models for software coordination. The first one aims at the design of Cloud Computing systems through the integration with OCCIware, an existing domain-specific tool. In that context, FSM specifications for the various components are either know in advance or assumed to be provided by the designers.

Listing 4.1: Specifying the coordination constraints

```
1  <annotations name="Specification" >
2    <annotation id="switchServer_3">(Monitor_3.switchServer)−(Switch.switchServer)</annotation>
3    <annotation id="addDB_3">(Monitor_3.addDatabase)−(HerokuController.addDatabase)</annotation>
4    <annotation id="data_3">data: String currentReq: Monitor_3−HerokuController</annotation>
5    <annotation id="more_than_one_server">prop: (>= Monitor_3.RNRequestReady 2)</annotation>
6    ...
7    <annotation id="MAIN">switchServer_3, addDB_3, data_3, more_than_one_server</annotation>
8  </annotations>
```

The key point is, therefore, the specification and generation of the glue constraints.

The second application aims for the safe reconfiguration of software systems, relying on Feature Models for the specification of valid configurations. Contrary to the first application, no component behaviour is provided a priori, whereas the glue constraints are obtained in a straightforward manner from input feature models.

4.4.1 Exogenous coordination of Cloud Computing systems

In [Le +23], we have proposed a proof-of-concept framework for the design and validation of Cloud Computing (CC) systems following the exogenous coordination approach. To this end, we have extended the OCCIware [ZCM17; ZCM19] model-driven cloud resource management framework with coordination capabilities using JavaBIP. For validation we rely on the iFinder tool [Ben+09] from the BIP framework. OCCIware allows cloud architects to construct CC modeling frameworks that target specific cloud domains and reduce development time by supporting the `Models@run.time` approach [BBF09] and code generation [Par+15]. In particular, OCCIware provides the means for specifying the behaviour associated with OCCI entities as FSMs. While FSMs in the OCCIware models can be leveraged to monitor and coordinate the activities of the corresponding entities, *there are currently no such mechanisms available.*

Based on the FSM specifications of component kinds, BIP connectors provided as additional annotations, and on the configuration model that specifies the component instances of each kind, our tool generates a BIP model for verification using iFinder. The verified design is then used to generate the Java implementation comprising the usual OCCI resource connector templates with additional JavaBIP annotations and the glue code for runtime coordination.

For the detailed presentation of OCCIware, we refer the reader to [ZCM19]. Here, we shall only mention that it relies on the OCCI Core specification, which is a simple resource-oriented model. Among others it comprises the following concepts: `Resource` represents any cloud computing resource, e.g. a virtual machine, a network, an application container, an application. `Link` is a relation between two Resource instances, e.g. a computer connected to a network, an application hosted by a container. `Entity` is the abstract base class of all resources and links. `Kind` is the notion of class/type within OCCI, e.g. Compute, Network, Container, Application. Every entity has one kind. `Action` represents an action that can be executed on entities, e.g. start a virtual machine, stop an application container, restart an application, resize a storage. In addition, the OCCIware metamodel defines, among others, the concept `FSM` modelling the behaviour of OCCI concepts such as kind instances. We use such FSMs to model (Java)BIP component behaviour without any transformation (OCCI actions correspond to BIP ports). Note that the *OCCI design* defines kinds—instances of these kinds are defined separately in the *configuration model*.

OCCIware annotations To provide the means for specifying the coordination constraints, we have introduced a dummy resource called "`Specification`" (see Listing 4.1), which leverages OCCIware annotations to define: 1) BIP connectors, 2) data wires between components, and 3) properties that must be satisfied at runtime.

BIP connectors are defined using a textual representation based on the Algebra of Connectors

(see lines 2 and 3). We use parentheses to identify port names (e.g. action *switchServer* of the kind *Monitor₃* in line 2) and dashes to visually separate ports. In our OCCIware extension, data wire annotations start with the keyword "`data:`". They specify the type and the name of the data item, and the two ends of the wire. The annotation in line 4 declares that the string value `currentReq` can be transferred from *Monitor₃* to *HerokuController*. Linear properties can be specified in the prefix notation using the keyword "`prop:`" (e.g. line 5). These properties are passed directly to iFinder for verification. Finally, since these specifications are added to the OCCI design before the configuration model is created, we do not require that they all be necessarily activated in the final model. The `MAIN` annotation defines the list of other constraints to be included in the model (line 7).

Model verification with iFinder   The iFinder input consists of 1) a *BIP model* containing components and connectors, 2) the list of components with the algorithms for the invariant computations, and 3) the linear properties specified in the OCCI design. The generation of such input is straightforward. Component and connector types for the BIP model are obtained directly from the FSMs associated to OCCI kinds and connector specifications. The compound system is obtained by instantiating these types based on the configuration model. For the invariant computation, we use the iFinder `atom-control` algorithm for the components and `control-reachability` for the composed system [Ben+09]. Our tool then runs iFinder on the generated BIP model to check the specified linear properties and deadlock freedom. If the result is `valid`, the design can be used for implementation. Otherwise, iFinder returns a counter-example that the cloud architects can use to refine the design.

JavaBIP specifications   Generation of the BIP specifications for the components is straightforward since the FSMs are provided in the OCCIware model explicitly. The glue specification is generated in the form of **Require**/**Accept** macros by analysing connector specifications recursively to exhibit causality relations between groups of ports in a manner similar to that described in [BS10].

Figure 4.10 shows a fragment of the OCCIware design of the Monitor-Switch application, which is part of the case study we use for the validation (see [Le +23] for additional details). In particular, this application defines four scenarii embodied by different monitors defined in the OCCI design. Only one of these monitors needs to be included in the actual system. *Selecting a monitor only involves a change in the configuration model and the `MAIN` specification but no changes in the code of other components.*

Figure 4.11 shows the generated **Require** macros for the connectors specified in Listing 4.1 (**Accept** macros are omitted for brevity).

4.4.2   Runtime software variability models

Feature modeling is a widely used approach to capture commonalities and variability across software systems that are part of a product line or system family [Kan+90; Sch+12]. In particular, such models specify the configurations of the system that are deemed valid. A key concern in that context is how to ensure that *reconfiguration of a running software system* only involves valid configurations? Of course, this does not only apply to the initial and target configurations: *all intermediate configurations must also be valid* according to the variability model.

In [FBD22], we proposed an approach that leverages feature models for acquiring a compact representation of a set of valid configurations of a system in the form of a JavaBIP model used to control the software system at run time. The JavaBIP model runs alongside the software system, intercepts reconfiguration requests and enforces the constraints ensuring *by construction* that all intermediate configurations are safe. Since each feature is represented by a dedicated JavaBIP component, this is achieved without the need of computing the configurations neither at design
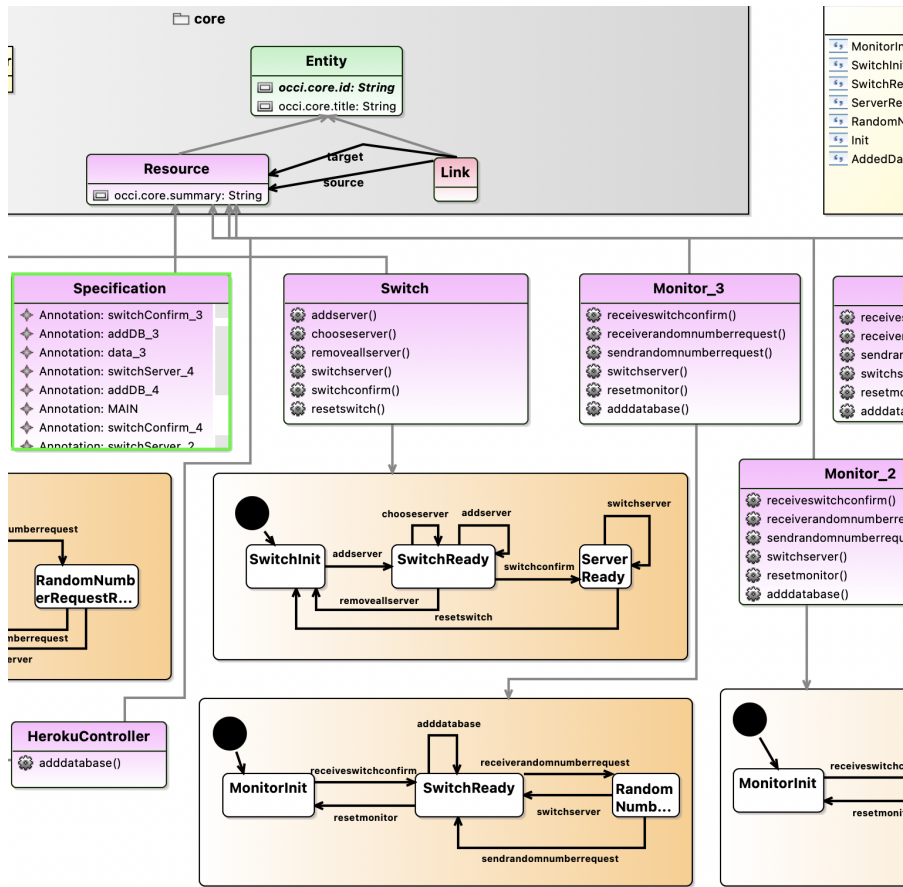
Figure 4.10: Fragment of the OCCIware design of *Monitor-Switch* Web application

```java
public class GlueBuilder_Specification extends GlueBuilder
{
    @Override
    public void configure() {
        //switchServer_3: (Monitor_3.switchServer)-(Switch.switchServer)
        port(Monitor_3Connector.class, "switchServer")
            .requires(SwitchConnector.class, "switchServer");
        port(SwitchConnector.class, "switchServer")
            .requires(Monitor_3Connector.class, "switchServer");
        // ... accepts part

        //switchConfirm_3: (Monitor_3.receiveSwitchConfirm)-(Switch.switchConfirm)
        port(Monitor_3Connector.class, "receiveSwitchConfirm")
            .requires(SwitchConnector.class, "switchConfirm");
        port(SwitchConnector.class, "switchConfirm")
            .requires(Monitor_3Connector.class, "receiveSwitchConfirm");
        // ... accepts part

        //addDB_3: (Monitor_3.addDatabase)-(HerokuController.addDatabase)
        port(Monitor_3Connector.class, "addDatabase")
            .requires(HerokuControllerConnector.class, "addDatabase");
        port(HerokuControllerConnector.class, "addDatabase")
            .requires(Monitor_3Connector.class, "addDatabase");
        // ... accepts part

        //data_3: data: String currentReq: Monitor_3-HerokuController
        data(Monitor_3Connector.class, "currentReq")
            .to(HerokuControllerConnector.class, "currentReq");
    }
}
```

Figure 4.11: Generated glue macros for the connectors specified in Listing 4.1
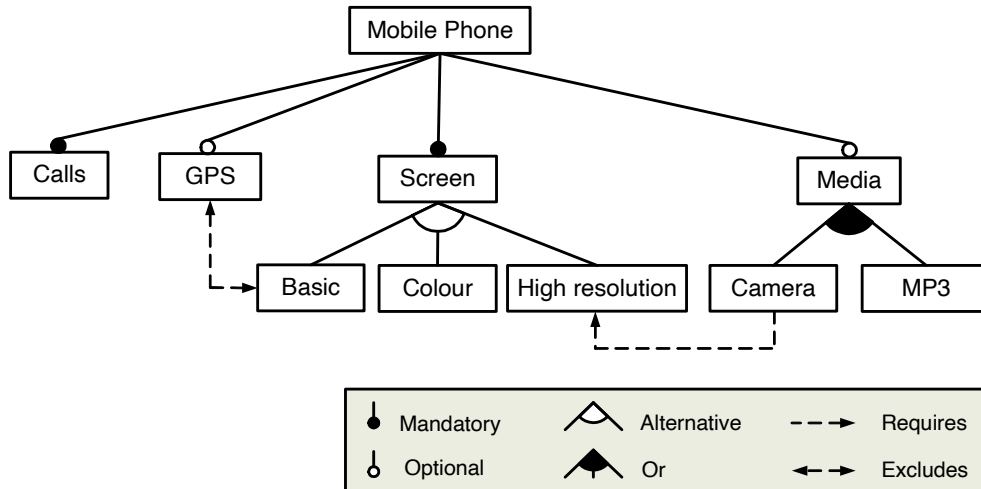
62

Figure 4.12: A sample feature model (from [BSR10])

nor at run time. Indeed, only the components corresponding to the features actually involved in the reconfiguration execute transitions.

Feature models   Feature models define variability in terms of features and their relationships. A feature could be a software artefact such as a part of code, a component, or a requirement. Feature models are usually depicted as tree diagrams whose nodes represent features that can be selected to build a software product. The tree hierarchy describes a composition relationship between features while additional constraints refine these relationships.

Every node of a feature model represents a *mandatory* or an *optional* feature. Furthermore, sub-features of a feature can form an OR- or a alternative (XOR) group. In addition to such structural constraints, feature models define two types of integrity constraints among features: one feature can *excludes* or *requires* another.

A simple example of the mobile phone variability model is shown in Figure 4.12. The features `Calls` and `Screen` are mandatory, whereas `GPS` and `Media` are optional. The `Screen` feature can be realised through at most one of the three alternative sub-features `Basic`, `Color`, and `High Resolution`. On the contrary, the `Media` feature can be realised through any combination of the sub-features `Camera` and `MP3`. The features `GPS` and `Basic` are mutually exclusive, whereas the `Camera` feature requires the `High resolution` screen.

The semantics of a feature model is the set of all valid configurations, i.e. sets of features. A configuration is *valid* if it satisfies all structural and integrity constraints of the feature model. A non-valid configuration that can be completed to a valid one by only adding features is called *partial*. All other configurations are *invalid*.

JavaBIP specifications   In the JavaBIP specification, one component is generated for each feature with the corresponding FSM constructed based on the feature type and on its subfeatures. For example, in an alternative group, only one feature can be selected at a time. Thus, if a feature is parent of an alternative group (e.g. the *Screen* feature in Figure 4.12), a corresponding component is created and all sub-features (*Basic*, *Colour* and *High resolution* in Figure 4.12) are represented by states in the FSM of that component.

More precisely, as illustrated in Figure 4.13, each sub-feature is represented by a group of states to reflect the fact that the feature can be requested, selected, requested for deselection and deselected. Requests for selection or deselection are intercepted and injected into the runtime
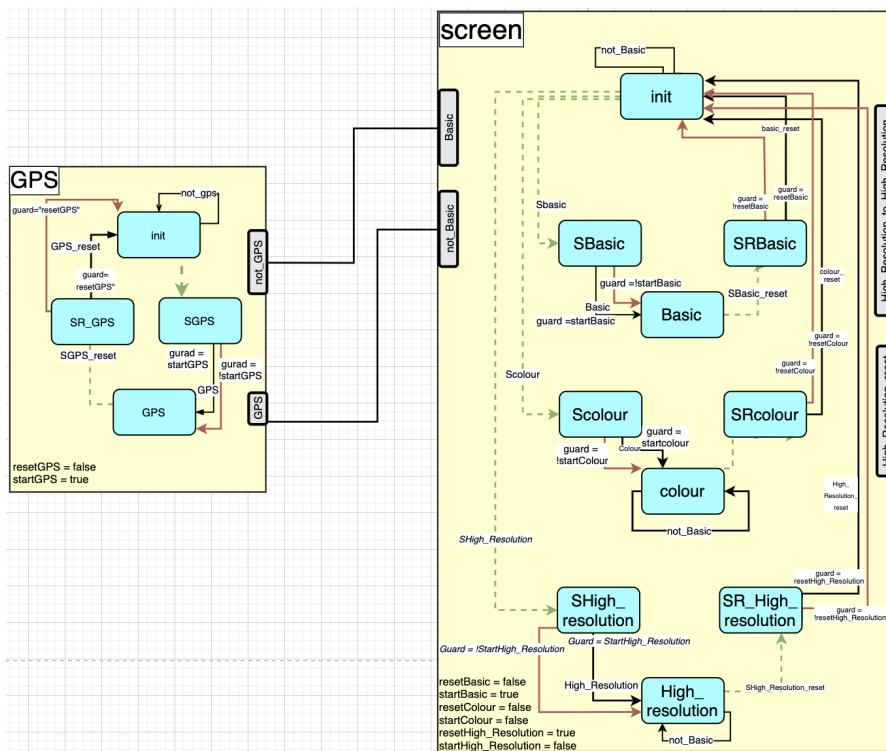
Figure 4.13: Fragment of the JavaBIP specification generated from the feature model in Figure 4.12 (enforceable, spontaneous and internal transitions are shown by solid black, dashed green and solid red lines, respectively)

JavaBIP variability model as notifications of spontaneous events. Thus the corresponding actions are only executed when the model allows to do so without violating the configuration validity. In particular, selection and deselection of features can happen out of order w.r.t. the corresponding requests.

While structural constraints are enforced by the component behaviours as above, parent-child relations among nodes of the feature model tree and integrity constraints are enforced by synchronising component behaviours. Synchronisations between parent and corresponding child nodes ensure that when a child feature is selected, the parent component is, indeed, in the corresponding state. Integrity constraints impose that two features must (resp. must not) be simultaneously selected. For a *requires* constraint, this is achieved by synchronising the transitions corresponding to the selection and the deselection of the two features. For an *excludes* constraint, we synchronise the selection of one feature with self-loop transitions (see Figure 4.13) signalling that the other feature is not selected (and vice versa). For example, consider the constraint imposing that the *Basic* feature excludes the *GPS* feature (Figure 4.12). The port *GPS* of component *GPS* is synchronized with the port *not_Basic* of component *Screen* to ensure that the *Basic* feature can be selected only if component *GPS* is not in the state *GPS*, meaning that feature *GPS* is not active.

4.5 Key contributions

Although the journal publication presenting the JavaBIP framework [Bli+17] came out only in 2017, initial developement was carried out at EPFL, in the context of a project funded by the Swiss Commission for Technology and Innovation (CTI), in 2012–13 by my former PhD students Anastasia Mavridou and Alina Zolotukhina with contributions and supervision by me and Radoslaw Szymanek who was representing our industrial partner.

The dynamicity extension was designed and implemented in a Master project by Valentin Rutz (EPFL) supervised mostly by my former PhD student Anastasia Mavridou [MRB17; Rut16].

Finally, two PhD students—Salman Farhat and Trình Lê Khánh—whom I have recently co-supervised at the Spirals team of Inria Lille and CRIStAL worked on the applications [Le +23; Far+23].

In chronological order

[Rut16]   Valentin Rutz. "Introducing dynamicity in JavaBIP". MA thesis. EPFL, School of Computer and Communication Sciences, June 2016.

[Bli+17]   Simon Bliudze, Anastasia Mavridou, Radoslaw Szymanek, and Alina Zolotukhina. "Exogenous coordination of concurrent software components with JavaBIP". In: *Software: Practice and Experience* 47.11 (Nov. 2017). Pp. 1801–1836. ISSN: 1097-024X. DOI: 10.1002/spe.2495.

[MRB17]   Anastasia Mavridou, Valentin Rutz, and Simon Bliudze. "Coordination of Dynamic Software Components with JavaBIP". In: *Proceedings of the 14th International Conference Formal Aspects of Component Software (FACS)*. Vol. 10487. Lecture Notes in Computer Science. Springer, 2017, pp. 39–57. DOI: 10.1007/978-3-319-68034-7_3.

[Far+23]   Salman Farhat, Simon Bliudze, Laurence Duchien, and Olga Kouchnarenko. "Toward Run-time Coordination of Reconfiguration Requests in Cloud Computing Systems". In: *Proc. of the 25th Int. Conf. on Coordination Models and Languages (COORDINATION)*. Vol. 13908. LNCS. Springer, June 2023, pp. 271–291. DOI: 10.1007/978-3-031-35361-1_15.

[Le +23]   Trinh Le Khanh, Hoang-Gia Nguyen, Simon Bliudze, and Philippe Merle. "Towards Exogenous Coordination of Concurrent Cloud Applications". In: *International Journal of Software Engineering and Knowledge Engineering* 0.0 (2023). Online ready, pp. 1–25. DOI: 10.1142/S0218194023500389.

Conclusion and future work

In this manuscript, I have presented some of my main contributions towards the overarching goal of ensuring software correctness.

As a post-doc at Verimag and a scientific collaborator at EPFL, I worked on the theoretical foundations and implementation of tools for the component-based design of correct-by-construction concurrent software and, specifically, on the BIP compositional framework. This approach provides a powerful and flexible way for the design of component-based heterogeneous systems, encompassing at the same time different paradigms of communication (broadcast, rendezvous), operation (synchronous, asynchronous) etc. A programming language implementing this model and a number of related tools have been developed at Verimag.

This work also underlies the more recent results, where we show that BIP *architectures* combining components and interaction models (but not priorities) enforcing behavioural properties can be composed in a constructive manner. We have shown that safety properties are always preserved and exhibited algorithmically verifiable conditions for the preservation of liveness properties.

Prior to that work, by-construction correctness provided by the BIP design flow was limited to the fact that automatically generated executable code was guaranteed to satisfy the properties established on the corresponding BIP models. The notion of architectures finally allows the by-construction correctness to be guaranteed already at the level of BIP models. This is achieved by establishing a direct association between behavioural properties and architectures that enforce such properties on BIP models. The architecture-based approach was validated by synthesising a BIP model for the on-board software of the CubETH nanosatellite, from a set of user requirements and a minimal number of initial atomic components. In the context of a collaborative project funded by the European Space Agency, we have systematised this approach and developped tools to support the users in the requirement specification.

With the goal of informing the choice of the coordination mechanisms for subsequent work. I have conducted a formal study of the expressiveness of the BIP composition operators. Deviating from the two traditional approaches defined, respectively, by the questions *What can be computed?* and *How concise is the program?*, this work has focused instead on the question of *What systems can be assembled from a given set of components?* In collaboration with my former PhD student Eduard Baranov, we have defined a simple algebraic framework for the comparison of expressiveness of component-based systems and applied it for a study of the expressiveness of BIP.

One of the conclusions of that study is that the key imperative underlying the complexity of the BIP expressiveness hierarchy is that application of priorities should not generate additional deadlocks in the composed system. Abandoning that imperative allowed us to define an alternative *offer* semantics for the BIP priorities operators, which collapses the expressiveness hierarchy and allows a Boolean encoding of arbitrary BIP glue, including both interactions and priorities. Furthermore—even though I have decided not to include this result in the present manuscript—in his PhD thesis, Eduard has shown that, with the offer semantics, the composability of architectures extends to architectures with priorities.

Such a Boolean encoding underlies JavaBIP—a framework for the coordination of Java components that implements the BIP coordination mechanisms. JavaBIP was initially developed in collaboration with another two of my former PhD students, Anastasia Mavridou and Alina

Zolotukhina and a company from the EPFL Innovation Park. We have 1) developed a set of libraries and annotations, which allow the specification of FSM associated to Java classes, and of the coordination constraints; 2) implemented an extensible engine that coordinates the execution of concurrent components based on a symbolic representation of coordination constraints provided by dedicated coordinators. Our implementation comprises the Glue Coordinator, implementing the BIP synchronization mechanism, and the Data Coordinator, allowing components to exchange data upon synchronizations. Other coordinators can be easily added with no or little impact on the framework architecture.

I have briefly presented an extension of JavaBIP, which allows starting, stopping, pausing and resuming components dynamically at run time, and two applications explored recently by Trình Lê Khánh and Salman Farhat—two PhD students whom I have co-supervised—which both illustrate from slightly different perspectives the issue of obtaining high-level models for the software coordination. The first one aims at the design of Cloud Computing systems through the integration with OCCIware, an existing domain-specific tool. The second aims for the safe reconfiguration of software systems, relying on Feature Models for the specification of valid configurations.

Directions of future work

The results presented in this manuscript contribute to the development of the Rigorous System Design approach, in general, and its implementation in the BIP and JavaBIP frameworks, in particular. Below, I briefly discuss five key directions that should be explored to further advance toward these goals.

Unifying modelling framework for self-adaptive systems The RSD approach consists in applying automatic model transformations, from a high-level specification model, through a sequence of intermediate transformations potentially using additional input models, to generation of executable code. Correctness guarantees are obtained based on two key factors: 1) availability and correctness of high-level models and 2) correctness of transformations. Thus, the cornerstone of the RSD approach is a unifying modelling framework, incorporating behavioural and composition models for the system components. Such a framework must be endowed with an unambiguous operational semantics, necessary to prove the correctness of both high-level models and of transformations. Given a modelling framework, an implementation of the RSD approach further relies on methods and tools for the *design of correct high-level models* and for the *generation of efficient executable code*.

The key challenges to address in the design of a unifying modelling framework for self-adaptive systems are the heterogeneity, reflexivity (the defining property of self-* systems) and dynamicity of the underlying systems arising from changes in both structure (instantiation and removal of components) and resource availability (e.g. due to failures or reconfigurations). Relegating the management of these aspects to component functionality is not an option, since that would dramatically curtail the modularity and maintainability of the system. Instead, they have to be explicitly taken into account both in theory—by the underlying component models and the corresponding composition operators—and in practice—by the coordination engines implementing their semantics.

Architecture styles Architectures, as presented in Chapter 2, are behaviour transforming operators applied to sets of components with essentially a *fixed* interface. On the contrary, an architecture style is a *parametrised* architecture that can be applied to an arbitrary set of components satisfying the minimal assumptions on their interfaces. The characteristic properties of architecture styles must be parametrised accordingly.

The first challenge w.r.t. architecture styles is to extend property preservation results. We also need an intuitive formalism that can be used by system designers without additional training

to specify, understand and apply architecture styles. Automatic synthesis of architecture styles from specifications expressed using notations commonly used in the industry (e.g. UML sequence diagrams) is key to ensuring a broad coverage of desired properties.

Currently the architecture-based design flow relies on the assumption that architectures do, indeed, enforce their stated characteristic properties. Broader adoption will be conditional on the possibility of providing formal guarantees that this is, indeed, the case. Although the BIP framework includes efficient verification tools for statically-defined systems, none are available for parametrised systems with an unbounded number of components. The parametrised model checking literature contains a wealth of techniques, such as the cut-off results for disjunctive and conjunctive guards [EK03], network decomposition techniques [Ami+14; Cla+04], techniques based on well-structured transition systems [Abd+96] and monotonic abstraction [Abd+09]. At least two different—yet complementary—approaches can be taken to address the validation of architecture styles with respect to their characteristic properties: 1) classifying the architecture styles and their characteristic properties, making them amenable to verification by the above techniques and 2) defining a bottom-up procedure for assembling architecture styles from simple constituents and inductively proving that the resulting architecture style indeed imposes the desired characteristic property.

Model extraction   Automatic code generation is widely accepted for the design of embedded systems. This is not the case in the general-purpose software engineering. Automatic code generation can conflict with the modifications of the underlying components, e.g. the changes of the software code carried out by the developers as part of software evolution due to both debugging and development of new versions. To ensure that the models obtained along the chain of the transformations remain relevant, we need mechanisms for the *detection of changes in the software code and their backward propagation to the high-level models.*

While the BIP framework assumes that the behaviour of all atomic components is explicitly defined in the BIP language, a BIP Spec in JavaBIP represents an abstraction of a pre-existing process. It serves as a mediator between the JavaBIP coordination engine and the actual component process being executed. BIP Specs are assumed to be *correct* and *provided by domain experts* posessing detailed knowledge of the meaningful states of a component. However, domain experts are not always available to provide BIP Specs. Thus, we need methods and tools for the automatic generation of BIP Specs by analysing the source code of the corresponding processes.

To obtain BIP Specs one could use existing tools (e.g. [Gha; Paw+15]) for parsing source code and building Abstract Syntax Trees (ASTs), which would then be explored to identify branching points (if, switch, loops, try-catch blocks etc.) for building the FSMs. Clearly, a naïve approach, considering all such points as control states, would result in FSMs that would be too large to be exploitable by designers. It will, however, provide the ground truth for the establishing the correctness of further advanced approaches. *The key difficulty is to determine what information is relevant for the purposes of component coordination and abstracting away the rest in a consistent manner.* This issue could be approached from different angles: using external configuration information, heuristics based on the analysis of class structure (fields tested by a large number of API methods are more likely to contribute significantly to the component state), branching conditions etc., or even AI-based tools [Den22]. Appropriate abstraction/refinement techniques, such as bisimilarity [Par81] with action refinement [GG01], would be necessary to ensure consistency between the resulting BIP Spec and the detailed FSM obtained by the naïve approach.

Model-software adequation   When a BIP Spec is made available, there is no guarantee that it *is and will remain* correct w.r.t. the process executing at run-time: the expert could have made a mistake or the process design could have evolved since the creation of the BIP Spec. Thus, we need run-time monitoring tools to test and detect the correspondence between BIP Specs and

the corresponding processes at run-time.

Both *static* and *dynamic* approaches must be developed to ensure that the models and the executable software stay coherent. Dynamic observation of the actual state of the software can reveal its inconsistencies with the runtime model (see also [Fal+14]). When such changes are detected, static analysis of the source code, potentially augemented with corresponding versionning histories, can be used to identify the changes to be incorporated into the runtime model and be propagated accordingly along the chain of the transformations, using techniques similar to [Par+11; Paw+15]. Alternatively, model learning [Ang87; Moe+16] and repair [ADS18] techniques must be used, when source code is not available.

Distributed implementation   Besides the issue of high-level models discussed above, another key challenge to be addressed to allow the adoption of the RSD approach in general and (Java)BIP in particular is the distributed implementation of the BIP Engine protocol. While synchronous interaction relied upon by both BIP and JavaBIP provides an abstraction that drastically simplifies reasoning about the coordinated system behaviour, it is rarely provided by target platforms and is hard to implement. Both BIP and JavaBIP implement the coordination semantics using centralised engines orchestrating the execution of system components, which represents a significant bottleneck for large systems. As a consequence, developers shy this abstraction in favour of message-passing communication primitives. These provide scalability but at the cost of a limit on the expressiveness of the coordination mechanisms, particularly so when multy-party coordination is required.

In the context of BIP, previous attempts [Bon+10; Qui13] at a distributed implementation exist but only partially address the problem. Most importantly, they disregard the inherent structure of BIP connectors considering only sets of flat interactions defined by a list of components that must all participate in the synchronisation. Eliminating structure from BIP connectors may lead to exponential explosion of the number of such flat interactions. Thus, we need new protocols that would take into account the interaction structure, if necessary, relaxing the atomicity of synchronisations.

Bibliography

[Abd+96] Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. "General decidability theorems for infinite-state systems". In: *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on.* IEEE. 1996, pp. 313–321.

[Abd+09] Parosh Aziz Abdulla, Giorgio Delzanno, Noomene Ben Henda, and Ahmed Rezine. "Monotonic abstraction: on efficient verification of parameterized systems". In: *International Journal of Foundations of Computer Science* 20.05 (2009), pp. 779–801.

[Ake78] Sheldon B. Akers. "Binary Decision Diagrams". In: *IEEE Transactions on Computers* C-27.6 (1978), pp. 509–516. ISSN: 0018-9340. DOI: 10.1109/TC.1978.1675141.

[Ami+14] Benjamin Aminof, Tomer Kotek, Sasha Rubin, Francesco Spegni, and Helmut Veith. "Parameterized model checking of rendezvous systems". In: *CONCUR 2014– Concurrency Theory.* Springer, 2014, pp. 109–124.

[Ang87] Dana Angluin. "Learning regular sets from queries and counterexamples". In: *Information and computation* 75.2 (1987), pp. 87–106.

[Att+14] Paul Attie, Eduard Baranov, Simon Bliudze, Mohamad Jaber, and Joseph Sifakis. "A General Framework for Architecture Composability". In: *12th International Conference on Software Engineering and Formal Methods (SEFM 2014)*. Ed. by D. Giannakopoulou and G. Salaün. LNCS 8702. Switzerland: Springer International Publishing, 2014, pp. 128–143. DOI: 10.1007/978-3-319-10431-7_10.

[Att+16] Paul Attie, Eduard Baranov, Simon Bliudze, Mohamad Jaber, and Joseph Sifakis. "A General Framework for Architecture Composability". In: *Formal Aspects of Computing* 18.2 (Apr. 2016). Open access, pp. 207–231. DOI: 10.1007/s00165-015-0349-8.

[ADS18] Paul C. Attie, Kinan Dak-Al-Bab, and Mouhammad Sakr. "Model and Program Repair via SAT Solving". In: *ACM Trans. Embedded Comput. Syst.* 17.2 (2018), 32:1–32:25. DOI: 10.1145/3147426.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN: 026202649X, 9780262026499.

[BB15] Eduard Baranov and Simon Bliudze. "Offer semantics: Achieving compositionality, flattening and full expressiveness for the glue operators in BIP". In: *Science of Computer Programming* 109.0 (2015). Pp. 2–35. ISSN: 0167-6423. DOI: 10.1016/j.scico.2015.05.011.

[BB16] Eduard Baranov and Simon Bliudze. "A Note on the Expressiveness of BIP". In: *Proceedings Combined 23rd International Workshop on Expressiveness in Concurrency and 13th Workshop on Structural Operational Semantics, EXPRESS/SOS 2016*. Vol. 222. EPTCS. 2016, pp. 1–14. DOI: 10.4204/EPTCS.222.1.

[BB20] Eduard Baranov and Simon Bliudze. "Expressiveness of component-based frameworks: A study of the expressiveness of BIP". In: *Acta Informatica* 57.6 (Dec. 2020). Pp. 761–800. DOI: 10.1007/s00236-019-00337-7.

[BCK12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 3rd. SEI Series in Software Engineering. Addison-Wesley Professional, Oct. 2012.

[Bas+16] Nick Bassiliades, Simon Bliudze, Panagiotis Katsaros, and Emmanouela Stachtiari. *General Concept and Technical Approach*. Tech. rep. ITT-AO1-7785-D6. EPFL IC IIF RiSD & AUTh, Feb. 2016.

[Bas96]   Twan Basten. "Branching bisimilarity is an equivalence indeed!" In: *Information Processing Letters* 58.3 (1996), pp. 141–147.

[BBS06]   Ananda Basu, Marius Bozga, and Joseph Sifakis. "Modeling Heterogeneous Real-time Components in BIP". In: 4th *IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM06)*. Invited talk. Sept. 2006, pp. 3–12. DOI: 10.1109/SEFM.2006.27.

[Bas+08]   Ananda Basu, Matthieu Gallien, Charles Lesire, Thanh-Hung Nguyen, Saddek Bensalem, Félix Ingrand, and Joseph Sifakis. "Incremental Component-Based Construction and Verification of a Robotic System". In: *ECAI*. 2008, pp. 631–635.

[BSR10]   David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. "Automated analysis of feature models 20 years later: A literature review". In: *Information Systems* 35.6 (Sept. 2010), pp. 615–636. DOI: 10.1016/j.is.2010.01.001.

[Ben+09]   Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. "D-Finder: A Tool for Compositional Deadlock Detection and Verification". In: *CAV*. Vol. 5643. Lecture Notes in Computer Science. (`iFinder` is a recent re-implementation of the D-Finder tool presented in this paper. The tool is available from the Verimag GitLab repository [accessed: 2022-07-18]). Springer, 2009, pp. 614–619.

[Ben+11]   Saddek Bensalem, Andreas Griesmayer, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. "D-Finder 2: towards efficient correctness of incremental design". In: *Proceedings of the 3rd international conference on NASA Formal methods*. NFM'11. Pasadena, CA: Springer-Verlag, 2011, pp. 453–458. ISBN: 978-3-642-20397-8. DOI: 10.1007/978-3-642-20398-5_32.

[Ben+12]   Albert Benveniste, Benoit Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas Henzinger, and Kim G. Larsen. *Contracts for System Design*. Research Report RR-8147. INRIA, Nov. 2012, p. 65.

[BBF09]   Gordon Blair, Nelly Bencomo, and Robert B. France. "Models@run.time". In: *Computer* 42.10 (2009), pp. 22–27.

[Bli+16]   Simon Bliudze, Nick Bassiliades, Panagiotis Katsaros, Anastasia Mavridou, Emmanouil Rigas, Joseph Sifakis, and Emmanouela Stachtiari. *Catalogue of System & Software Properties – Characterization of the Approach*. Tech. rep. ITT-AO1-7785-D13. EPFL IC IIF RiSD - AUTh, Dec. 2016.

[Bli+19]   Simon Bliudze, Sébastien Furic, Joseph Sifakis, and Antoine Viel. "Rigorous Design of Cyber-Physical Systems: Linking Physicality and Computation". In: *International Journal on Software and System Modeling* 18.3 (2019), pp. 1613–1636. DOI: 10.1007/s10270-017-0642-5.

[BHM19]   Simon Bliudze, Ludovic Henrio, and Eric Madelaine. "Verification of concurrent design patterns with data". In: *Proc. of the 21st International Conference on Coordination Models and Languages (COOORDINATION 2019)*. Ed. by Emilio Tuosto and Hanne-Riis Nielsen. Vol. 11533. LNCS. Springer, June 2019, pp. 161–181. DOI: 10.1007/978-3-030-22397-7_10.

[Bli+17]   Simon Bliudze, Anastasia Mavridou, Radoslaw Szymanek, and Alina Zolotukhina. "Exogenous coordination of concurrent software components with JavaBIP". In: *Software: Practice and Experience* 47.11 (Nov. 2017). Pp. 1801–1836. ISSN: 1097-024X. DOI: 10.1002/spe.2495.

[BS07]   Simon Bliudze and Joseph Sifakis. "The Algebra of Connectors—Structuring Interaction in BIP". In: *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software, EMSOFT 2007*. ACM SigBED. Salzburg, Austria, Oct. 2007, pp. 11–20. DOI: 10.1145/1289927.1289935.

[BS08a]   Simon Bliudze and Joseph Sifakis. "A Notion of Glue Expressiveness for Component-Based Systems". In: *CONCUR 2008*. Ed. by Franck van Breugel and Marsha Chechik. Vol. 5201. LNCS. Springer, 2008, pp. 508–522. DOI: 10.1007/978-3-540-85361-9_39.

[BS08b]   Simon Bliudze and Joseph Sifakis. "The Algebra of Connectors—Structuring Interaction in BIP". In: *IEEE Transactions on Computers* 57.10 (2008). Pp. 1315–1330. ISSN: 0018-9340. DOI: 10.1109/TC.2008.26.

[BS10]   Simon Bliudze and Joseph Sifakis. "Causal semantics for the algebra of connectors". In: *Formal Methods in System Design* 36.2 (June 2010). Pp. 167–194. DOI: 10.1007/s10703-010-0091-z.

[BS11]   Simon Bliudze and Joseph Sifakis. "Synthesizing Glue Operators from Glue Constraints for the Construction of Component-Based Systems". In: *10th International Conference on Software Composition*. Ed. by Sven Apel and Ethan Jackson. Vol. 6708. LNCS. Zurich, Switzerland: Springer, 2011, pp. 51–67. DOI: 10.1007/978-3-642-22045-6_4.

[Bli+14]   Simon Bliudze, Joseph Sifakis, Marius Dorel Bozga, and Mohamad Jaber. "Architecture Internalisation in BIP". In: *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering (CBSE '14)*. Marcq-en-Barœul, France: ACM, 2014, pp. 169–178. ISBN: 978-1-4503-2577-6. DOI: 10.1145/2602458.2602477.

[Blo+15]   Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, Sept. 2015.

[Bon+10]   Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. "From high-level component-based models to distributed implementations". In: *Proceedings of the 10th ACM international conference on Embedded software*. EMSOFT '10. Scottsdale, Arizona, USA: ACM, 2010, pp. 209–218. ISBN: 978-1-60558-904-6. DOI: 10.1145/1879021.1879049.

[Boz+12a]   Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis. "Modeling Dynamic Architectures Using Dy-BIP". In: *Software Composition*. Ed. by Thomas Gschwind, Flavio Paoli, Volker Gruhn, and Matthias Book. Vol. 7306. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 1–16. ISBN: 978-3-642-30563-4. DOI: 10.1007/978-3-642-30564-1_1.

[Boz+12b]   Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis. "Modeling dynamic architectures using Dy-BIP". In: *Software Composition*. Springer. 2012, pp. 1–16.

[BJS09]   Marius Bozga, Mohamad Jaber, and Joseph Sifakis. "Source-to-source architecture transformation for performance optimization in BIP". In: *Industrial Embedded Systems, 2009. SIES '09. IEEE International Symposium on*. 2009, pp. 152–160. DOI: 10.1109/SIES.2009.5196211.

[Cla+04]   Edmund Clarke, Muralidhar Talupur, Tayssir Touili, and Helmut Veith. "Verification by network decomposition". In: *CONCUR 2004*. Springer, 2004, pp. 276–291.

[DG08]   Deepak D'Souza and Madhu Gopinathan. "Conflict-Tolerant Features". In: *CAV*. Vol. 5123. LNCS. Springer, 2008, pp. 227–239. DOI: 10.1007/978-3-540-70545-1_22.

73

[Den22]   Denigma developers. *10 Best AI-Powered Coding Tools.* https://denigma.app/blog/
           posts/10-ai-powered-coding-tools/. Blog entry. [Accessed on 16/09/2022]. July
           2022.

[EK03]   E Allen Emerson and Vineet Kahlon. "Model checking guarded protocols". In: *Logic
           in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on.* IEEE.
           2003, pp. 361–370.

[EN95]   E. Allen Emerson and Kedar S. Namjoshi. "Reasoning About Rings". In: *Proceedings
           of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming
           Languages.* POPL '95. New York, NY, USA: Association for Computing Machinery,
           1995, pp. 85–94. DOI: 10.1145/199448.199468.

[Fal+14]   Jean-Rémy Falleri, Xavier Blanc, Reda Bendraou, Marcos Aurélio Almeida da Silva,
           and Cédric Teyton. "Incremental inconsistency detection with low memory over-
           head". In: *Software: Practice and Experience* 44.5 (2014), pp. 621–641. ISSN: 1097-
           024X. DOI: 10.1002/spe.2171.

[FBD22]   Salman Farhat, Simon Bliudze, and Laurence Duchien. "Safe Dynamic Reconfigu-
           ration of Concurrent Component-based Applications". In: *IEEE 19th International
           Conference on Software Architecture Companion, ICSA Companion 2022, Honolulu,
           HI, USA, March 12-15, 2022.* IEEE, 2022, pp. 108–111. DOI: 10.1109/ICSA-C54293.
           2022.00027.

[Far+23]   Salman Farhat, Simon Bliudze, Laurence Duchien, and Olga Kouchnarenko. "Toward
           Run-time Coordination of Reconfiguration Requests in Cloud Computing Systems".
           In: *Proc. of the 25th Int. Conf. on Coordination Models and Languages (COORDI-
           NATION).* Vol. 13908. LNCS. Springer, June 2023, pp. 271–291. DOI: 10.1007/978-
           3-031-35361-1_15.

[Fel90]   Matthias Felleisen. "On the expressive power of programming languages". In: 3rd
           European Symposium on Programming (ESOP'90). Vol. 432. LNCS. Springer, 1990,
           pp. 134–151. DOI: 10.1007/3-540-52592-0_60.

[Gam+94]   Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns:
           Elements of Reusable Object-Oriented Software.* 1st. Addison-Wesley Professional,
           Nov. 1994.

[Gha]   Seyed Mohammad Ghaffarian. *PROGEX (Program Graph Extractor): a cross plat-
           form tool for extracting graphical program representations from software source code.*
           GitHub repository. Accessed on 28/11/2019.

[GG01]   Rob van Glabbeek and Ursula Goltz. "Refinement of actions and equivalence notions
           for concurrent systems". In: *Acta Informatica* 37.4 (2001), pp. 229–327. ISSN: 1432-
           0525. DOI: 10.1007/s002360000041.

[GS03]   Gregor Gössler and Joseph Sifakis. "Priority Systems". In: *Formal Methods for Com-
           ponents and Objects, Second International Symposium, FMCO 2003, Leiden, The
           Netherlands, November 4-7, 2003, Revised Lectures.* Ed. by Frank S. de Boer, Mar-
           cello M. Bonsangue, Susanne Graf, and Willem P. de Roever. Vol. 3188. Lecture
           Notes in Computer Science. Springer, 2003, pp. 314–329. ISBN: 3-540-22942-6. DOI:
           10.1007/978-3-540-30101-1_15.

[Har87]   David Harel. "Statecharts: a visual formalism for complex systems". In: *Science of
           Computer Programming* 8.3 (June 1987), pp. 231–274. ISSN: 0167-6423. DOI: 10.
           1016/0167-6423(87)90035-9.

[Hoa85]   C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International
           Series in Computer Science. Prentice Hall, Apr. 1985.

[Kan+90]   Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. Carnegie-Mellon Univ, Pittsburgh, PA, Software Engineering Inst, 1990.

[Le +23]   Trinh Le Khanh, Hoang-Gia Nguyen, Simon Bliudze, and Philippe Merle. "Towards Exogenous Coordination of Concurrent Cloud Applications". In: *International Journal of Software Engineering and Knowledge Engineering* 0.0 (2023). Online ready, pp. 1–25. DOI: 10.1142/S0218194023500389.

[Ler+16]   Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. "CompCert – A Formally Verified Optimizing Compiler". In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. SEE. Toulouse, France, Jan. 2016.

[MRB17]   Anastasia Mavridou, Valentin Rutz, and Simon Bliudze. "Coordination of Dynamic Software Components with JavaBIP". In: *Proceedings of the 14th International Conference Formal Aspects of Component Software (FACS)*. Vol. 10487. Lecture Notes in Computer Science. Springer, 2017, pp. 39–57. DOI: 10.1007/978-3-319-68034-7_3.

[Mav+16]   Anastasia Mavridou, Emmanouela Stachtiari, Simon Bliudze, Anton Ivanov, Panagiotis Katsaros, and Joseph Sifakis. "Architecture-based Design: A Satellite On-board Software Case Study". In: *13th International Conference on Formal Aspects of Component Software (FACS 2016)*. Vol. 10231. Lecture Notes in Computer Science. 2016, pp. 260–279. DOI: 10.1007/978-3-319-57666-4_16.

[Mil89]   Robin Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.

[Moe+16]   Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michal Szynwelski. "Learning Nominal Automata". In: *CoRR* abs/1607.06268 (2016).

[Par81]   David M. R. Park. "Concurrency and Automata on Infinite Sequences". In: *Proceedings of the 5th GI-Conference on Theoretical Computer Science* (1981), pp. 167–183. DOI: 10.1007/BFb0017309.

[Par+15]   Jean Parpaillon, Philippe Merle, Olivier Barais, Marc Dutoo, and Fawaz Paraiso. "OCCIware — A Formal and Tooled Framework for Managing Everything as a Service". In: *Projects Showcase @ STAF'15*. Ed. by CEUR. Vol. 1400. Proceedings of the Projects Showcase @ STAF'15. L'Aquila, Italy, July 2015, pp. 18–25.

[Par+11]   Carlos Parra, Xavier Blanc, Anthony Cleve, and Laurence Duchien. "Unifying design and runtime software adaptation using aspect models". In: *Science of Computer Programming* 76.12 (2011). Special Issue on Software Evolution, Adaptability and Variability, pp. 1247–1260. ISSN: 0167-6423. DOI: 10.1016/j.scico.2010.12.005.

[Paw+15]   Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. "SPOON: A library for implementing analyses and transformations of Java source code". In: *Software: Practice and Experience* (2015). ISSN: 1097-024X. DOI: 10.1002/spe.2346.

[PR01]   Malte Plath and Mark Ryan. "Feature integration using a feature construct". In: *Science of Computer Programming* 41.1 (2001), pp. 53–84. ISSN: 0167-6423. DOI: 10.1016/S0167-6423(00)00018-6.

[Plo81]   Gordon D. Plotkin. *A Structural Approach to Operational Semantics*. Tech. rep. DAIMI FN-19. University of Aarhus, 1981.

[Qui13]   Jean Quilbeuf. "Distributed Implementations of Component-based Systems with Prioritized Multiparty Interactions". PhD thesis. University of Grenoble, 2013.

[Ros+15]   Roland Rosen, Georg von Wichert, George Lo, and Kurt D. Bettenhausen. "About The Importance of Autonomy and Digital Twins for the Future of Manufacturing". In: *IFAC-PapersOnLine* 48.3 (2015). 15th IFAC Symposium on Information Control Problems in Manufacturing, pp. 567–572. ISSN: 2405-8963. DOI: 10.1016/j.ifacol. 2015.06.141.

[Rut16]   Valentin Rutz. "Introducing dynamicity in JavaBIP". MA thesis. EPFL, School of Computer and Communication Sciences, June 2016.

[Sch+12]   Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. "Software diversity: state of the art and perspectives". In: *International Journal on Software Tools for Technology Transfer* 14.5 (July 2012), pp. 477–495. DOI: 10.1007/s10009-012-0253-y.

[Sif05]   Joseph Sifakis. "A Framework for Component-based Construction". In: 3rd *IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM05)*. Keynote talk. Sept. 2005, pp. 293–300. DOI: 10.1109/SEFM.2005.3.

[Sif12]   Joseph Sifakis. "Rigorous System Design". In: *Foundations and Trends$^®$ in Electronic Design Automation* 6.4 (2012), pp. 293–362. ISSN: 1551-3939. DOI: 10.1561/1000000034.

[Sta+18]   Emmanouela Stachtiari, Anastasia Mavridou, Panagiotis Katsaros, Simon Bliudze, and Joseph Sifakis. "Early validation of system requirements and design through correctness-by-construction". In: *Journal of Systems and Software* 145 (Nov. 2018). Pp. 52–78. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2018.07.053.

[TASF]   The Apache Software Foundation. *Apache Camel:Routes*. http://camel.apache.org/routes.html. [Accessed on 11/08/2021].

[Ver95]   Chris Verhoef. "A congruence theorem for structured operational semantics with predicates and negative premises". In: *Nordic Journal of Computing* 2.2 (1995), pp. 274–302. ISSN: 1236-6064.

[ZCM17]   Faiez Zalila, Stephanie Challita, and Philippe Merle. "A Model-Driven Tool Chain for OCCI". In: *On the Move to Meaningful Internet Systems. OTM 2017 Conferences*. Ed. by Herve Panetto, Christophe Debruyne, Walid Gaaloul, Mike Papazoglou, Adrian Paschke, Claudio Agostino Ardagna, and Robert Meersman. Cham: Springer International Publishing, 2017, pp. 389–409. ISBN: 978-3-319-69462-7.

[ZCM19]   Faiez Zalila, Stéphanie Challita, and Philippe Merle. "Model-Driven Cloud Resource Management with OCCIware". In: *Future Generation Computer Systems* 99 (Oct. 2019), pp. 260–277. DOI: 10.1016/j.future.2019.04.015.