Modelling Architecture Styles

THÈSE N^{O} 7324 (2016)

PRÉSENTÉE LE 21 DÉCEMBRE 2016

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE POUR LA CONCEPTION RIGOUREUSE DES SYSTÈMES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Anastasia MAVRIDOU

acceptée sur proposition du jury:

Prof. B. Faltings, président du jury
Prof. J. Sifakis, Dr S. Bliudze, directeurs de thèse
Prof. J. Sztipanovits, rapporteur
Prof. B. Rumpe, rapporteur
Prof. R. Guerraoui, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2016

Σα βγεις στον πηγαιμό για την Ιθάκη,
να εύχεσαι νάναι μακρύς ο δρόμος,
γεμάτος περιπέτειες, γεμάτος γνώσεις.
Κ. Π. Καβάφης

Στους παππούδες μου Αναστασία & Ιωάννη,
στους γονείς μου Ισαβέλλα & Ιωάννη,
& σε όλους τους δασκάλους μου.

To my grandparents Anastasia & Ioanni,
to my parents Isavella & Ioanni,
& to all my teachers.

Acknowledgements

I had the fortune, the honour and the absolute privilege to be advised by Prof. Joseph Sifakis and Dr. Simon Bliudze. Prof. Sifakis gave me the opportunity to explore very interesting research problems, while teaching me the beauty of theoretical rigour. Working with Prof. Sifakis has been a unique experience; I truly admire his passion and energy, as well as the enthusiasm with which he disseminates his knowledge. Dr. Bliudze always kept an open door, was happy to help with any kind of issue and patiently answer my numerous questions. I thank him for the insightful scientific discussions, which I greatly enjoyed. Ultimately, a few sentences would show nothing about how indebted I feel towards both Prof. Sifakis and Dr. Bliudze. The support and guidance I received was unbounded and highly motivational. I sincerely hope that my future work would partly meet their efforts.

I am grateful to Prof. Boi Faltings, Prof. Rachid Guerraoui, Prof. Bernhard Rumpe and Prof. Janos Sztipanovits for honouring me by accepting to serve on my thesis committee. I thank them for taking the time to read my thesis and for their invaluable comments on my work and its perspectives. I would like to additionally thank Prof. Sztipanovits for being my mentor during a summer internship at Vanderbilt University, which helped me to significantly increase my knowledge on model-based software engineering.

Special thanks go to Stefanos Skalistis, Goran Radanovic, Manos Karpathiotakis and Aron Laszka for proofreading my thesis. They helped me improve the flow and presentation of the thesis by finding several places in which further clarifications were needed.

I want to thank all my colleagues with whom I co-authored a paper: Emmanouela Stachtiari, Prof. Panagiotis Katsaros, Dr. Anton Ivanov and Alina Zolotukhina. Special thanks go to Dr. Radoslaw Szymanek for his invaluable guidance on the implementation parts of this thesis. Special thanks also go to Eduard Baranov with whom I had the pleasure to extensively interact, constructively argue and co-author several papers. I also thank my colleagues Alena Simalatsar, Wang Qiang, Wajeb Saab, Alexandre Sikiaridis and Vladimir Ilievski for creating a pleasant and motivating working environment. I thank Mrs. Arianne Staudenmann for her continuous help on administrative tasks while always having a smile on her face.

I would also like to express my gratitude to my dear friends for their continuous support and for making my life happier. Thank you Alex, Christos, Eleni, Eralia, Evi, Giannis, Iliana, Katerina, Kiki, Loukia, Manos, Matt, Pavlos, Stefanos, Theano and Zoi.

Last but not least, I thank my parents, Isavella and Ioannis, and my sister, Ifigeneia for their unconditional love throughout the years. They were there for me, even when we were physically apart.

Lausanne, 14 December 2016 A. M.

Abstract

Software systems tend to increase over time in size and complexity. Their development usually spans a long period of time and may result in systems that are hard to understand, debug and maintain. *Architectures* are common means for organising coordination between components in order to build complex systems and make them manageable. They allow thinking on a higher plane and avoiding low-level mistakes. Grouping architectures that share common characteristics into *architecture styles* assists component re-use and thus, the cost-effective development of systems. Architecture styles provide means for ensuring correctness-by-construction by enforcing global properties. The main goal of this thesis is to propose and study formalisms for modelling architectures and architecture styles.

For the specification of architectures, we study *interaction logics*, which are Boolean algebras on a set of component actions. We study a modelling methodology based on first-order interaction logic for writing architecture constraints. To validate the applicability of the approach, we developed the JavaBIP framework that integrates architectures into mainstream software development. JavaBIP receives as input architecture specifications, which it then uses to coordinate software components without requiring access to their source code. JavaBIP implements the principles of the BIP component framework.

For the specification of architecture styles, we propose *configuration logics*, which are powerset extensions of interaction logic. Propositional configuration logic formulas are generated from formulas of interaction logic by using the operators union, intersection and complementation, as well as a coalescing operator. We provide a complete axiomatisation of the propositional configuration logic and a decision procedure for checking that an architecture satisfies given logical specifications. To allow genericity of specifications, we study higher-order extensions of the propositional configuration logic. We provide several examples illustrating the application of configuration logics to the characterisation of architecture styles.

For the specification of architecture styles, we also propose *architecture diagrams*, which is a graphical language rooted in rigorous semantics. We provide methods to assist software developers to specify consistent architecture diagrams, generate the conforming architectures of a style and check whether an architecture model meets given style requirements. We present a full encoding of architecture diagrams into configuration logics. Finally, we report on applications of architecture diagrams to modelling architecture styles identified in realistic case studies of on-board satellite software.

Keywords: architecture styles, architectures, interaction logics, configuration logics, architecture diagrams, architecture-based design, BIP, JavaBIP, component coordination.

Résumé

Les logiciels informatiques ont tendance à augmenter au fil du temps en taille et en complexité. Leur développement couvre généralement une longue période de temps et se traduit souvent par des systèmes qui sont difficiles à comprendre, réparer et maintenir. Les architectures sont un moyen répandu pour organiser la coordination de composants afin de construire des systèmes complexes et simplifier leur gestion. Ils permettent de penser à un niveau d'abstraction supérieur et éviter les erreurs de bas niveau. Regrouper des architectures partageant des caractéristiques communes en des styles d'architecture aide à la réutilisation des composants et ainsi, au développement rentable de systèmes. En outre, les styles d'architecture offrent des moyens pour assurer l'exactitude par construction de systèmes, par l'application de propriétés globales. Le principal objectif de cette thèse est de proposer et étudier des formalismes pour modéliser des architectures et des styles d'architecture.

Pour la spécification d'architectures, nous étudions des logiques d'interaction, c'est-à-dire l'application d'algèbre booléenne sur des ensembles d'actions de composants. Nous étudions une méthodologie de modélisation basée sur la logique d'interaction de premier ordre pour l'écriture des contraintes d'architecture. Pour valider cette approche, nous avons développé le cadre JavaBIP, qui intègre les architectures dans le développement de logiciels grand public. JavaBIP reçoit comme entrée les spécifications d'architectures, qui sont ensuite utilisées pour coordonner les composants logiciels sans nécessiter l'accès à leur code source. JavaBIP met ainsi en œuvre les principes du cadre de programmation de composants BIP.

Pour la spécification des styles d'architecture, nous proposons des logiques de configuration, qui sont des extensions en ensemble des parties d'un ensemble (powerset en anglais) de la logique d'interaction. Les formules logiques de configuration propositionnelles sont générées à partir des formules logiques d'interaction à l'aide des opérateurs d'union, d'intersection et de complémentation, ainsi qu'un opérateur coalescent. Nous fournissons une axiomatisation complète de la logique de configuration propositionnelle et une procédure de décision pour vérifier qu'une architecture satisfait les spécifications logiques données. Pour permettre la généricité des spécifications, nous étudions les extensions d'ordre supérieur de la logique de configuration propositionnelle. Nous proposons plusieurs exemples illustrant l'application de la logique de configuration à la caractérisation des styles d'architecture.

Pour la spécification des styles d'architecture, nous proposons également des diagrammes d'architecture, c'est-à-dire un langage graphique basé sur une sémantique rigoureuse. Nous fournissons des méthodes pour aider les développeurs de logiciels à spécifier des diagrammes d'architecture cohérents, à générer des architectures conformes à un style, et à vérifier qu'un modèle d'architecture réponde aux exigences d'un style donné. Nous présentons un encodage complet des diagrammes d'architecture dans les logiques de configuration. Enfin, nous présentons des applications de diagrammes d'architecture à des styles d'architecture de modélisation, identifiées dans des études de cas réalistes de logiciel embarqué pour un satellite.

Mots clefs : styles d'architecture, architectures, logiques d'interaction, logiques de configuration, diagrammes d'architecture, design basé sur une architecture, BIP, JavaBIP, coordination de composants.

Contents

List of Figures

List of Tables

1 Introduction

Component-based engineering is essential for modern systems because it enhances correctness and productivity [33, 9]. It has been successfully used in many engineering disciplines such as civil, mechanical and electrical engineering. Based on the principles of component-based engineering, hardware systems can be constructed using a limited and well-defined number of components, such as registers, memories, buses, ALUs, etc., for which there exist specific composition rules [92]. Since hardware components have reached an adequate level of reliability [48, 59], the focus of system designers has shifted from the hardware counterparts of the systems to the software ones [89, 83].

Modern software systems are highly concurrent. They consist of multiple components that run simultaneously and share access to resources provided by the execution platform. To deal with this high level of concurrency, software architects use a vast variety of heterogeneous components with different behaviour and coordination mechanisms: synchronous or asynchronous, object-based or actor-based, event-based or data-based. Nevertheless, according to Sifakis [90], software lacks a well-defined component taxonomy and theory for component composition.

Two approaches exist for designing and developing component-based software systems. The first approach, namely the *architecture-agnostic approach* [23], considers the distinction between components and their associated coordination mechanisms to be irrelevant. It uses coordination primitives, which are spread across component behaviour. For instance, in Java, C, C++ and other programming languages the operations of concurrent programs are implemented as built-in features of the language [65, 103]. Synchronization primitives such as locks, semaphores and monitors are used to express coordination constraints. These low-level primitives are mixed up with the functional code, forcing developers to keep in mind both aspects simultaneously at design time, which often results in synchronization errors that introduce race conditions and deadlocks. 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 is exponential in the number of components, making *a posteriori verification* of their correctness practically infeasible. In fact, software architects and researchers at Microsoft underline the need for higher-level parallel constructs that more clearly express a programmer's intent, so that the parallel architecture of a program is more visible, easily understood and verifiable [95].

The second approach, namely the *architecture-based approach* [23], allows software architects to think on a higher abstraction level, separating functional and coordination aspects of the system behaviour. This approach uses *architectures* [84, 88, 96] to shift the focus of architects from low-level code to high-level structures ensuring coordination. Informally, architectures are characterized by the structure of the *interactions* between a set of *component types*. The structure is usually specified as a relation, e.g. *connectors* between component *ports*. Architectures can be understood as constraints that adequately restrict the behaviour of the coordinated components so as to achieve a desired coordination property. Software developers extensively use libraries of reference architectures ensuring both functional and non-functional properties; examples include fault-tolerant architectures, architectures for resource management and QoS control, time-triggered architectures and security architectures. When architectures are described in formal languages, they provide means for ensuring *correctness-by-construction*, shifting the burden of verification from a full model to architectures, which are considerably smaller in size and can be reused.

The importance of software architecture has been greatly acknowledged by the industry and academia. Standards have been produced by various organisations, the latest one being ISO/IEC/IEEE 42010 [53], which aims at standardising conventions on architectural descriptions. Additionally, there has been an increasing interest in defining languages that support the architecture-based approach, e.g. UML [78] and architecture description languages (ADLs) [75, 104, 80]. All these works rely on the distinction between behaviour of individual components and their coordination in the overall system organization. Nevertheless, they have a number of shortcomings. For instance, the connectivity primitives of some ADLs [45, 68, 61] allow only binary interactions. To specify n-ary interactions, these ADLs require an additional entity connected by n binary links with the interacting components. Additionally, these languages often lack formal semantics [87, 99, 79, 78]. As a result, analysis is carried out on models that cannot be rigorously related to system development formalisms. This introduces gaps in the design process which reduce productivity and limit the ability for ensuring correctness. In fact, in a survey conducted in the industrial sector regarding architecture description languages, it is stated that practising architects nowadays emphasize the need to reconcile informal notations with more formal and analysable ones [69].

Over the past two decades, there has been considerable progress in developing architecture-based design. Nevertheless, according to Garlan [42] "the field of software architecture remains relatively immature". A lot of foundational issues remain open. We consider that a theory of architectures must address, among others, the following fundamental questions:

1. *How does one model and implement architectures?* A rigorous theory requires formal definition of semantics. Characteristic properties of architectures must be understandable and verifiable by architects. Architecture specifications must be sufficiently

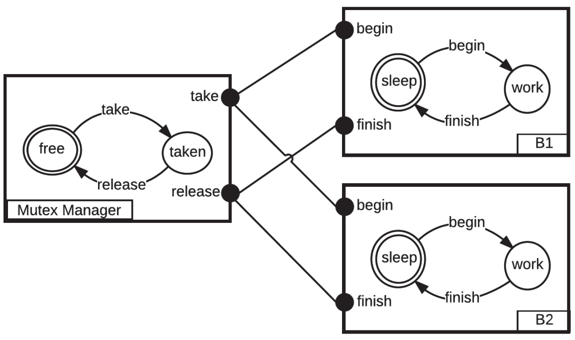


Figure 1.1 – Mutual exclusion model in BIP

versatile to be applicable to a variety of components and must not rely on the characteristics of any specific execution platform. Tools must provide support for integrating architectures into mainstream software development by raising the abstraction level and supporting separation of concerns, e.g. decoupling between behaviour and interaction.

2. *How to guide software developers in their choice of architectures?* Several architectures might enforce the same characteristic property, involve the same component types and coordination structure. In this case, we can group architectures in *architecture styles* to create a taxonomy of them and assist component-reuse.

3. *How does one specify architecture styles?* To be applicable to a variety of architecture styles, a rigorous theory requires formal semantics and versatile specifications. The specification language must be expressive and usable by architects. Additionally, mechanisms must exist to assist architects to correctly specify architecture styles, generate architectures from a style and check the conformance of an architecture to a specific style.

1.1 Architectures in BIP

A formal notion of architectures was proposed by Attie et al. [6], based on the *Behaviour-Interaction-Priority* (BIP) framework [9]. BIP is a component-based framework that provides an expressive formal language for the design of correct-by-construction systems. BIP superposes three layers. In the first layer, component *behaviour* is described by Finite State Machines having transitions labelled with *ports*. Ports form the interface of a component and are used to define its *interactions* with other components. The second layer defines component coordination by means of sets of interactions, i.e. sets of ports. A set of interactions is defined in a structured manner by using connectors [20]. In the third layer, priorities are used to impose scheduling constraints and to resolve conflicts when multiple interactions are enabled simultaneously.

Figure 1.1 shows a simple BIP model for mutual exclusion between two tasks. `B1`, `B2` model the tasks and `Mutex Manager` coordinates their actions. Initial states of the components are

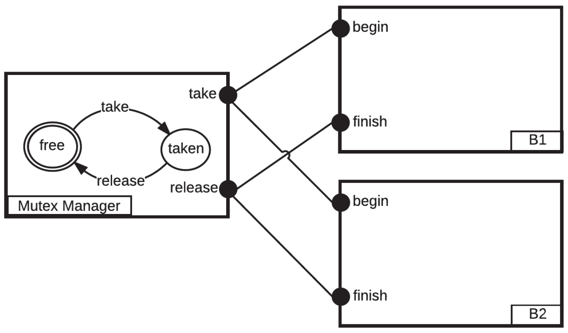


Figure 1.2 – Mutual exclusion architecture

shown with double lines. The four binary connectors synchronise each of the `begin` actions (resp. `finish`) of the tasks with the action `take` (resp. `release`) of the coordinator.

An architecture can be viewed as a BIP model, in which some of the atomic components are considered as *coordinators*, while the rest are *parameters*. When an architecture is applied to a set of components, these components are used as *operands* to replace the parameters of the architecture. Figure 1.2 shows an architecture that enforces the mutual exclusion *characteristic property* on two components `B1` and `B2` with interfaces `begin, finish`. The characteristic property of the architecture can be expressed in Computation Tree Logic (CTL) as $\mathtt{AG}\neg(cs[1] \wedge cs[2])$, where $cs[i]$ is an atomic predicate that evaluates to true when the component $B[i]$ is in its critical section (e.g. in the state `work`, for `B1`, `B2` of Figure 1.1). The architecture can be successfully applied if `B1` and `B2` satisfy the formula $\mathtt{AG}(finish[i] \rightarrow \mathtt{A}\ [\neg cs[i]\ \mathtt{W}\ begin[i]])$, where $begin[i]$ and $finish[i]$ are atomic predicates that evaluate to true in the states right after the execution of actions `begin, finish`, respectively.

1.2 Architecture styles

The definition of an architecture implicitly defines the number of components it is applied to. Architecture styles can be used to generalize this for any number of components. Architecture styles characterize not a single architecture but a family of architectures sharing common characteristics, such as the type of the involved components and the characteristic properties they impose [73]. Examples of architecture styles are Pipeline, Mutual Exclusion, Triple Modular Redundancy, Ring, Master/Slave, Pipe and Filter.

For instance, let us consider the Mutual exclusion architecture in Figure 1.2, which is applied on a set of precisely two operand components. The Mutual exclusion architecture style characterizes all the architectures that satisfy mutual exclusion for any number of operand components. The characteristic property of the Mutual exclusion architecture style is the following: '*no two components can be in their critical section simultaneously*'. This property is expressed by the CTL formula: $\mathtt{AG}\neg(\bigvee_{i \neq j \in [1,n]} cs[i] \wedge cs[j])$, where $cs[i]$ is an atomic predicate that evaluates to true when the component B_i is in its critical section.

The benefits of using architecture styles are numerous [96]. In particular, styles:

1. provide a disciplined and coherent way to specify and group coordination mechanisms as a means to master complexity in system design;
2. are means for ensuring correctness-by-construction since they enforce global properties characterizing the coordination among components;
3. provide a common vocabulary that promotes understanding of design decisions and makes communication between architects more efficient;
4. allow component re-use, enable the use of code generation and the development of systems in a cost-effective manner.

1.3 Our contributions

This thesis proposes a framework for the specification of architectures and architectures styles. The framework includes three specification languages: 1) *interaction logics* for the specification of architectures; 2) *configuration logics* and 3) *architecture diagrams* for the specification of architecture styles. The presented results build on previous work on architecture modelling in BIP [9]. In particular, they are based on propositional interaction logic and connectors studied in [20, 21, 22, 91].

Table 1.1 presents the global picture of formalisms for architecture-based design in BIP. For modelling architectures and architecture styles, we distinguish between: 1) logic-based formalisms, i.e. interaction logics and configuration logics and 2) graphical formalisms, i.e. connectors and architecture diagrams. Depending on the features of the formalisms we can achieve different levels of expressiveness. Component variables are needed to describe generic models and properties, while variables over sets of components are needed to describe dynamic creation/deletion and dynamic configurations. The darker grey colour elements of the table indicate languages that are proposed in this thesis. The next subsections present in more detail our contributions.

1.3.1 Modelling and implementation of architectures

To answer the first fundamental question of “how does one model and implement architectures?”, we studied first-order interaction logic (FOIL) for modelling architectures and developed the JavaBIP tool that supports architecture-based software development. FOIL encompasses quantification over instances of component types and FOIL formulas characterise sets of interactions. The semantics of FOIL is defined via a satisfaction relation $\models_i$ between interactions and formulas. Given a FOIL formula, a feasible interaction is any set of ports characterised by a valuation which satisfies the formula.

JavaBIP—a Java adaptation of BIP [9]—allows coordination of existing concurrent software

Table 1.1 – Architecture-based design in BIP: the global picture and our contributions

	Modelling Architectures		Modelling Architecture Styles	
Formalism Features	Interaction Logics	Connectors	Configuration Logics	Architecture Diagrams
Fixed set of components	Propositional interaction logic	Static connectors	Propositional configuration logic	Architecture diagrams with constant parameters
Typed components, variables over components	First-order interaction logic	Generic connectors	First-order configuration logic	Interval architecture diagrams
Typed, ordered components, index variables	First-order interaction logic with ordered components	Index connectors	First-order configuration logic with ordered components	General architecture diagrams
Typed components, variables over components and sets of components	Second-order interaction logic	Dynamic connectors	Second-order configuration logic	Dynamic architecture diagrams

components in an exogenous manner, i.e. without requiring access to the components' source code. In order to better address the requirements of mainstream software engineering (as opposed to embedded software design), JavaBIP does not involve code generation; it rather relies on annotations of existing Java code and external configuration files. JavaBIP adheres to the architecture-based design and provides a strong separation of concerns. It separates architecture constraints from behaviour constraints. Such separation results in highly modular code, which allows software architects to reason compositionally and to reuse code implementing functionality and coordination independently.

List of contributions

- We studied a modelling methodology based on FOIL for writing architecture constraints associated with ports. For a port p, the associated constraint is decomposed into two types of constraints characterising interaction among ports: 1) a causal constraint which defines required ports and their cardinality; 2) an acceptance constraint which defines optional ports for participation.

- We designed and developed the JavaBIP engine, which is an extensible symbolic engine. The engine orchestrates the execution of components by deciding which component transitions must be executed at each cycle. It receives as input a set of constraints, resolves them on-the-fly by using constraint resolution techniques based on Binary Decision Diagrams (BDD) and notifies the components of the selected transitions.

1.3.2 Modelling architecture styles

To answer the second and third fundamental questions of "how to guide software developers in their choice of architectures?" and "how does one specify architecture styles?", we studied configuration logics and architecture diagrams for the specification of architecture styles. Configuration logic is a powerful and expressive declarative formalism. Architecture diagrams is an imperative, graphical formalism. Using architecture diagrams instead of purely logic-based specifications confers the usability advantages of graphical formalisms.

Configuration logics

Configuration logic formulas characterize interaction configurations between instances of *component types*. Configuration logic formulas are generated from formulas of the propositional interaction logic by using the operators union, intersection and complementation, as well as a *coalescing operator* $+$. To avoid ambiguity, we refer to the formulas of the configuration logic that syntactically are also formulas of the interaction logics as *interaction formulas*.

The semantics of the configuration logic is defined via a satisfaction relation $\models$ between configurations $\gamma = \{a_1, ..., a_n\}$ and formulas. An interaction formula f represents any configuration consisting of interactions satisfying it; that is $\gamma \models f$ if, for all $a \in \gamma$, $a \models_i f$. For set-theoretic operators, we take the standard meaning. The meaning of formulas of the form $f_1 + f_2$ is all configurations γ that can be decomposed into γ_1 and γ_2 ($\gamma = \gamma_1 \cup \gamma_2$) satisfying, respectively, f_1 and f_2. The formula $f_1 + f_2$ represents configurations obtained as the union of configurations of f_1 with configurations of f_2.

The following configuration logic formula characterizes the Mutual exclusion architecture style:

$$\begin{aligned}
&\exists m\!:\!MutexManager.\ \Sigma b\!:\!B.\ \Big((b.begin\ m.take)\ +\ (b.finish\ m.release)\Big) \wedge \\
&\forall m\!:\!MutexManager.\ \forall m'\!:\!MutexManager.\ (m = m') \wedge \\
&\forall m\!:\!MutexManager.\ \forall b\!:\!B.\ \forall b'\!:\!B\ (b' \neq b). \\
&\Big((\overline{b.begin\ b'.begin}\,) \wedge (\overline{b.finish\ b'.finish}\,) \wedge (\overline{b.begin\ b'.finish}\,) \wedge (\overline{m.take\ m.release}\,)\Big).
\end{aligned}$$

Architecture diagrams

Architecture diagrams allow the specification of generic coordination mechanisms based on the concept of *connector*. An architecture diagram consists of a set of *component types*, a *cardinality function* and a set of *connector motifs*. Component types are characterised by sets of *port types*. The cardinality function associates each component type with its *cardinality*, i.e. number of instances. Cardinality can be a fixed value or an interval. The architecture diagram of Figure 1.3 defines the Mutual exclusion architecture style. The cardinality of the

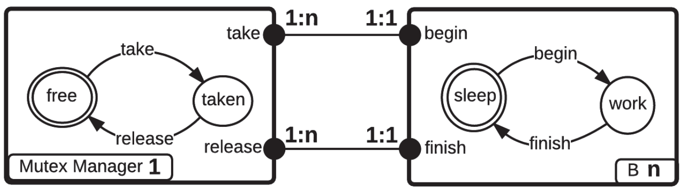


Figure 1.3 – Architecture diagram of the Mutual exclusion style

`Mutex Manager` component type is 1, while the cardinality of the `B` component is n.

Connector motifs are non-empty sets of port types. Each port type p in a connector motif has associated *multiplicity* and *degree* constraints, represented as a pair $m : d$. Multiplicity m specifies the number of port instances p_i of type p that are involved in each connector defined by the motif. Degree d specifies the number of connectors that each port instance p_i of type p is attached to. Multiplicities and degrees can be either fixed values or intervals. In Figure 1.3, the multiplicities of all port types are equal to 1, meaning that exactly one instance of each port type must be involved in each connector and thus, all connectors must be binary. The degrees require that each port instance of type `begin` and `finish` be attached to a single connector and each port instance of type `take` and `release` be attached to n connectors.

List of contributions

- We provide a full axiomatisation of the propositional configuration logic and a normal form similar to the disjunctive normal form in Boolean algebras. The existence of such a normal form implies the decidability of formula equality and satisfaction of a formula by an architecture model.

- To allow genericity of specifications, we studied higher-order extensions of the propositional configuration logic. Higher-order configuration logics allow to specify architecture styles for any number of components. First-order logic formulas involve quantification over component variables, while second-order logic formulas involve quantification over variables representing sets of components. Second-order logic is needed to express some interesting properties, e.g. the existence of cycles of interactions.

- We studied a hierarchy consisting of four classes of architecture diagrams: 1)*simple architecture diagrams*, where the cardinality, multiplicity and degree constraints are positive integers; 2) *index architecture diagrams*, where we introduce index variables and arithmetic predicates on index variables; 3) *interval architecture diagrams*, where the cardinality, multiplicity and degree constraints are intervals and 4) *general architecture diagrams*, which we obtained by merging 2) and 3) and additional arithmetic predicates on multiplicities and degrees.

- We present methods to assist software architects and developers to correctly specify architecture styles. In particular, we present a polynomial-time algorithm to check if an architecture model conforms to the architecture style specified by an architecture diagram. We also present a set of consistency conditions and a method that compositionally characterises all the architectures of an architecture diagram.

- We use configuration logics as a unified semantic framework. To this end, we studied a full encoding of architecture diagrams into configuration logics, thus, proving that configuration logics are at least as expressive as architecture diagrams.

- We illustrate the application of architecture diagrams in non-trivial case studies. We identified recurring patterns in on-board satellite software which we formalised with architecture diagrams to create a taxonomy of architecture styles. From the identified styles we generated architectures, which were used to develop satellite software that is correct-by-construction.

1.4 Thesis outline

The rest of the thesis is structured as follows:

Chapter 2 provides background information on the BIP framework, the propositional interaction logic, the notion of architectures in BIP and the architecture composition operator.

Chapter 3 presents the first-order interaction logic and its application in the JavaBIP framework. It describes the JavaBIP design work-flow, the formal component and coordination model underlying JavaBIP, the implementation details of the extensible symbolic engine and performance evaluation results. Related work on on-the-fly coordination mechanisms for software components is discussed. The obtained results is presented in [17, 18].

Chapter 4 presents the propositional configuration logic, its properties and the definition of a normal form which allows the decidability of equality of formulas and the satisfaction of a formula by an architecture. Additionally, Chapter 4 presents first-order and second-order extensions of the propositional logic. The use of configuration logics is illustrated through the specification of several architecture styles. Related work on declarative architecture languages is discussed. The obtained results is presented in [70, 73].

Chapter 5 presents the four classes architecture diagrams. It proposes a set of necessary and sufficient consistency conditions and a method that allows characterising compositionally the specified architectures. The chapter also presents a polynomial-time algorithm for checking that a given architecture conforms to an architecture diagram. We additionally present a full encoding of architecture diagrams into configuration logics and discuss related work on graphical architecture languages. The obtained results is presented in [71, 72].

Chapter 6 describes applications of architecture diagrams on specifying architecture styles identified in non-trivial case-studies of on-board satellite software. The obtained results is presented in [74].

Chapter 7 summarises the contributions of the thesis to the specification of architectures and architecture styles. The chapter also discusses directions for future work.

List of papers

The main results of this thesis were presented in the following papers (reference numbers coincide with the ones in the Bibliography section):

[17] Bliudze, S., Mavridou, A., Szymanek, R., & Zolotukhina, A. (2014). *Coordination of software components with BIP: application to OSGi.* 6th International Workshop on Modeling in Software Engineering (pp. 25-30). ACM.

[18] Bliudze, S., Mavridou, A., Szymanek, R., & Zolotukhina, A. (2016). *Exogenous Coordination of Concurrent Software Components with JavaBIP.* Submitted to: Software: Practice and Experience. Under review.

[70] Mavridou, A., Baranov, E., Bliudze, S., & Sifakis, J. (2015). *Configuration Logics: Modelling Architectures Styles.* 12th International Conference on Formal Aspects of Component Software. Lecture Notes in Computer Science 9539.

[71] Mavridou, A., Baranov, E., Bliudze, S., & Sifakis, J. (2016). *Architecture Diagrams: A Graphical Language for Architecture Style Specification.* 9th Interaction and Concurrency Experience. Electronic Proceedings in Theoretical Computer Science.

[72] Mavridou, A., Baranov, E., Bliudze, S., & Sifakis, J. (2016). *Configuration Logics and Architecture Diagrams for Modelling Architecture Styles.* Submitted to: Science of Computer Programming. Under review.

[73] Mavridou, A., Baranov, E., Bliudze, S., & Sifakis, J. (2017). *Configuration Logics: Modelling Architectures Styles.* Journal of Logical and Algebraic methods in Programming. 86(1):2 - 29.

[74] Mavridou, A., Stachtiari, E., Bliudze, S., Ivanov, A., Katsaros, P., & Sifakis, J. (2016). *Architecture-based Design: A Satellite On-board Software Case Study.* Accepted to the 13th International Conference on Formal Aspects of Component Software.

2 Preliminaries

In this chapter, we provide the formal component and coordination model underlying BIP. We present two different representations of the BIP interaction model by using 1) *algebra of connectors* and 2) *propositional interaction logic*. Connectors are most appropriate for graphical design and representation of interactions, whereas interaction logic formulas are most appropriate for manipulation and efficient encoding in the BIP Engine. We additionally present the notion of architectures in BIP and theoretical results on architecture composability.

The formal model of BIP has been used as the basis for the definition of the formal model of JavaBIP presented in Chapter 3. Algebra of connectors is necessary for the definition of architecture diagrams in Chapter 5. Propositional interaction logic serves as a basis for the definition of first-order interaction logic in Chapter 3 and propositional configuration logic in Chapter 4. Finally, the architecture composability results are used in the case-studies presented in Chapter 6.

2.1 BIP component framework

BIP [12] is a framework for component-based design of correct-by-construction applications. It provides a simple but powerful mechanism for coordination of concurrent components by superposing three layers: Behaviour, Interaction, and Priority. The first layer describes the behaviour of components as Finite State Machines (FSM) having transitions labelled with *ports* and extended with data stored in local variables. Ports form the interface of a component and are used to define its interactions with other components. They can also export part of the local variables, allowing access to the component's data.

The second layer defines component coordination by means of *interaction models*, i.e. sets of interactions. Interactions are sets of ports that define allowed synchronizations between components. An interaction model is defined in a structured manner by using connectors [20] or by using interaction logic [20, 21, 22], which is a Boolean algebra on the set of ports of the composed system. For each interaction, a connector also specifies how the data is retrieved, filtered and updated in each of the participating components. In particular, a Boolean guard can be associated to an interaction. The interaction is only enabled if the data provided by the components satisfies the guard [23]. In the third layer, priorities are used to impose scheduling constraints and to resolve conflicts when multiple interactions are enabled simultaneously.

The BIP component framework encompasses an expressive language and a dedicated tool-set. The BIP language offers primitives and constructs for modelling and composing layered components. The BIP tool-set comprises translators from various programming models into BIP, source-to-source transformers as well as compilers for generating code executable by dedicated engines. The tool-set also provides tools for deadlock detection, state reachability analysis and an interface with the nuXmv model checker.

The execution of a BIP system is driven by the BIP engine applying the following protocol in a cyclic manner:

1. upon reaching a state, each component notifies the engine about the possible outgoing transitions;
2. the engine chooses an interaction satisfying the input specifications, performs the data transfer and notifies all the involved components;
3. the notified components execute the functions associated with the transitions that were chosen by the engine at step 2.

2.2 Basic semantic model of BIP

We present the formal component model underlying the BIP framework focusing on the operational semantics of component interaction and priorities as defined in [46, 19].

Definition 2.2.1. For a set of ports P, an *interaction* is a non-empty subset $a \subseteq P$ of ports. To simplify notation we represent an interaction $\{p_1, p_2, \dots p_n\}$ as $p_1 p_2 \dots p_n$.

Behaviour

Definition 2.2.2. A *Finite State Machine* (FSM) given by a triple $(Q, P, \rightarrow)$, where Q is a set of *states*, P is a set of communication *ports*, and $\rightarrow \subseteq Q \times 2^P \times Q$ is a set of *transitions* labelled by sets of ports. For any pair of states $q, q' \in Q$ and an interaction $a \in 2^P$, we write $q \xrightarrow{a} q'$ iff $(q, a, q') \in \rightarrow$.

An interaction a is *enabled* in state q, denoted $q \xrightarrow{a}$, iff there exists $q' \in Q$ such that $q \xrightarrow{a} q'$. We abbreviate $q \overset{a}{\not\rightarrow} \overset{def}{=} \neg(q \xrightarrow{a})$. A port p is *active* iff it belongs to an enabled interaction.

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 [76, 50].

In BIP, a system can be obtained as the composition of n components, each modelled by FSMs $B_i = (Q_i, P_i, \to_i)$, for $i \in [1, n]$, such that their sets of ports are pairwise disjoint, i.e. $i \neq j$ implies $P_i \cap P_j = Q_i \cap Q_j = \emptyset$. We take $P \overset{def}{=} \bigcup_{i=1}^n P_i$, the set of all ports in the system.

In the rest of the thesis, we will drop the indices on transition relations and denote them by $\to$, whenever the indices are clear from the context.

To define operational semantics, we use Structural Operational Semantics (SOS) [85] rules. An SOS rule has the following form: $\dfrac{premises}{conclusion}$, where *premises* is a set of state predicates on the components composing the system, while *conclusion* is a state predicate on its global state. A rule is interpreted as follows: if all premises are satisfied, then the conclusion is satisfied.

Interaction

Definition 2.2.3. An *interaction model* is a set of interactions $\gamma \subseteq 2^P$. The component $\gamma(B_1, \ldots, B_n)$ is defined by the behaviour $(Q, P, \to_\gamma)$, with $Q = \prod_{i=1}^n Q_i$ and the transition relation $\to_\gamma$ inductively defined by 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 $I = \{ i \in [1, n] \,|\, a \cap P_i \neq \emptyset \}$.

Notice that an interaction $a \in \gamma$ is enabled in $\gamma(B_1, \ldots, B_n)$, only if, for each $i \in [1, n]$, the interaction $a \cap P_i$ is enabled in B_i; the states of components that do not participate in the interaction remain unchanged.

Priority

Several distinct interactions can be enabled at the same time, thus introducing non-determinism in the product behaviour. This can be restricted by means of priorities.

Definition 2.2.4. Given a system $B = \gamma(B_1, \ldots, B_n)$, a *priority model* π is a strict partial order on γ. We write $a \prec b$ as a shorthand for $(a, b) \in \pi$, which means that interaction a has a lower priority than interaction b.

For $B = (Q, P, \to)$ and a priority model π, the transition system $\pi(B) = (Q, P, \to_\pi)$ is

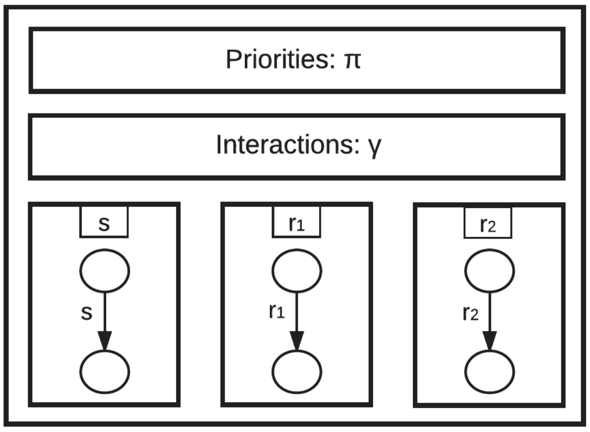


Figure 2.1 – A BIP system with three atomic components

defined by the rule

$$\frac{q \xrightarrow{a} q' \quad \left\{ q \overset{b}{\not\rightarrow} \,\middle|\, a \prec b \right\}}{q \xrightarrow{a}_{\pi} q'} . \tag{2.2}$$

An interaction is enabled in $\pi(B)$ only if it is enabled in B and is maximal according to π.

Example 2.2.5. Consider the component $\pi\gamma(S, R_1, R_2)$ shown in Figure 2.1 which is the composition of three atomic components: a sender S and two receivers R_1, R_2. The sender has a port s for sending messages and each receiver has a port r_i $(i = 1, 2)$ for receiving them. We consider the following four coordination schemes:

- *Rendezvous* ensures strong synchronization between S and all R_i. Rendezvous is specified by a single interaction involving all ports: $\gamma = \{sr_1r_2\}$. This interaction can occur only if all of the components are in states enabling transitions labelled, respectively, by s, r_1 and r_2.

- *Broadcast* allows all interactions involving S and any (possible empty) subset of R_i. Broadcast is specified by the set of all interactions containing s: $\gamma = \{s, sr_1, sr_2, sr_1r_2\}$. These interactions can only occur if S is in a state enabling s. Each R_i participates in an interaction only if it is in a state enabling r_i.

- *Atomic broadcast* ensures that either all or none of the receivers are involved in the interaction: $\gamma = \{s, sr_1r_2\}$. the s interaction can occur only if at least one of the receiving ports is not enabled. The sr_1r_2 interaction corresponds to a strong synchronization among the sender and all receivers.

- *Causal chain* ensures that for a message to be received by R_i, it has to be received at the same time by all R_j, for $i < i$. This coordination scheme is common in reactive systems.

For strong synchronization, the priority model is empty. For all other coordination schemes, whenever several interactions are possible, the interaction involving the maximal number of ports has the highest priority, i.e. $\pi = \{(a, a')|a \subset a'\}$. In BIP, this is known as the *maximal progress* rule.

2.3 Representations of the BIP interaction model

In this section, we recall the syntax and semantics of the different representations of BIP interaction models. Consider a family of components, indexed by I and equipped with sets of ports P_i, for $i \in I$, through which they can interact. Let $P = \cup_{i \in I} P_i$ be a set of ports of the system and assume that $0, 1 \notin P$.

2.3.1 Algebra of interactions

The *algebra of interactions* $\mathcal{AI}(P)$ was introduced in [20] as an auxiliary algebra to define the interaction semantics of the algebra of connectors presented in the Subsection 2.3.2. Each element in $\mathcal{AI}(P)$ should be considered as a set of possible interactions. The elements of this algebra can be bijectively mapped to interaction models, i.e. subsets of 2^P.

Syntax. The syntax of the $\mathcal{AI}(P)$ is defined by the following grammar

$$x \quad ::= \quad 0 \mid 1 \mid p \mid x \cdot x \mid x + x \,, \tag{2.3}$$

with $p \in P$ as an arbitrary port and +' and '$\cdot$' as the binary operators, respectively called *union* and *synchronisation*. Synchronisation binds stronger than union.

Semantics. The semantics of $\mathcal{AI}(P)$ is given by the function $\|\cdot\| : \mathcal{AI}(P) \to 2^{2^P}$, defined by

$$\begin{aligned} \|0\| \;=\; \emptyset, \quad \|1\| \;=\; \{\emptyset\}, \quad \|p\| \;&=\; \Big\{\{p\}\Big\}, \\ \|x_1 + x_2\| \quad &= \quad \|x_1\| \cup \|x_2\|, \\ \|x_1 \cdot x_2\| \quad &= \quad \Big\{a_1 \cup a_2 \,\Big|\, a_1 \in \|x_1\|, a_2 \in \|x_2\|\Big\}, \end{aligned} \tag{2.4}$$

for $p \in P$ and $x_1, x_2 \in \mathcal{AI}(P)$. Terms of $\mathcal{AI}(P)$ represent sets of interactions.

The corresponding equivalence relation on $\mathcal{AI}(P)$ is defined as follows: *two terms $x, y \in \mathcal{AI}(P)$ are* equivalent*, denoted by $x \simeq y$ iff* $\|x\| = \|y\|$. A sound and complete axiomatisation of $\mathcal{AI}(P)$ with respect to the semantic equivalence is provided in [19].

Example 2.3.1. In $\mathcal{AI}(P)$, the interaction sets of the four coordination schemes of Example 2.2.5 are as follows (also presented in the second column of Table 2.1):

- *Rendezvous* is represented by sr_1r_2;

- *Broadcast* is represented by $s(1+r_1)(1+r_2)$;
- *Atomic Broadcast* is represented by $s(1+r_1r_2)$;
- *Causal Chain* is represented by $s(1+r_1(1+r_2))$.

This representation is more compact and exhibits more information: e.g. the expression $(1+r_1)$ suggests that the port r_i is optional.

2.3.2 Algebra of connectors

The *algebra of connectors* $\mathcal{AC}(P)$ was introduced in [20], which formalises the concept of connector supported by the BIP language. Connectors can express complex coordination schemes combining synchronisation by rendezvous and broadcast.

Syntax. The syntax of $\mathcal{AC}(P)$ is defined by the following grammar

$$\begin{aligned} s &::= [0] \mid [1] \mid [p] \mid [x] && (\textit{synchrons}) \\ t &::= [0]' \mid [1]' \mid [p]' \mid [x]' && (\textit{triggers}) \\ x &::= s \mid t \mid x \cdot x \mid x + x\,, \end{aligned} \tag{2.5}$$

for $p \in P$, and where '+' is a binary operator called *union*, '·' is a binary operator called *fusion*, and brackets '[·]' and '[·]′' are unary *typing* operators. Fusion binds stronger than union.

Union has the same meaning as union in $\mathcal{AI}(P)$. Fusion is a generalisation of the synchronisation in $\mathcal{AI}(P)$. Typing is used to form typed connectors: '[·]' defines *synchrons* (Figure 2.2a) which require synchronisation with other ports in order to interact and '[·]′' defines *triggers* (Figure 2.2a) which can initiate an interaction.

In order to simplify notation, we will omit brackets on 0, 1, and ports $p \in P$, as well as '·' for the fusion operation.

Semantics. The semantics of $\mathcal{AC}(P)$ is given by the function $|\cdot| : \mathcal{AC}(P) \to \mathcal{AI}(P)$ (we use the $\sum$ and $\prod$ notation for the union and fusion of multiple terms of $\mathcal{AC}(P)$):

$$|[p]| = p\,, \qquad |x_1 + x_2| = |x_1| + |x_2|\,, \qquad \left|\prod_{i=1}^n [x_i]\right| = \prod_{i=1}^n |x_k|\,, \tag{2.6}$$

$$\left|\prod_{i=1}^n [x_i]' \prod_{j=1}^m [y_j]\right| = \sum_{i=1}^n |x_i| \left(\prod_{k \neq i} \Big(1 + |x_k|\Big) \ \prod_{j=1}^m \Big(1 + |y_j|\Big)\right) \tag{2.7}$$

for $n > 0$, $m \geq 0$, $x_1, \ldots, x_n, y_1, \ldots, y_m \in \mathcal{AC}(P)$ and $p \in P \cup \{0, 1\}$.

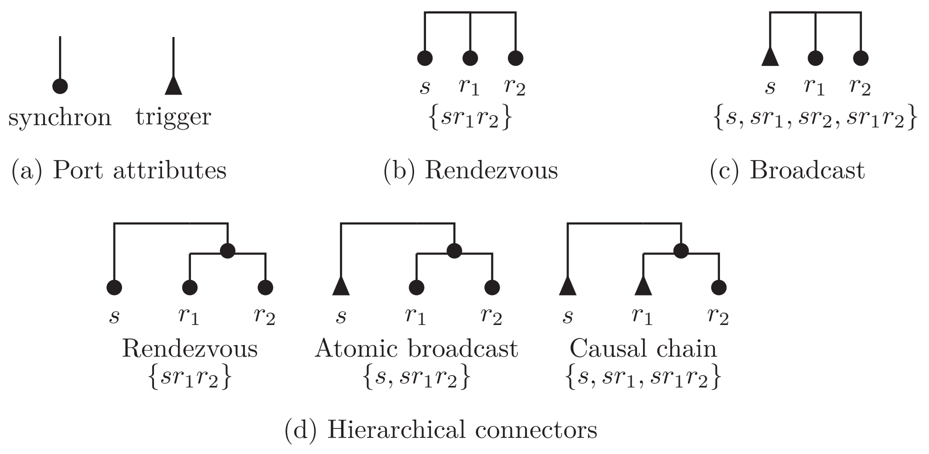


Figure 2.2 – BIP connectors: below each connector, we show the set of interactions it defines

A complete axiomatisation of $\mathcal{AC}(P)$ with respect to the semantic equivalence is provided in [20].

Example 2.3.2. The interaction sets of the four coordination schemes of Example 2.2.5 are represented in $\mathcal{AC}(P)$ as follows (also presented in the third column of Table 2.1):

- *Rendezvous* is represented by sr_1r_2, where all connected ports are synchrons (Figure 2.2b);
- *Broadcast* is represented by $s'r_1r_2$, where there is at least one trigger (Figure 2.2c);
- *Atomic Broadcast* is represented by $s'[r_1r_2]$ which corresponds to a hierarchical connector (Figure 2.2d);
- *Causal Chain* is represented by $s'[r_1'r_2]$ which corresponds to a hierarchical connector (Figure 2.2d).

$\mathcal{AC}(P)$ allows compact representation of interactions and, moreover, explicitly captures the difference between broadcast and rendezvous.

2.3.3 Propositional interaction logic

The propositional interaction logic (PIL), studied in [20, 21], is a Boolean logic used to characterize the interactions between components.

Syntax. The propositional interaction logic is defined by the grammar:

$$\phi ::= true \mid p \mid \overline{\phi} \mid \phi \vee \phi\,, \qquad \text{with any } p \in P. \tag{2.8}$$

Conjunction is defined as follows: $\phi_1 \wedge \phi_2 \overset{def}{=} \overline{(\overline{\phi_1} \vee \overline{\phi_2})}$. Implication is defined as follows: $\phi_1 \Rightarrow \phi_2 \overset{def}{=} \overline{\phi_1} \vee \phi_2$. To simplify the notation, we omit conjunction in monomials, e.g. writing sr_1r_2 instead of $s \wedge r_1 \wedge r_2$.

Semantics. The meaning of a PIL formula ϕ is defined by the following satisfaction relation. For an interaction $a \subseteq P$ we define $a \models_i \phi$ iff ϕ evaluates to *true* for the valuation $p = true$, for all $p \in a$ and $p = false$, for all $p \notin a$. We denote by $|\phi| \overset{def}{=} \sum_{a \models_i \phi} a$ the union (from $\mathcal{AI}(P)$) of the interactions satisfying ϕ. Thus we have $|\cdot| : PIL(P) \to \mathcal{AI}(P)$, where $PIL(P)$ is the Boolean algebra over the set of port variables P.

The operators meet the usual Boolean axioms.

An interaction a can be associated to a *characteristic monomial* $m_a = \bigwedge_{p \in a} p \wedge \bigwedge_{p \notin a} \overline{p}$ such that $a' \models_i m_a$ iff $a' = a$.

Table 2.1 – $\mathcal{AI}(P)$, $\mathcal{AC}(P)$ and $PIL(P)$ representations of four basic coordination mechanisms

Coordination mechanism	$\mathcal{AI}(P)$	$\mathcal{AC}(P)$	$PIL(P)$
Rendezvous	sr_1r_2	sr_1r_2	sr_1r_2
Broadcast	$s(1+r_1)(1+r_2)$	$s'r_1r_2$	s
Atomic broadcast	$s(1+r_1r_2)$	$s'[r_1r_2]$	$s\overline{r_1}\,\overline{r_2} \vee sr_1r_2$
Causal chain	$s(1+r_1(1+r_2))$	$s'[r_1'r_2]$	$s\overline{r_1}\,\overline{r_2} \vee sr_1\overline{r_2} \vee sr_1r_2$

Example 2.3.3. The coordination schemes of Example 2.2.5 can be expressed in PIL as follows (also presented in the fourth column of Table 2.1) :

- *Rendezvous* is represented by the monomial sr_1r_2;
- *Broadcast* is represented by the formula s, which can be expanded to $s\overline{r_1}\,\overline{r_2} \vee sr_1\overline{r_2} \vee s\overline{r_1}r_2 \vee sr_1r_2$;
- *Atomic broadcast* can be characterised by the formula $s\overline{r_1}\,\overline{r_2} \vee sr_1r_2$;
- *Causal chain* can be characterised by the formula $(r_1 \Rightarrow r_2) \wedge (r_1 \Rightarrow s)$, which can be expanded to $s\overline{r_1}\,\overline{r_2} \vee sr_1\overline{r_2} \vee sr_1r_2$.

Definition 2.3.4. For an interaction model $\gamma \subseteq 2^P$ over a set of ports P, its *characteristic predicate* is a disjunction of characteristic monomials for all interactions in γ:

$$\phi_\gamma = \bigvee_{a \in \gamma} \Big(\bigwedge_{p \in a} p \wedge \bigwedge_{p \notin a} \overline{p} \Big).$$

A predicate ϕ uniquely defines an interaction model γ_ϕ such that $\|\phi\| = \gamma_\phi$.

2.4 Architectures in BIP

A formal notion of architectures was proposed in [6], based on BIP [9]. An architecture can be viewed as a BIP model, where some of the atomic components are considered as *coordinators*, while the rest are *parameters*. When an architecture is applied to a set of components, these components are used as *operands* to replace the parameters of the architecture. Architectures generalise the BIP interaction model by introducing stateful coordinator 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 belong to operand components.

Definition 2.4.1. An *architecture* is a tuple $A = (\mathcal{C}, P_A, \gamma)$, where $\mathcal{C}$ is a finite set of *coordinating components* with pairwise disjoint sets of ports, P_A is a set of ports, such that $\bigcup_{C\in\mathcal{C}} P_C \subseteq P_A$, and $\gamma \subseteq 2^{P_A}$ is an interaction model over P_A.

An architecture A can be applied to any set of components $\mathcal{B}$ that contains all the dangling ports of A. Intuitively, an architecture enforces coordination constraints on the components in $\mathcal{B}$. The interface P_A of an architecture A contains all ports of the coordinating components $\mathcal{C}$ and some additional ports, which must belong to the components in $\mathcal{B}$. In the application $A(\mathcal{B})$, the ports belonging to P_A can only participate in the interactions defined by the interaction model γ of A. Ports that do not belong to P_A are not restricted and can participate in any interaction.

Definition 2.4.2. Let $A = (\mathcal{C}, P_A, \gamma)$ be an architecture and let $\mathcal{B}$ be a set of components, such that $\bigcup_{B\in\mathcal{B}} P_B \cap \bigcup_{C\in\mathcal{C}} P_C = \emptyset$ and $P_A \subseteq P \overset{def}{=} \bigcup_{B\in\mathcal{B}\cup\mathcal{C}} P_B$. The *application of an architecture* A to the components $\mathcal{B}$ is the component

$$A(\mathcal{B}) \overset{def}{=} \Big(\gamma \bowtie 2^{P\setminus P_A}\Big)(\mathcal{C}\cup\mathcal{B})\,, \tag{2.9}$$

where, for interaction models γ' and γ'' over disjoint domains P' and P'' respectively,

$$\gamma' \bowtie \gamma'' \overset{def}{=} \{a' \cup a'' \,|\, a' \in \gamma', a'' \in \gamma''\}$$

is an interaction model over $P' \cup P''$.

Notice that, when the interface of the architecture covers all ports of the system, i.e. $P = P_A$, we have $2^{P\setminus P_A} = \{\emptyset\}$ and the only interactions allowed in $A(\mathcal{B})$ are those belonging to γ.

Example 2.4.3. Consider the components B_1 and B_2 in Figure 2.3(a). In order to ensure mutual exclusion of their `work` states, we apply the architecture $A_{12} = (\{C_{12}\}, P_{12}, \gamma_{12})$, where C_{12} is shown in Figure 2.3(b), $P_{12} = \{b_1, b_2, b_{12}, f_1, f_2, f_{12}\}$, $\gamma_{12} = \{\emptyset,\ b_1b_{12}, b_2b_{12},\ f_1f_{12},\ f_2f_{12}\}$.

The interface P_{12} of A_{12} covers all ports of B_1, B_2 and C_{12}. Hence, the only possible interactions are those explicitly belonging to γ_{12}. Assuming that the initial states of B_1 and B_2 are `sleep`, and that of C_{12} is `free`, neither of the two states (`free`, `work`, `work`) and

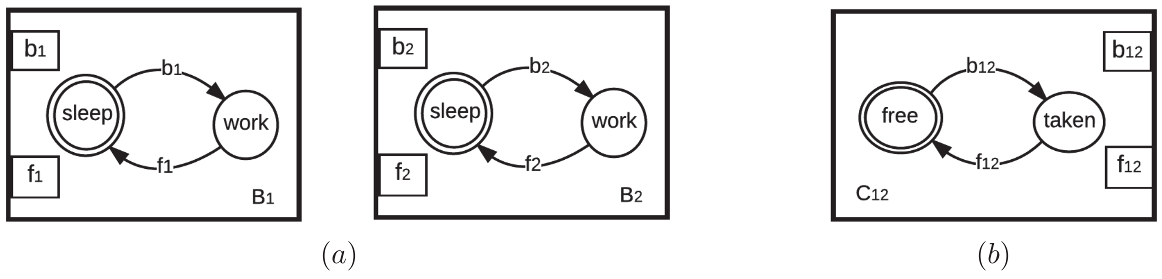


Figure 2.3 – Components (a) and coordinator (b) for Example 2.4.3

$(\mathtt{taken}, \mathtt{work}, \mathtt{work})$ is reachable, i.e. the mutual exclusion property $(q_1 \neq \mathtt{work}) \vee (q_2 \neq \mathtt{work})$—where q_1 and q_2 are state variables of B_1 and B_2 respectively—holds in $A_{12}(B_1, B_2)$.

Let B_3 be a third component, similar to B_1 and B_2, with the interface $\{b_3, f_3\}$. Since $b_3, f_3 \notin P_{12}$, the interaction model of the application $A_{12}(B_1, B_2, B_3)$ is $\gamma_{12} \bowtie \{\emptyset, b_3, f_3\}$. (We omit the interaction $b_3 f_3$, since b_3 and f_3 are never enabled in the same state and, therefore, cannot be fired simultaneously.) Thus, the component $A_{12}(B_1, B_2, B_3)$ is the unrestricted product of the components $A_{12}(B_1, B_2)$ and B_3. The application of A_{12} enforces mutual exclusion between the `work` states of B_1 and B_2, but does not affect the behaviour of B_3.

Architectures can be intuitively understood as enforcing constraints on the global state space of the system [22, 101]. More precisely, component coordination is realized by limiting the allowed interactions, thus enforcing constraints on the transitions components can take. From this perspective, architecture composition can be understood as the conjunction of their respective constraints. This intuitive notion is formalized by the following definition.

Definition 2.4.4. Let $A_i = (\mathcal{C}_i, P_{A_i}, \gamma_i)$, for $i = 1, 2$ be two architectures and let $\varphi_{\gamma_1}, \varphi_{\gamma_2}$ be characteristic predicates (Definition 2.3.4) of γ_1, γ_2, respectively. The *composition* of A_1 and A_2 is an architecture $A_1 \oplus A_2 = (\mathcal{C}_1 \cup \mathcal{C}_2, P_{A_1} \cup P_{A_2}, \gamma_\varphi)$, where $\varphi = \varphi_{\gamma_1} \wedge \varphi_{\gamma_2}$ and $\gamma_\varphi = \|\varphi\|$ is an interaction model defined by the predicate φ.

The architecture composition operator is commutative, associative and idempotent if all coordinating components are deterministic [6]. Architecture composition preserves the safety properties enforced by the individual architectures, while liveness properties are preserved under an additional non-interference assumption [6].

3 Interaction logic and JavaBIP

In most mainstream programming languages, including Java and C++, basic coordination primitives are implemented as built-in features of the language [103, 65]. Different variations of locks, semaphores and monitors are used to express coordination constraints. However, these low-level primitives are mixed up with the functional code, forcing developers to keep both aspects simultaneously in mind—not only at design time, but also during debugging and maintenance. Since in concurrent environments it is practically infeasible to envision all possible execution scenarios, synchronization errors can result in race conditions and deadlocks.

Furthermore, an observable trend in software engineering is the increasing utilisation of *declarative design techniques*. Developers provide specifications of *what* must be achieved, rather than *how* this must be achieved. These specifications are then interpreted by engines, which generate—often on the fly—the corresponding software entities. It is not always possible to instrument or even access the actual source code. Even if the code is generated explicitly, it is usually not desirable to modify it, since this can lead to a considerable increase of maintenance costs. This precludes the use of low-level primitives for the coordination of concurrent components that have to be reusable in different configurations and assemblies. Therefore, the problem of coordinating concurrent components calls for an exogenous solution that would allow software developers to think on a higher level of abstraction, separating functional and coordination aspects.

In this chapter, we present the JavaBIP framework, which provides strong separation of concerns and raises the abstraction level by *integrating architectures into mainstream software development*. For component coordination, JavaBIP provides two primitive mechanisms: 1) multi-party synchronization of component transitions and 2) asynchronous event notifications. JavaBIP follows an exogenous approach without requiring access to the source code of the coordinated components; instead, it relies exclusively on existing APIs and external specifications files. The latter include architecture constraints, written in a macro-notation based on first-order interaction logic, that define the possible synchronizations among actions of different components. Such a separation, between functionality and coordination, results

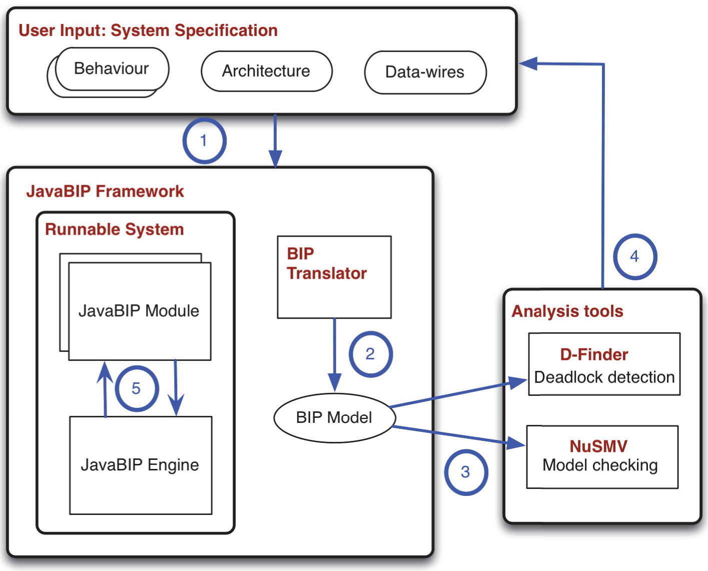


Figure 3.1 – JavaBIP design workflow

in modular code, which allows developers to reason compositionally and reuse code. Furthermore, JavaBIP components do not carry coordination logic that relies on the characteristics of any specific execution environment.

3.1 JavaBIP design workflow

JavaBIP—a Java adaptation of BIP [12]—relies on the following observations. Domain specific components have states (e.g. idle, working, suspended) that are known to component users with domain expertise. Furthermore, components always provide APIs that allow programs to invoke operations (e.g. suspend or resume) in order to change their state, or to be notified when the component changes the state spontaneously. Thus, component behaviour can be represented by FSM (Definition 2.2.2). An FSM has a finite set of states and a finite set of transitions between these states. Transitions are associated with calls to API functions, which force a component to take an action, or with event notifications that allow reacting to external events coming from the environment.

Figure 3.1 shows the steps of the JavaBIP design flow. In the step ①, the *system specification* is designed, consisting of the following annotated Java classes and XML configuration files:

- A *behaviour specification* for each component, given by an FSM extended with ports and data. This is provided as an annotated Java class, whereof the methods can call APIs provided by the coordinated components.
- The *architecture specification*, which is the interaction model of the system that specifies how the transitions of different components must be synchronized. This is specified in

a macro-notation that allows compact and high-level expression and is provided as an XML configuration file. The semantics of the macro-notation is expressed in first-order interaction logic.

- Optionally, data transfer can be defined, by providing the *data-wire specification* for each data variable of every component. Data wires specify which data are exchanged between components and are provided as an XML configuration file.

The optional analysis loop starts in step ②, where the system specification is automatically translated into an equivalent model of the system expressed in the BIP language. This model can then be verified for deadlock freedom or other safety properties, using DFinder [10], ESST or nuXmv [16] (step ③). Other analyses can be performed using any tool for which a model transformation from BIP is available. If the required safety properties are not satisfied by the model, the specification can be refined by the developers (step ④) and analysed anew. Finally, when developers are satisfied with the design, the refined specification can be executed (step ⑤). Steps ②, ③, and ④ are optional and can be repeated several times.

As shown in Figure 3.1, the *runnable system* consists of two major parts: the *engine* and several *modules*, one for each component to be coordinated. Each module is composed of a dedicated executor, a behaviour specification and the corresponding functional code of the component (Figure 3.8). The executor has access to the behaviour specification of the component and uses it to drive the module execution. The behaviour specification of each component along with the architecture and data-wire specifications are provided to the engine. The engine orchestrates the overall execution of the system by deciding which component transitions must be executed at each cycle. It then notifies the executors of the selected transitions and they make the corresponding calls to the functional code. The execution protocol of the JavaBIP engine is the same as the one described in Section 2.1.

3.2 JavaBIP by example: Camel routes

We present a motivating example illustrating the modelling concepts of JavaBIP. FSM transitions 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 environment. 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 its state based on internal information—when enabled, they are executed immediately. Spontaneous and internal transitions cannot be used for synchronization with other components. In Figure 3.2, the initial states of all FSMs are denoted by a double circle; Boolean guards on transitions are shown in square brackets; enforceable transitions are shown with solid arrows and spontaneous transitions with dashed arrows.

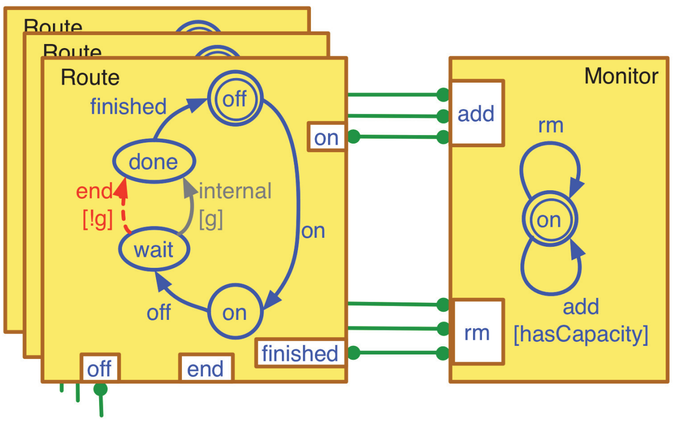


Figure 3.2 – JavaBIP models of three routes and a monitor

A Camel route [98] transfers data among a number of data sources. 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 `resumeRoute` and `suspendRoute` to control the activation of a route. For simplicity, we assume here that the memory used by an active route is known, whereas the memory used by a suspended route is negligible.

Consider the Route and Monitor models, shown in Figure 3.2. 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. The behaviour specifications of the Route and Monitor component types are shown in Figures 3.3 and 3.4, respectively. The precise syntax is explained in Section 3.4.1.

The Route model, shown in the left-hand side of Figure 3.2, has four states: `off`, `on`, `wait` and `done`. Its initial state is `off`. When the route is at state `off`, it can start working by executing the `on` transition. Similarly, when the route is at state `on`, it can suspend its work by executing the `off` transition. The `on` and `off` transitions are both enforceable (Figure 3.3: lines 3–4) and are associated with the `resumeRoute` and `suspendRoute` methods of the Camel API (Figure 3.3: lines 19–27).

Following the call to `suspendRoute` associated with the transition `off`, the route moves to the state `wait`. At this point, if the route has finished processing the previous data batch, it can be suspended immediately—represented by the internal transition to the `done` state (Figure 3.3: lines 32–33). Otherwise, the internal transition is disabled. Instead, to move to state `done`, the route has to wait for the processing termination event, associated with the spontaneous transition `end` (Figure 3.3: lines 29–30). The guard `g` (Figure 3.3: lines 38–42) is used to check whether the route has finished processing.

```java
@Ports({
  @Port(name = "end", type = PortType.spontaneous),
  @Port(name = "on",  type = PortType.enforceable),
  @Port(name = "off", type = PortType.enforceable),
  @Port(name = "finished", type = PortType.enforceable)
})
@ComponentType(initial = "off", name = "Route")
public class Route implements CamelContextAware {

  private CamelContext camelContext;
  private String routeId;
  private int deltaMemory = 100; // Dummy value, for the sake of simplicity

  public Route(String routeId, CamelContext camelContext) {
    this.routeId = routeId;
    this.camelContext = camelContext;
  }

  @Transition(name = "on", source = "off", target = "on")
  public void startRoute() throws Exception {
    camelContext.resumeRoute(routeId);
  }

  @Transition(name = "off", source = "on", target = "wait")
  public void stopRoute() throws Exception {
    camelContext.suspendRoute(routeId);
  }

  @Transition(name = "end", source = "wait", target = "done", guard = "!g")
  public void spontaneousEnd() {} // "!g" in the guard above means "not g"

  @Transition(name = "", source = "wait", target = "done", guard = "g")
  public void internalEnd() {}

  @Transition(name = "finished", source = "done", target = "off")
  public void finishedTransition() {}

  @Guard(name = "g")
  public boolean isFinished() {
    return camelContext.getInflightRepository().
      size(camelContext.getRoute(routeId).getEndpoint()) == 0;
  }

  @Data(name = "deltaMemoryOnTransition",
    accessTypePort = AccessType.allowed, ports = { "on", "finished" })
  public int deltaMemoryOnTransition() {
    return deltaMemory;
  }
}
```

Figure 3.3 – Annotations for the Route component type

```
 1 @Ports({
 2  @Port(name = "add", type = PortType.enforceable),
 3   @Port(name = "rm", type = PortType.enforceable)
 4 })
 5
 6 @ComponentType(initial = "on", name = "MemoryMonitor")
 7 public class MemoryMonitor {
 8
 9   final private int memoryLimit;
10   private int currentCapacity = 0;
11
12   public MemoryMonitor(int memoryLimit) {
13     this.memoryLimit = memoryLimit;
14   }
15
16   @Transition(name = "add", source = "on", target = "on",
17               guard = "hasCapacity")
18   public void addRoute(@Data("memoryUsage") Integer deltaMemory) {
19     currentCapacity += deltaMemory;
20   }
21
22   @Transition(name = "rm", source = "on", target = "on")
23   public void removeRoute(@Data(name="memoryUsage") Integer deltaMemory) {
24     currentCapacity -= deltaMemory;
25   }
26
27   @Guard(name = "hasCapacity")
28   public boolean hasCapacity(@Data("memoryUsage") Integer memoryUsage) {
29     return currentCapacity + memoryUsage < memoryLimit;
30   }
31 }
```

Figure 3.4 – Annotations for the Monitor component type

The Monitor model, shown in the right-hand side of Figure 3.2, has only one state and two enforceable transitions: `add` (Figure 3.4: lines 16–20) for adding running routes and `rm` (Figure 3.4: lines 22–25) for removing them. The `add` transition has the guard `hasCapacity` (Figure 3.4: lines 27–30) that checks whether the available memory limit of the system, defined through the constructor of the MemoryMonitor class (Figure 3.4: lines 12–14), is sufficient for adding more running routes.

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. The port `on` of each route component must synchronize with the port `add` of the monitor. This means that when a route component is executing the `on` transition, the monitor component must execute the `add` transition simultaneously. 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. Similarly, transition `finished` of each route must be synchronized with the transition `rm` of the monitor.

At each execution cycle, the monitor decides whether there is sufficient amount of memory in the system to add another route. To do so, data is exchanged between the non-running routes and the monitor. Every route has a value of how much memory it consumes if it resumes working (Figure 3.3: lines 44–48) and sends this to the monitor. The monitor then decides by computing the `hasCapacity` guard value and informs the JavaBIP engine of its decision. Data exchange between the routes and the monitor happens if the `on-add` synchronization can be executed, i.e. some of the routes are at state `off`. Additionally, data exchange occurs before the `finished-rm` synchronization is executed: the corresponding route tells the monitor how much memory it will release upon suspending its execution.

3.3 Theoretical foundations

The formal model underlying the JavaBIP framework extends the BIP formal model (Section 2.2) by considering three types of transitions:

- *enforceable* transitions represent the controllable behaviour of the component;
- *spontaneous* transitions represent changes in the environment that affect the component behaviour, but cannot be controlled;
- *internal* transitions represent computations independent of the component environment.

We consider a system of components, each represented by a Finite State Machine (FSM) extended with ports and data. An FSM is specified by its states and the guarded transitions between them. Each transition has a function and a port associated to it. In general, one port can be associated to several transitions. However, the FSM is deterministic in the following sense: there cannot be two simultaneously enabled transitions leaving the same state that are both internal or both labelled by the same port.

For the sake of clarity, we will first present the model without data and then extend it by introducing data variables and data-wires used to transfer data among the components.

3.3.1 Component model without data

Definition 3.3.1. A *component* is a Finite State Machine (FSM) given by a quadruple $B = (Q, P^e, P^s, \rightarrow)$, where:

- Q is a finite set of states,
- P^e and P^s are disjoint finite sets of, respectively, enforceable and spontaneous ports,
- $\rightarrow \; \subseteq Q \times P \times Q$, with $P = P^e \cup P^s \cup \{\tau\}$ and $\tau \notin P^e \cup P^s$, is a transition relation, where the special symbol τ labels internal transitions.

Below, we will use the common notation, writing $q \xrightarrow{p} q'$ as a shorthand for $(q, p, q') \in \rightarrow$.

Remark 3.3.2. For the practical implementation, we require that all the components be deterministic, i.e. such that, for any $q \in Q$ and any $p \in P$, there is at most one outgoing transition from q labelled by p or, formally, $|\{q' \in Q \,|\, (q, p, q') \in \rightarrow\}| \leq 1$. While this additional assumption does not have any impact on the theoretical foundations of the JavaBIP framework, as presented in this section, it noticeably simplifies the implementation.

Let $B_i = (Q_i, P_i^e, P_i^s, \rightarrow)$, for $i \in [1, n]$, be a set of components with pairwise disjoint sets $P_i = P_i^e \cup P_i^s$, i.e. $P_i \cap P_j = \emptyset$, for all $i \neq j$. We denote $P^e = \bigcup_{i=1}^n P_i^e$ and $P^s = \bigcup_{i=1}^n P_i^s$, respectively, the sets of all enforceable and spontaneous ports in the system.

A JavaBIP interaction contains only enforceable ports.

Definition 3.3.3. An interaction is a non-empty subset of enforceable ports $a \subseteq P^e$, such that $|a \cap P_i^e| \leq 1$, for all $i \in [1, n]$, labelling the transitions to be synchronised.

We require that any interaction contain at most one port from any given component, reflecting the fact that a component can only execute one transition at a time. An *interaction model* is the set of interactions that are allowed in the system. Let $\gamma \subseteq 2^{P^e} \setminus \{\emptyset\}$ be such a set of interactions.

Definition 3.3.4. The composed component is defined as the tuple $\gamma(B_1, \dots, B_n) = (Q, \gamma, P^s, \rightarrow)$, where $Q = \prod_{i=1}^n Q_i$ is the Cartesian product of the sets of states of the individual components and $\rightarrow$ is the set of transitions defined as follows:

- spontaneous port

$$\frac{p \in P^s \qquad q_i \xrightarrow{p} q_i'}{q_1 \dots q_i \dots q_n \xrightarrow{p} q_1 \dots q_i' \dots q_n}, \quad \text{for any } i \in [1, n]; \tag{3.1}$$

- internal transition

$$\frac{q_i \xrightarrow{\tau} q_i'}{q_1 \dots q_i \dots q_n \xrightarrow{\tau} q_1 \dots q_i' \dots q_n}, \quad \text{for any } i \in [1, n]; \tag{3.2}$$

- interaction

$$\frac{a \in \gamma \quad \left\{q_i \xrightarrow{a \cap P_i^e} q_i' \,\middle|\, i \in I\right\} \quad \left\{q_i = q_i' \,\middle|\, i \notin I\right\}}{q_1 \dots q_n \xrightarrow{a}_\gamma q_1' \dots q_n'}, \tag{3.3}$$

  where $I = \{i \in [1, n] \,|\, a \cap P_i^e \neq \emptyset\}$.

An interaction $a \in \gamma$ is enabled in $\gamma(B_1, \dots, B_n)$, only if, for each component C_i participating in a—that is, such that $a \cap P_i^e = \{p_i\}$, for some $p_i \in P_i^e$—the port p_i is enabled in C_i. Notice that the states of components that do not participate in the interaction remain unchanged.

3.3.2 Extension of the model with data

Data transfer is essential for allowing information flow between components. For each component, we consider two types of data:

- *input data* that the component receives from its environment (including other components),
- *output data* that the component provides to other components.

To each type of data, we associate the corresponding set of variables. For simplicity, we assume in this section that all variables have the same domain $\mathcal{D}$. In our implementation, variables can have any type that can be defined in a Java program.

In the presence of data, the state of a component is defined by the combination of the control location and the valuation of its output variables. Since the input data is provided by the environment, it does not contribute to the state of the component. This is analogous to the combination of the program counter and the valuation of variables in the semantics of programming languages. In the absence of data (Section 3.3.1), the state and the control location coincide. Therefore, for the sake of presentation uniformity, we will continue referring as “state” to the control location of the FSM underlying the component; we will refer as “complete state” to the combination of the control location and the valuation of output variables.

Let us first introduce some additional notation. Assume that X^{in} and X^{out} are two given sets of, respectively, input and output variables of a component. We denote

$\mathbb{B}[X^{in}, X^{out}]$,	the set of Boolean predicates on input and output variables,
$\mathbb{E}[X^{in}, X^{out}]$,	the set of assignment expressions of the form $Y := e(X)$, with $X \subseteq X^{in} \cup X^{out}$ and $Y \subseteq X^{out}$.

For a guard $g \in \mathbb{B}[X^{in}, X^{out}]$ or an expression $f \in \mathbb{E}[X^{in}, X^{out}]$, we denote $in(g), in(f) \subseteq X^{in}$ and $out(g), out(f) \subseteq X^{out}$ the corresponding sets of input and output variables used in g and f.

Definition 3.3.5. A *component with data* is a tuple $B = (Q, P^e, P^s, X^{in}, X^{out}, d, \rightarrow\ )$, where:

- Q is the finite set of states,
- P^e and P^s are disjoint finite sets of, respectively, enforceable and spontaneous ports,
- X^{in} and X^{out} are disjoint finite sets of, respectively, input and output variables,
- $d : P^e \rightarrow X^{out}$ is a *data access mapping*, associating to each enforceable port $p \in P^e$ the list of output variables that are accessible through p;

- $\rightarrow \; \subseteq Q \times P \times \mathcal{G} \times \mathcal{F} \times Q$ is a set of transitions (as above, we write $q \xrightarrow{p,g,f} q'$ as a shorthand for $(q,p,g,f,q') \in \rightarrow$), where
  - $\mathcal{G} = \mathbb{B}[X^{in}, X^{out}]$ is the set of *guards*,
  - $\mathcal{F} = \mathbb{E}[X^{in}, X^{out}]$ is the set of *update expressions*
  - $P = P^e \cup P^s \cup \{\tau\}$ with $\tau \notin P^e \cup P^s$;

  such that $in(f) = in(g) = \emptyset$, for any internal transition $q \rightarrow \tau, g, fq'$.

Remark 3.3.6. Similarly to Remark 3.3.2, in the practical implementation, we require that all the components be deterministic. Formally, for components with data, we require that for any $q \in Q$, $p \in P$ and any valuation of variables $v : X^{in} \cup X^{out} \rightarrow \mathcal{D}$, we have $\big|\{(g,f,q') \in \mathcal{G} \times \mathcal{F} \times Q \,|\, q \rightarrow p, g, fq',\ g(v) = true\}\big| \leq 1$.

Notice that in a component with data, there can be several outgoing transitions in the same state, labelled with the same port. However, we require that at most one of them be enabled—meaning that its guard evaluates to *true*—with any valuation of the component variables.

The input variables do not have persistent values—they represent the arguments of transition guards and update expressions, and have to be assigned a value provided by the environment every time when the corresponding guard or update expression must be evaluated. Hence, indeed, the sets X^{in} and X^{out} are disjoint.

As mentioned above, at any given execution cycle, the complete state of the component is formed by the combination of its current control location and the valuation of its output variables. The component can change its complete state by executing a transition only if its environment provides all the necessary inputs—i.e. the input data required by the guard and the update expression of the transition—and the guard is satisfied. Upon executing the transition, the values of the output variables are modified using the update expression. Notice that the values of the output variables that a component provides are those before the execution of the transition. Furthermore, these need not be the same as the output variables updated by the transition. In other words, for a transition $q \xrightarrow{p,g,f} q'$, with $p \in P^e$, there is no *a priori* relation between $d(p)$ and $out(f)$.

We denote $\widetilde{B} = (Q, P^e, P^s, \xrightarrow{da})$, with $q \xrightarrow{da} pq'$ iff $q \xrightarrow{p,g,f} q'$, for some $g \in \mathcal{G}$ and $f \in \mathcal{F}$, the component obtained by abstracting the data from the component with data $B = (Q, P^e, P^s, X^{in}, X^{out}, d, \rightarrow)$.

Let $B_i = (Q_i, P_i^e, P_i^s, X_i^{in}, x_i^{out}, d_i \rightarrow)$, for $i \in [1,n]$, be a set of components with data with pairwise disjoint sets of ports $P_i = P_i^e \cup P_i^s$ and variables $X_i = \cup X_i^{in} \cup X_i^{out}$, i.e. $P_i \cap P_j = X_i \cap X_j = \emptyset$, for all $i \neq j$. We denote $P^e = \bigcup_{i=1}^n P_i^e$ and $P^s = \bigcup_{i=1}^n P_i^s$, respectively, the sets of all enforceable and spontaneous ports in the system; $X^{in} = \bigcup_{i=1}^n X_i^{in}$ and $X^{out} = \bigcup_{i=1}^n X_i^{out}$, respectively, the sets of all input and output variables.

The composition of components with data comprises, in addition to a set of interactions $\gamma \subseteq 2^{P^e}$, the *data wire* relation $\delta \subseteq X^{in} \times X^{out}$, associating input and output variables in the system.

Definition 3.3.7. A system is *closed* if, for each interaction allowed by γ, all input variables have corresponding output variables. Denoting, for an interaction $a = \{p_i \,|\, i \in I\} \in \gamma$, $in(a) = \bigcup_{i\in I}\big(in(g_i) \cup in(f_i)\big)$ and $out(a) = \bigcup_{i \in I} d(p_i)$, this can be formally defined as follows: for any set of component transitions $\{q_i \xrightarrow{p_i,g_i,f_i} q_i' \,|\, i \in I\}$, the following property holds:

$$\forall x \in in(a), \big(\exists x' \in out(a) : (x, x') \in \delta\big).$$

A given interaction is possible in the composed system $\delta\gamma(B_1, \dots, B_n)$ if and only if the same interaction would be possible in the data-abstract system $\gamma(\widetilde{B}_1, \dots, \widetilde{B}_n)$ and, for each of the involved components, all the necessary inputs—i.e. the input data required by the guard and the update expression of the corresponding transition—are available and the guard is satisfied.

Definition 3.3.8. The composed component is defined as the tuple $\delta\gamma(B_1, \dots, B_n) = (Q, \gamma, P^s, X^{in}, X^{out}, d \rightarrow)$, where $Q = \prod_{i=1}^{n} Q_i$ is the Cartesian product of the sets of states of the individual components, the data access mapping is defined, for any interaction $a = \{p_i \,|\, i \in I\}$, by letting

$$d(a) = \bigcup_{i \in I} d_i(p_i),$$

and $\rightarrow$ is the set of transitions defined as follows:

- spontaneous port

$$\frac{p \in P^s \qquad q_i \xrightarrow{p,g,f} q_i'}{q_1 \dots q_i \dots q_n \xrightarrow{p,g,f} q_1 \dots q_i' \dots q_n}, \quad \text{for any } i \in [1, n]; \tag{3.4}$$

- internal transition

$$\frac{q_i \xrightarrow{\tau,g,f} q_i'}{q_1 \dots q_i \dots q_n \xrightarrow{\tau,g,f} q_1 \dots q_i' \dots q_n}, \quad \text{for any } i \in [1, n]; \tag{3.5}$$

- interaction

$$\frac{a \in \gamma \quad \left\{q_i \xrightarrow{a \cap P_i^e, g_i, f_i} q_i' \,\middle|\, i \in I\right\} \quad \left\{q_i = q_i' \,\middle|\, i \notin I\right\} \quad g = \bigwedge_{i \in I} \widetilde{g}_i \quad f = \big(\widetilde{f}_i\big)_{i \in I}}{q_1 \dots q_n \xrightarrow{a,g,f} q_1' \dots q_n'}, \tag{3.6}$$

where $I = \{i \in [1, n] \,|\, a \cap P_i^e \neq \emptyset\}$. $\widetilde{g}_i$ and $\widetilde{f}_i$ denote the expressions obtained, respectively, from g_i and f_i, by substituting each input variable x by the corresponding output variable $\delta'(x)$; for any mapping $\delta' : in(a) \to out(a)$, such that $(x, \delta'(x)) \in \delta$, for any $x \in in(a)$. $(\widetilde{f_i})_{i\in I}$ denotes the combined execution of the update expressions $\widetilde{f_i}$.

Notice that, in the definition of f, all the assignments are executed in parallel using the previous values provided as inputs to the component update expressions. Hence, the order of updates is irrelevant.

3.3.3 First-order interaction logic

We extend the propositional interaction logic presented in Section 2.3.3 with quantification over components to define interactions independently from the number of component instances. This extension is particularly useful because, in practice, systems are built from multiple component instances of the same component type. A first-order interaction logic was also presented in [25] for modelling dynamic architectures with additional history variables.

We make the following assumptions:

- A finite set of component types $\mathcal{T} = \{T_1, \dots, T_n\}$ is given. Instances of a component type have the same interface and behaviour. We write $B\!:\!T$ to denote that a component B is of type T.
- The interface of each component type has a distinct set of ports. We denote by $T.p$ the *port type* p, i.e. a port belonging to the interface of the type T. We write $B.p$, for a component $B\!:\!T$, to denote the *port instance* of the type $T.p$ and $B.P^e$ to denote the set of all enforceable ports of the component B.

Let ϕ denote any formula in propositional interaction logic.

Syntax. The first order interaction logic (FOIL) is defined by the grammar:

$$\Phi ::= true \mid \phi \mid \neg\,\Phi \mid \Phi \vee \Phi \mid \exists c : T(Pr(c)).\Phi\,, \tag{3.7}$$

where T is a component type, which represents a set of component instances with identical interfaces and behaviour. Variable c ranges over component instances and must occur in the scope of a quantifier. $Pr(c)$ is some set-theoretic predicate on c (omitted when $Pr = true$).

Additionally, we define the usual notation for universal quantifier:

$$\forall c\!:\!T(\mathrm{Pr}(c)).\Phi \overset{def}{=} \neg\,\exists c\!:\!T(\mathrm{Pr}(c)).\neg\,\Phi.$$

Semantics. The semantics is defined for closed formulas, where, for each variable in the formula, there is a quantifier over this variable in a higher nesting level. We assume that the finite set of component types $\mathcal{T} = \{T_1, \dots, T_n\}$ is given. Models are pairs $\langle \mathcal{B}, a \rangle$, where $\mathcal{B}$

is a set of component instances of types from $\mathcal{T}$ and a is an interaction on the set of ports P of these components. For quantifier-free formulas, the semantics is the same as for PIL formulas. For formulas with quantifiers, the satisfaction relation is defined as follows:

$$\langle \mathcal{B}, a\rangle \models_i \exists c : T(Pr(c)).\Phi\,, \qquad\qquad \text{iff } a \models_i \bigvee_{c':T\in B\wedge Pr(c')} \Phi[c'/c],$$

where $c' : T$ ranges over all component instances of type $T \in \mathcal{T}$ and $\Phi[c'/c]$ is obtained by replacing all occurrences of c in Φ by c'.

Example 3.3.9. Coordination schemes of Example 2.2.5 can be expressed in FOIL, as follows:

- *Rendezvous*: $\exists s{:}S.\ \forall r{:}R.\ (s.s\ r.r) \wedge\ \forall s'{:}S\ (s \neq s').\ (\overline{s.s\ s'.s}\,)$;
- *Broadcast*: $\exists s{:}S.\ (s.s) \wedge\ \forall s'{:}S\ (s \neq s').\ (\overline{s.s\ s'.s}\,)$;
- *Atomic broadcast*: $\exists s{:}S.\ \forall r{:}R.\ \big((s.s\ r.r) \vee (s.s\ r.\bar{r})\big) \wedge\ \forall s'{:}S\ (s \neq s').\ (\overline{s.s\ s'.s}\,)$.

Example 3.3.10. In a *Star architecture*, a single component of type S acts as the center, and all other components of type C communicate only with the center through binary rendezvous. The components communicate via rendezvous. This architecture is used in client-server applications. The Star architecture can be expressed in FOIL, as follows:

$$\exists s{:}S.\ \forall c{:}C.\ (s.p\ c.q) \wedge\ \forall c'{:}C\ (c \neq c').\ (\overline{c.q\ c'.q}\,) \wedge \forall s' : S(s = s').$$

3.3.4 Macro notation based on component types

JavaBIP relies on component types, rather than on component instances for the definition of architecture constraints. The same approach was also taken in [25]. We use two macros to define architecture constraints: 1) the **Require** macro and 2) the **Accept** macro.

The Require macro

Let us first illustrate the use of the Require macro without component types, i.e. for a system consisting of a given set of components. We want to define constraints of the form:

$$p \Rightarrow a_1 \vee \cdots \vee a_n\,, \tag{3.8}$$

with p being a port and each a_i being a conjunction of several ports. We call p the *effect* and $a_1 \dots a_n$ the *causes*. For p to participate in an interaction, all the ports belonging to at least one of $a_1 \dots a_n$ 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*.

To specify the constraint (3.8), we can use the macro:

$$p \ \textbf{Require} \ a_1; \dots ; a_n \, .$$

We now extend the Require macro notation for component types, such that all instances of a given component type are restricted with the same set of synchronisation constraints. Let $T^1, T^2 \in \mathcal{T}$ be component types and let $B^1_i : T^1$ and $B^2_j : T^2$ (with $i \in [1, n]$, $j \in [1, m]$) be the corresponding component instances. We define

$$T^1.p \ \textbf{Require} \ T^2.q \qquad \equiv \qquad \bigwedge_{i=1}^{n} \left(B^1_i.p \ \Rightarrow \ \bigvee_{j=1}^{m} \Big(B^2_j.q \wedge \bigwedge_{k \neq j} \overline{B^2_k.q} \Big) \right) ,$$

which means that, to participate in an interaction, each of the ports $B^1_i.p$ (with $i \in [1, n]$) requires the participation of *precisely one* of the ports $B^2_j.q$ (with $j \in [1, m]$). This is in contrast with the meaning used in [25], where a separate macro is used to enforce uniqueness of the cause port. Thus, we have opted for a macro notation where the cardinality of the causes is explicit: should two instances of the same port type be required, this is specified by explicitly putting the cause port type twice:

$$T^1.p \ \textbf{Require} \ T^2.q \ T^2.q \qquad \equiv \qquad \bigwedge_{i=1}^{n} \left(B^1_i.p \ \Rightarrow \ \bigvee_{j=1}^{m} \bigvee_{k \neq j} \Big(B^2_j.q \ B^2_k.q \wedge \bigwedge_{l \neq j,k} \overline{B^2_l.q} \Big) \right)$$

and so on for higher cardinalities. This choice of notation is motivated by our observation that cardinalities higher than one are very rare in practical examples.

The general form of the Require macro, for any number of component types and corresponding component instances, can be expressed in FOIL as follows:

$$\begin{aligned} T^1.p \ \textbf{Require} \ \Big(T^1.q_1\Big)^{m_1} \dots \Big(T^k.q_k\Big)^{m_k} \equiv & \\ \forall B^1 : T^1 . \ \exists B^1_1 : T^1, \dots, B^1_{m_1} : T^1, \dots, B^k_1 : T^k, \dots, B^k_{m_k} : T^k . & \\ \forall B^l : T^l \ (B^1 \neq B^1_1 \neq \dots \neq B^k_{m_k} \neq B^l) . & \\ \Big(B^1.p \Rightarrow B^1_1.q_1 \dots B^1_{m_1}.q_1 \dots B^k_1.q_k \dots B^k_{m_k}.q_k \overline{B^l.q_l} \Big) , & \end{aligned}$$

where $T^l \in \{T^1, \dots, T^k\}$. The cardinality of a cause containing component type T_i is denoted by m_i.

The Accept macro

Accept macros define optional ports for participation, i.e. they define the boundary of interactions. They are expressed by explicitly excluding from interactions all the ports that are not optional. At propositional level, they are of the form $p \rightarrow false$, where port p is excluded from the interaction. However, adding such conjuncts explicitly for all ports that do not participate in a connector is rather tedious. Furthermore, it is often convenient to define the constraints for a subsystem independently of the way it will be used. In such case, some ports of the system, which will not participate in the defined interactions, are not yet known. Hence, one needs a notation to specify that only a certain set of ports are *accepted* for interaction, others being implicitly excluded. This is achieved by the macro

$$p \;\; \textbf{Accept} \;\; a\,, \qquad \text{which formally means} \qquad p \Rightarrow \bigwedge_{\substack{q \in P^e \setminus a \\ q \neq p}} \overline{q}\,.$$

We now extend the Accept macro notation for component types. Let $T^1, T^2 \in \mathcal{T}$ be component types and let $B^1_i : T^1$ and $B^2_j : T^2$ (with $i \in [1, n]$, $j \in [1, m]$) be the corresponding component instances. We define:

$$\begin{aligned} T^1.p \;\; \textbf{Accept} \;\; T^2.q \quad &\equiv \quad \bigwedge_{i=1}^{n} \left(B^1_i.p \;\Rightarrow \bigwedge_{\substack{r \in P^e \setminus \{B^2_j.q \,|\, j \in [1,m]\} \\ r \neq B^1_i.p}} \overline{r} \right), \\ \text{where} \qquad P^e \quad &= \quad \bigcup_{T \in \mathcal{T}} \bigcup_{B:T} B.P^e\,. \end{aligned}$$

The general form of the Accept macro, for any number of component types and corresponding component instances, can be expressed in FOIL as follows:

$$\begin{aligned} &T^1.p \;\; \textbf{Accept} \;\; T^1.q_1 \dots T^k.q_k \equiv \\ &\qquad \forall B^1 : T^1. \Big(\bigwedge_{T^i \in \mathcal{T} \setminus \{T^1, \dots, T^k\}} \;\; \bigwedge_{r_i \in T^i.P^e} \forall B^i : T^i.\; (B^1.p \Rightarrow \overline{B^i.r_i}) \;\wedge \\ &\qquad\qquad \bigwedge_{T^j \in \{T^1, \dots, T^k\}} \;\; \bigwedge_{r_k \in T^j.P^e \setminus T^j.q_j} \forall B^j : T^j.\; (B^1.p \Rightarrow \overline{B^j.r_k}) \Big)\,. \end{aligned}$$

Example 3.3.11. To illustrate the use of the above macro notation, let us consider the Star architecture example 3.3.10. The architecture constraints are specified by the following combination of macros:

$$\begin{array}{lll} \mathtt{C.q} \;\; \textbf{Require} \;\; \mathtt{S.p} & \qquad \mathtt{C.q} \;\; \textbf{Accept} \;\; \mathtt{S.p} & \\ \mathtt{S.p} \;\; \textbf{Require} \;\; - & \qquad \mathtt{S.p} \;\; \textbf{Accept} \;\; \mathtt{C.q} & \qquad , \end{array}$$

where the dash '−' in the last line indicates that the port `S.p` does not require synchronisation with any other port.

Example 3.3.12. The architecture constraints of the Camel routes example (Section 3.2) are specified by the following combination of macros:

Monitor.add	**Require**	Route.on	Monitor.add	**Accept**	Route.on
Monitor.rm	**Require**	Route.finished	Monitor.rm	**Accept**	Route.finished
Route.on	**Require**	Monitor.add	Route.on	**Accept**	Monitor.add
Route.finished	**Require**	Monitor.rm	Route.finished	**Accept**	Monitor.rm
Route.off	**Require**	–	Route.off	**Accept**	–

,

where the dashes '–' in the last line indicate that the port `Route.off` neither requires, nor accepts synchronisation with any other port. Recall that the port `Route.end` is spontaneous. Hence, it does not have any associated architecture constraints. Finally, notice that this set of macros is independent of the number of Camel routes in the system.

Example 3.3.13. Now, let us consider an alteration of the previous example, where we require that the Monitor component removes two routes simultaneously at the execution of the port `rm`. Thus, in the require macro for the `rm` port type, the cardinality of the causes must be equal to two. This is specified by putting the cause port type `finished` twice as shown below. Notice that the accept macro for the `finished` port type has also changed, by *accepting* not only `rm` but also `finished`.

Monitor.add	**Require**	Route.on	Monitor.add	**Accept**	Route.on
Monitor.rm	**Require**	Route.finished Route.finished	Monitor.rm	**Accept**	Route.finished
Route.on	**Require**	Monitor.add	Route.on	**Accept**	Monitor.add
Route.finished	**Require**	Monitor.rm	Route.finished	**Accept**	Monitor.rm Route.finished
Route.off	**Require**	–	Route.off	**Accept**	–

Example 3.3.14. Let us now consider a more general example to illustrate the expressiveness of the JavaBIP glue. Assume that there are three component types `A, B, C` with port types `a, b, c`, respectively. Through the require macros, we enforce the following three constraints: 1) `A.a` requires synchronization with two instances of `B.b`; 2) `B.b` requires synchronization either with a) a single instance of `A.a` and a single instance of `C.c` or b) just two instances of `C.c`; 3) `C.c` does not require synchronizations with other ports, however it accepts synchronizations with any possible combination of ports `A.a, B.b, C.c`. Notice that by the combination of the first two require macros, a synchronization involving exactly an instance of `A.a` and two instances of `B.b` is not allowed, since `B.b` requires at least one instance of `C.c` to also participate in the synchronization.

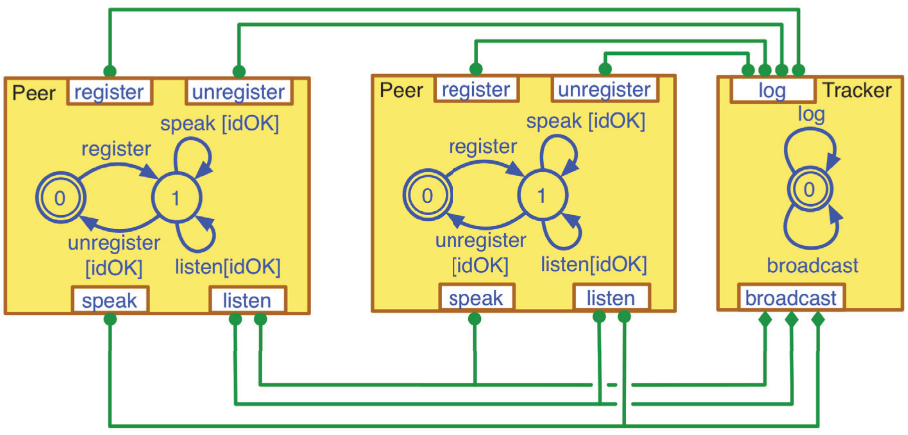


Figure 3.5 – JavaBIP models of one tracker and two peers

`A.a`	**Require**	`B.b B.b`	`A.a`	**Accept**	`A.a B.b C.c`
`B.b`	**Require**	`A.a C.c ; C.c C.c`	`B.b`	**Accept**	`A.a B.b C.c`
`C.c`	**Require**	–	`C.c`	**Accept**	`A.a B.b C.c`

Example 3.3.15. We show the Trackers and Peers example, which has a more complex architecture specification. This example, which was initially presented in [25], considers a simplified wireless audio protocol for reliable multicast communication. There are two component types: Tracker and Peer. The protocol allows an arbitrary number of peers to communicate along an arbitrary number of wireless communication channels. Each channel is managed by a unique tracker.

The model for two peers and one tracker is shown in Figure 3.5. 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` transition that is synchronized with the `log` transition of the tracker. During this synchronization, 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. To ensure atomicity of each communication, every tracker enforces that 1) at most one registered component is speaking and 2) all other registered components are listening.

In the connectors enforcing the above constraints, the `broadcast` port of a tracker is a trigger. This allows the `broadcast` transition to happen on its own, without requiring synchronization with transitions of other components. However, `broadcast` accepts synchronization with the `speak` and `listen` transitions. The `listen` and `speak` ports are *synchrons*, i.e. their corresponding transitions require synchronization with `broadcast`. Peers can register (resp.

unregister) through the `register-log` (resp. `unregister-log`) synchronization. The set of interactions allowed by these connectors can be specified as follows (notice the semicolon in the require constraint for `Tracker.log`):

`Peer.speak`	**Require**	`Tracker.broadcast`		
`Peer.speak`	**Accept**	`Tracker.broadcast`	`Peer.listen`	
`Peer.listen`	**Require**	`Tracker.broadcast`		
`Peer.listen`	**Accept**	`Tracker.broadcast`	`Peer.speak`	`Peer.listen`
`Peer.register`	**Require**	`Tracker.log`		
`Peer.register`	**Accept**	`Tracker.log`		
`Peer.unregister`	**Require**	`Tracker.log`		
`Peer.unregister`	**Accept**	`Tracker.log`		
`Tracker.broadcast`	**Require**	–		
`Tracker.broadcast`	**Accept**	`Peer.speak`	`Peer.listen`	
`Tracker.log`	**Require**	`Peer.register` ;	`Peer.unregister`	
`Tracker.log`	**Accept**	`Peer.register`	`Peer.unregister`	

Notice that the interaction structure of the example is exponentially complex. In particular, the number of all possible interactions is $(2^T - 1) \cdot (2^P + P \cdot 2^{P-1} + 2 \cdot P)$, where P is the total number of peers and T is the total number of trackers.

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 synchronize. Thus, all transitions of the system are enforceable and in all possible interactions (except when a tracker is broadcasting without any registered peers) data is exchanged between components.

3.4 System specification

The following subsections present the constructs used to provide behaviour, architecture and data-wire specifications (Figure 3.1 step ①). We refer to the Camel routes (Section 3.2) and Trackers and Peers (Example 3.3.15) examples to illustrate these constructs.

3.4.1 Behaviour specification

Developers must specify component behaviour through FSMs extended with ports and data. An FSM has states and guarded transitions between them. Each transition has a method and a port associated with it. Although, in general, one port can be associated with several transitions, such transitions must have different origin state. In other words, FSM are deterministic: there cannot be two transitions leaving the same state and labelled by the same port.

The Behaviour specification can be provided via annotations, associated with class, method and parameter declarations. To write a Behaviour specification, developers must use the following annotations:

- `@ComponentType`: Annotates a Java class. Declares a component type by specifying its name and the initial state of the underlying FSM (Figure 3.3: line 7).
- `@Port`: Annotates a Java class. Declares a port by specifying its name and type—"spontaneous" or "enforceable" (Figure 3.3: line 2).
- `@Ports`: Annotates a Java class. Groups all `@Port` annotations associated to a given component type (Figure 3.3: line 1).
- `@Guard`: Annotates a method returning a Boolean value. Declares that the method can be used as part of a transition guard, by specifying the guard name (Figure 3.3: line 38).
- `@Transition`: Annotates a method returning void. Declares an FSM transition, by specifying the name of the corresponding port, the source and the target states and the guard, which is a Boolean expression on the guard names declared with the `@Guard` annotation (Figure 3.3: line 29). Guard expressions can be defined using parenthesis and three logical operators: negation '!', conjunction '&' and disjunction '|'.

  The type of the transition is defined by the type of the port it is labelled with (Figure 3.3: lines 2–35). Internal transitions are specified by leaving the transition name empty (Figure 3.3: line 32).
- `@Data`: Annotates a non-void method or a method parameter. Defines the data required (input) or provided (output) by the component:
  - *input data*, when associated with a parameter of a guard or transition method (Figure 3.4: line 23);
  - *output data*, when associated with a method returning a value (Figure 3.3: line 44).

  `@Data` annotations always have the field `name`. Data names are used to establish connections (called *data-wires*—see Section 3.4.3) between inputs and outputs provided by the application components. When the `@Data` annotation is used to define an output data, it has two additional fields: `accessTypePort` (`allowed`, `disallowed` or `any`) and `ports`. The latter is a list of ports of the component in question, which is used to specify how this data can be accessed, based on the value of the `accessTypePort` field:
  - `allowed` means that the data can be accessed only through the ports in `ports`;

- `disallowed` means that the data can be accessed only through the ports *not* in `ports`;
- `any` means that the data can be accessed at any execution point—the list of ports must be empty.

Notice that the output values are provided by the `@Data`-annotated methods. The methods associated to transitions do not return any values—they must be declared as `public void`.

The usage of data as parameters in guards allows components to disable interactions based on the data values proposed by other components. For example, in Figure 3.4, transition `add` (lines 16–20) depends on the guard `hasCapacity`, which requires the data `memoryUsage` (lines 27–30). This data is received from another component potentially participating in an interaction. If the proposed data does not satisfy the guard, the interaction among these particular components is disabled.

3.4.2 Architecture specification

To define the interaction model, a developer must specify the architecture constraints of the system; i.e. which ports of different components must synchronize. Architecture constraints must be specified once for each *component type* of the system.

Architecture constraints are specified using the macro-notation presented in Section 3.3.4:

- *Causal constraints* (`Require`) specify ports of other components, necessary for any interaction involving the port with which the constraint is associated.
- *Acceptance constraints* (`Accept`) define optional ports of other components, accepted in the interactions involving the port with which the constraint is associated.

The Architecture specification must be provided in an XML file (Figure 3.6). Each constraint has two parts: `effect` and `causes`. The former defines the port to which the constraint is associated—intuitively, the effect is the firing of a transition labelled by this port. The latter lists the ports that are necessary to "cause" the "effect". For the `require` constraints, all causes must be present; for the `accept` constraints, any (possibly empty) combination of the causes is accepted.

The Architecture specification for the `broadcast` and `log` ports of the Tracker component type of Example 3.3.15 is shown in Figure 3.6. The causal constraint (Figure 3.6: lines 1–5) means that the `broadcast` ports of trackers do not require any synchronization with ports of other components. However, they accept synchronization with the `speak` and `listen` ports of the peers (Figure 3.6: lines 6–12). Notice that the require constraint for the port `log` has two `causes` sections, corresponding to the two independently sufficient causes (`Peer.register` and `Peer.unregister`).

```xml
<require>
  <effect id="broadcast" specType="Tracker"/>
   <causes>
   </causes>
</require>
<accept>
  <effect id="broadcast" specType="Tracker"/>
   <causes>
      <port id="speak" specType="Peer"/>
      <port id="listen" specType="Peer"/>
   </causes>
</accept>
<require>
  <effect id="log" specType="Tracker"/>
   <causes>
      <port id="register" specType="Peer"/>
   </causes>
   <causes>
      <port id="unregister" specType="Peer"/>
   </causes>
</require>
<accept>
  <effect id="log" specType="Tracker"/>
   <causes>
      <port id="register" specType="Peer"/>
      <port id="unregister" specType="Peer"/>
   </causes>
</accept>
```

Figure 3.6 – Architecture specification for the Trackers and Peers example (Example 3.3.15)

```xml
<wire>
  <from specType="Tracker" id="trackerId"/>
  <to specType="Peer" id="inputID"/>
</wire>
```

Figure 3.7 – Data-wire specification for the Trackers and Peers example (Example 3.3.15)

3.4.3 Data-wire specification

JavaBIP components exchange data and make decisions concerning the components they want to interact with based on the data they receive. Data wires specify data that can be exchanged between components, by connecting the input data with the output data provided by other components.

The data-wire specifications must be provided in an XML file, as shown in Figure 3.7. For Trackers and Peers (Example 3.3.15), once a peer registers to a tracker, it collects the tracker's ID. The data wire of Figure 3.7 connects the input data of the peer (`inputID`) with the output data of the Tracker (`trackerID`). The data transfer is finalized only if the associated transitions that require data can be executed, i.e. the corresponding guards evaluate to *true*.

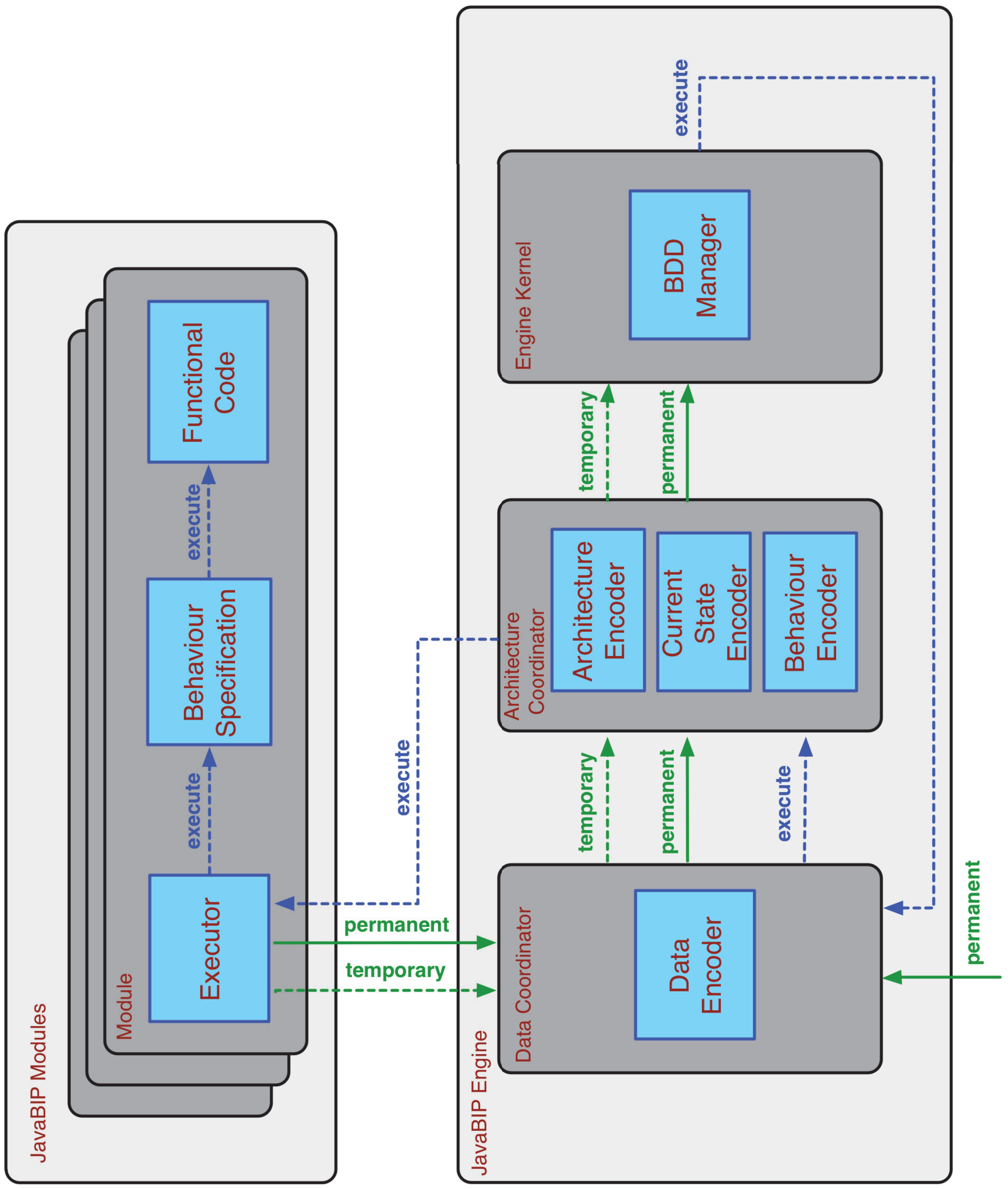


Figure 3.8 – JavaBIP software architecture

3.5 Implementation of the JavaBIP engine

The software architecture of the *JavaBIP runnable system* is shown in Figure 3.8. It consists of two main parts: the modules and the engine, as shown, respectively, in the top and bottom parts of the figure. The exchange of information between the engine and the modules is illustrated in Figure 3.8 with arrows. Information is either exchanged only once at initialization of the engine (illustrated in Figure 3.8 with continuous arrows) or at each execution cycle (illustrated in Figure 3.8 with dashed arrows).

A module comprises the functional code and the behaviour specification of the corresponding component (Section 3.4.1), as well as a dedicated executor. The behaviour specification contains the FSM with calls to the API methods provided by the component. It is used by the executor to drive the interaction with the engine and the environment. The JavaBIP engine creates a new executor instance upon registration of each module. Thus, executors are almost entirely transparent for developers. Indeed, the Executor class implements several interfaces, among which the only one that is visible to developers is the interface that provides only one method for sending spontaneous events to the executor.

The engine takes as input behaviour, architecture and data-wire specifications. The specifications are categorized either as permanent or temporary, illustrated in Figure 3.8 with arrows labelled permanent and temporary, respectively. Permanent specifications are sent only at initialization of the engine, while temporary specifications are sent at each execution cycle. 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 specifications acquired from the modules into permanent and temporary constraints that are sent to the kernel.

The kernel solves the combined constraints imposed by the behaviour, architecture and data-wire specifications and passes the solution back to the coordinators. 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 source code are made. How the solution is forwarded from the kernel to the functional code is illustrated in Figure 3.8 with dashed arrows labelled `execute`. If the kernel cannot find a solution because the combined constraints are contradictory, a deadlock occurs.

3.5.1 Engine kernel

The kernel combines and solves the various constraints of the system. Its implementation is based on Binary Decision Diagrams (BDDs)[1] [2], which are efficient data structures to store and manipulate Boolean formulas. The kernel applies the three-step protocol presented in Section 2.1 in a cyclic manner. In particular, it receives from the coordinators constraints in the form of Boolean formulas and assembles them by taking their conjunction to find a

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

solution. The solution is sent back to the coordinators which interpret it and notify the components accordingly. The imposed constraints can be of two types:

- *Permanent constraints* that are received only once at initialization. They include information about the Behaviour, Architecture and Data-wires of the components. In Figure 3.8, permanent constraints are shown with arrows labelled permanent.

- *Temporary constraints* that are received at each execution cycle. They include information about the enabled transitions of components. In Figure 3.8, temporary constraints are shown with arrows labelled temporary.

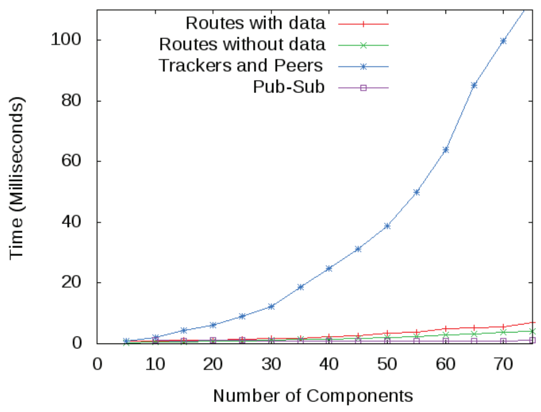
3.5.2 Coordinators

Coordinators manage the flow of information between components and the engine kernel. They receive different types of information and encode them as Boolean constraints using dedicated encoders. We have developed two coordinators that produce different types of constraints: the Architecture coordinator and the Data coordinator.

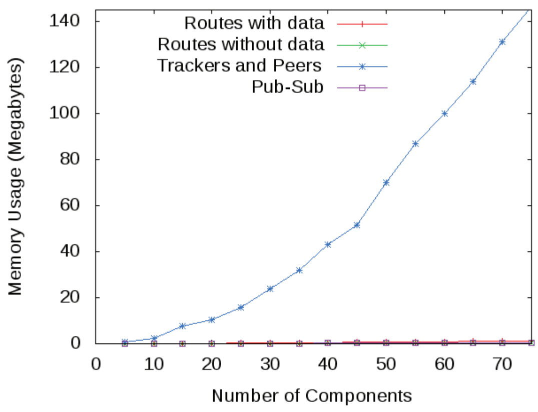
The *Architecture coordinator* manages the information about the behaviour, architecture and current state of the components. It encompasses three dedicated encoders (Figure 3.8): the Behaviour encoder, the Architecture encoder and the Current State encoder. The Boolean constraints encoding component behaviour and architecture are permanent, hence only computed—by the Behaviour and Architecture encoders, respectively—once at initialization. The Boolean constraints encoding the current states of components are temporary, hence recomputed at each execution cycle.

The *Data coordinator*, relying on one encoder (Data encoder), is used on top of the Architecture coordinator. It encodes as permanent constraints the information about data-wires, which connect input and output data provided by the components. At each execution cycle, the Data coordinator produces temporary constraints imposed on component interaction by the guards associated to component inputs. These temporary constraints disable the interactions involving data transfer, 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.

As shown in Figure 3.8, the coordinators form a chain. Depending on the needs of the application, different coordinators can be used. For instance, if there is data transfer, the Data coordinator must be used on top of the Architecture coordinator. Otherwise, the use of solely the Architecture coordinator is sufficient. Other coordinators can be easily added to manage other types of constraints—the implementation of the coordinator stack renders the architecture extensible.



(a) Average engine execution time.



(b) BDD Manager peak memory usage.

Figure 3.9 – Performance diagrams

3.5.3 Experimental evaluation

Experimental setup

We show experimental results for three case-studies: 1) two implementations of the Camel routes example (Section 3.2), i.e. one with and one without data transfer; 2) the Trackers-and-Peers example (Example 3.3.15) and 3) the Publish-Subscribe example that we have added in the appendix (Appendix A.1) of the thesis. The experiments were run on Intel Core i7-2640M CPU at 2.80GHz x 4 with 8GB RAM. The JavaBIP models of these systems are available at the JavaBIP website[2].

In the Camel routes examples (with and without data), we consider $C - 1$ routes and 1 monitor, where C is the total number of components. In the Trackers-and-Peers example, there are always four times more peers than trackers. In the Publish-Subscribe example there is always one buffer, one topic-manager, and varying numbers of TCPReaders, handlers and topics. The number of client-proxies is the same as the number of TCPReaders.

Figure 3.9a 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 3.9b shows the peak memory usage of the BDD Manager for each of the three examples. Table 3.1 summarizes all results shown in Figures 3.9a and 3.9b. The Camel routes implementations illustrate the impact of data transfer on the performance of the engine. The behavior and interaction models of the two Camel route implementations 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 3.9a.

[2]http://risd.epfl.ch/javabip

Table 3.1 – Engine times and BDD Manager peak memory usages. Time is in milliseconds and memory is in Megabytes.

Nb of	**Routes no data**		**Routes with data**		**Trackers & Peers**		**Publish-Subscribe**	
comps	Time	Memory	Time	Memory	Time	Memory	Time	Memory
5	< 1	0.010	< 1	0.015	< 1	0.640	< 1	0.026
10	< 1	0.029	< 1	0.075	2.103	2.278	< 1	0.048
15	< 1	0.047	1.147	0.099	4.264	7.584	< 1	0.084
20	< 1	0.079	1.254	0.180	6.002	10.338	< 1	0.108
25	< 1	0.099	1.585	0.220	8.980	15.932	< 1	0.130
30	1.254	0.120	1.614	0.324	12.329	23.670	< 1	0.161
35	1.328	0.169	1.895	0.456	18.643	31.896	< 1	0.184
40	1.459	0.200	2.393	0.560	24.727	43.045	< 1	0.233
45	1.874	0.238	2.731	0.700	31.187	51.598	< 1	0.251
50	2.167	0.280	3.568	0.780	38.943	69.984	< 1	0.295
55	2.346	0.340	3.796	0.840	49.766	87.097	< 1	0.315
60	2.786	0.387	5.093	0.920	63.766	99.983	< 1	0.338
65	3.286	0. 410	5.345	1.028	85.327	113.983	< 1	0.366
70	3.749	0.450	5.548	1.105	99.876	131.237	1.001	0.394
75	4.133	0.488	6.970	1.170	113.657	146.476	1.125	0.437

Table 3.2 – Number of Boolean variables used for States (S), Ports (P) and Data Variables (DV).

Nb of	**Routes no data**			**Routes with data**			**Trackers & Peers**			**Publish-Subscribe**		
comps	S	P	DV	S	P	DV	S	P	DV	S	P	DV
5	17	14	-	17	14	8	9	18	16	5	6	5
50	197	149	-	197	149	98	90	180	160	50	51	50
75	297	224	-	297	224	148	135	270	240	75	76	75

It should be noted that the complexity and the overhead induced by the JavaBIP engine depends on the size of the kernel BDD. This is characterised by the number of Boolean variables used in the encoding and their ordering in relation with the encoded constraints. Table 3.2 summarizes the number of Boolean variables used by the engine for each of the four case-studies for 5, 50 and 75 components. Notice that we take into account only enforceable ports, which correspond to transitions that are controlled by the JavaBIP engine. Table 3.3 presents the total number of variables (which is the sum of states, ports and data variables shown in Table 3.2) and the number of possible interactions computed by the engine for each of the four case-studies for 5, 50 and 75 components. For 75 components, the total number of variables is: 1) 521 for Camel Routes without data; 2) 669 for Camel Routes with data; 3) 645 for Trackers & Peers and 4) 226 for Publish-Subscribe.

Although the increase of the number of variables results in increasing the overhead of the engine, what really affects the performance of the engine is the complexity of the glue specification. We have used the Trackers & Peers case-study as a stress test for the JavaBIP engine. In particular, in the Trackers & Peers case-study, the number of possible interactions increases exponentially with the number of components, e.g., for 75 components, there exist $1.17 \cdot 10^{24}$ possible interactions (Table 3.3). Coordinating a system of such complexity with

Table 3.3 – Total number of Boolean Variables (V). Number of possible Interactions (I).

Nb of	Routes no data		Routes with data		Trackers & Peers		Publish-Subscribe	
comps	V	I	V	I	V	I	V	I
5	31	8	39	8	43	46	16	5
50	346	98	444	98	430	$2.36 \cdot 10^{16}$	151	50
75	521	148	669	148	645	$1.17 \cdot 10^{24}$	226	75

traditional techniques would be very difficult and error-prone. In JavaBIP, the full glue specification does not exceed 20 lines of code and is specified externally for the global system.

To compare the engine's execution time, we have used Camel Routes to transfer files of different size on an Intel Core i7-2640M CPU at 2.80GHz x 4 with 8GB RAM. We measured that a Camel Route needs 113ms to transfer a 3KB file, while for a 75MB file a Camel Route needs 890ms. Notice that this is "an ideal scenario" since only one Camel route is running at each time. In the case where 4 Camel Routes are running simultaneously, to transfer the 75MB file we need more than 900ms. We argue that the engine's overhead which is < 1ms for 4 Camel Routes and 1 Monitor is negligible when compared to 900ms or even to 113ms. Additionally, the memory usage of the BDD Manager remains very low, less than 2MB for 75 components, for the Camel Routes with data case-study.

3.6 Related work

Locking [103, 65] provides very efficient means for coordinating concurrent access to shared resources. However, it leads to code that is hard to understand, debug and modify. Several solutions have been proposed for dealing with these difficulties [1, 30]. However, they focus primarily on scalability at the expense of the expressiveness of coordination primitives. They do not impose the separation of computation from coordination, thus only partially alleviating the above difficulties and reducing modularity, since the component code substantially depends on its environment. Xcd [81] and SIP [52], impose the separation of concerns principles while dealing with coordination of concurrent software components. Xcd makes the choice of emphasizing specifically the realizability of distributed architectures, thus excluding up-front the possibility of centralized synchronization. SIP represents the functional logic of an application as a state machine and manages synchronization contracts that specify application needs in resources. Even though this work bears similarity to our approach, it cannot be used to enforce complex synchronization scenarios as in JavaBIP.

A number of other approaches further explore the issue of coordination in concurrent environments. An aspect-oriented approach is taken in [105], where the authors focus on separating the design and the choice of a specific synchronization mechanism and present a library allowing declarative specification of synchronization using tagging. In [38], discrete-event systems theory is used to generate concurrency control code. Using control-flow graphs of threads coupled with predefined relevant control events, the authors build simplified FSMs for each thread. Based on these FSMs, a general supervisor is constructed, from which the concurrent code is generated using semaphores. In both of these approaches, tags or events are embedded in the code, resulting in poor separation of concerns.

In [82], a component model with explicit symbolic protocols based on Symbolic Transition Systems is presented. The authors define a component language to describe the interfaces and the protocol of a component. Connection mechanisms are represented by channels. However, only one-to-one connections are allowed. Furthermore, even though a controller is implemented whose role bears similarity to the JavaBIP executor, controller classes have to be manually created for each component in the architecture.

Using FSMs to model program behaviour constitutes a common practice. In [102], the authors propose a methodology to automatically extract the finite-state models of object-oriented class interfaces. In the extracted FSMs, the states correspond to methods, while the transitions represent acceptable sequencing of method calls. The extracted models may serve as documentation, can be used to check whether the program exhibits expected behaviour or as complementary material in a test suite. In our approach, FSMs define a relevant abstract view of the behaviour of the software, which is more complex and complete than just method call sequencing and therefore can be effectively used for more complex system analysis. For instance, a state is the combination of the control location (between method calls) and data valuations, which allows, among others, to consider guards.

Similarly, in [15, 14, 13] behavioural types, i.e. FSMs, are used to describe the behaviour of component types and their interaction protocols. Behavioural types are used for compatibility checks between component types and to facilitate correctness through runtime monitors and discovery of components at runtime. They strictly describe method calls between components, while data transfer and memory usage are not included. Behavioural types are also used in the Ptolemy framework with a focus on real-time systems [66].

In [3], the authors argue for the extension of objects with state information through Typestate-Oriented Programming. A number of works [37, 57] support the importance of this approach, which allows for the specification of classes of temporal safety properties in the form of FSMs. This extension of types serves in checking whether an operation is allowed in a specific context. These works address the problems of retrieving or verifying method call sequences, whereas JavaBIP focuses on enforcing coordination.

The Behaviour Driven Development [93] software approach puts a strong focus on explicitly stating the behaviour of system components. In particular, Behaviour Driven Development focuses on understanding the behaviour of the components defined as simple *given-when-then* scenarios. Such a scenario can be simply explained as follows: *given* the state of the component, *when* an action is executed the component reaches the corresponding *then* state. The JBehave tool [77] that supports Behaviour Driven Development also requires a user to create an object model (an instance of a Java class) that directly maps scenarios to functions within this object model. Similarly to Behaviour Driven Development, our approach puts a strong focus on explicitly stating the behaviour of the system components. The JavaBIP mechanism for behaviour specification could be used to define given-when-then scenarios for all transitions within a Behaviour specification.

3.7 Summary

The main goal of our work was to integrate architectures into mainstream software development with Java, in order to raise the abstraction level and separate functionality from coordination. JavaBIP follows an exogenous approach to the coordination of software components relying, for the interaction with the controlled components, on existing APIs. To this end, JavaBIP specifications are defined as separate Java classes that, on one hand, allow the interaction with *other* components through the mediation of the JavaBIP engine and, on the other hand, interact with the *controlled* components through the provided APIs and notifications of spontaneous events. All JavaBIP specifications can be defined and modified without altering the functional code, which is impossible with traditional approaches. We have presented experimental results that validate the effectiveness of JavaBIP for the coordination of software components. Verification tools, e.g., DFinder and nuXmv, can be used to validate JavaBIP specifications, ensuring correctness of the designed systems. JavaBIP coordination has been tested and validated in Connectivity Factory^{TM} by Crossing-Tech S.A.

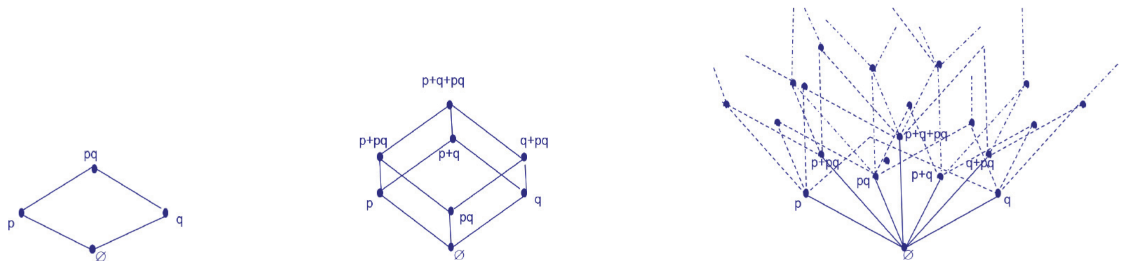
4 Configuration logics

Configuration logic is a powerset extension of interaction logic. Configuration logic formulas are generated from the formulas of propositional interaction logic by using the operators union, intersection and complementation, as well as a *coalescing operator* $+$. The meaning of a configuration logic formula is a configuration set. A configuration on a set of components represents a particular architecture. Thus, configuration logic formulas describe architecture sets. The definition of configuration logics requires considering the following three hierarchically structured semantic domains (Figure 4.1):

1. **The lattice of interactions.** An interaction a is a non-empty subset of P, the set of ports of the integrated components. Its execution implies the atomic synchronization of component actions (at most one action per component) associated with the ports of a.

2. **The lattice of configurations.** Configurations are non-empty sets of interactions characterizing architectures.

3. **The lattice of configuration sets.** Sets of configurations are properties described by the configuration logic.



(a) $I(P) = 2^P$ (b) $C(P) = 2^{I(P)\setminus\{\emptyset\}}$ (c) $CS(P) = 2^{C(P)\setminus\{\emptyset\}}$

Figure 4.1 – Lattices of interactions (a), configurations (b) and configuration sets (c) for $P = \{p, q\}$

We provide a full axiomatisation of the propositional configuration logic and a normal form similar to the disjunctive normal form in Boolean algebras. The existence of such a normal form implies the decidability of formula equality and satisfaction of a formula by an architecture model. To allow genericity of specifications, we study higher-order extensions of the propositional configuration logic.

4.1 Propositional configuration logic (PCL)

Syntax. The propositional configuration logic is an extension of propositional interaction logic (PIL) (Section 2.3.3) defined by the grammar:

$$f ::= \mathit{true} \mid \phi \mid \neg\, f \mid f + f \mid f \sqcup f\,, \tag{4.1}$$

where ϕ is a PIL formula; $\neg$, $+$ and $\sqcup$ are, respectively, the *complementation*, *coalescing* and *union* operators.

Additionally, we define the usual notation for intersection and implication:

$$\begin{aligned} f_1 \sqcap f_2 &\overset{def}{=} \neg\,(\neg\, f_1 \sqcup \neg\, f_2)\,, \\ f_1 \Rightarrow f_2 &\overset{def}{=} \neg\, f_1 \sqcup f_2\,. \end{aligned}$$

The language of PCL formulas is generated from PIL formulas by using union, coalescing and complementation operators. The binding strength of the operators is as follows (in decreasing order): PIL negation, complementation, PIL conjunction, PIL disjunction, coalescing, union.

Henceforth, to avoid confusion, we refer as *interaction formulas* to the subset of PCL formulas that syntactically are also PIL formulas. We will use Latin letters f, g, h for general PCL formulas and Greek letters ϕ, ψ, ξ for interaction formulas. Interaction formulas inherit all axioms of PIL.

Semantics. Let P be a set of ports. The semantic domain of PCL is the lattice of configuration sets $CS(P) = 2^{C(P)\setminus\{\emptyset\}}$ (Figure 4.1c). The meaning of a PCL formula f is defined by the following satisfaction relation. Let $\gamma \in C(P)$ be a non-empty configuration. We define:

$\gamma \models \mathit{true}\,,$	always,	(4.2)
$\gamma \models \phi\,,$	if $\forall a \in \gamma, a \models_i \phi$, where ϕ is an interaction formula and $\models_i$ is the satisfaction relation of PIL,	(4.3)
$\gamma \models f_1 + f_2\,,$	if there exist $\gamma_1, \gamma_2 \in C(P) \setminus \{\emptyset\}$, such that $\gamma = \gamma_1 \cup \gamma_2$, $\gamma_1 \models f_1$ and $\gamma_2 \models f_2$,	(4.4)
$\gamma \models f_1 \sqcup f_2\,,$	if $\gamma \models f_1$ or $\gamma \models f_2$,	(4.5)
$\gamma \models \neg\, f\,,$	if $\gamma \not\models f$ (i.e. $\gamma \models f$ does not hold).	(4.6)

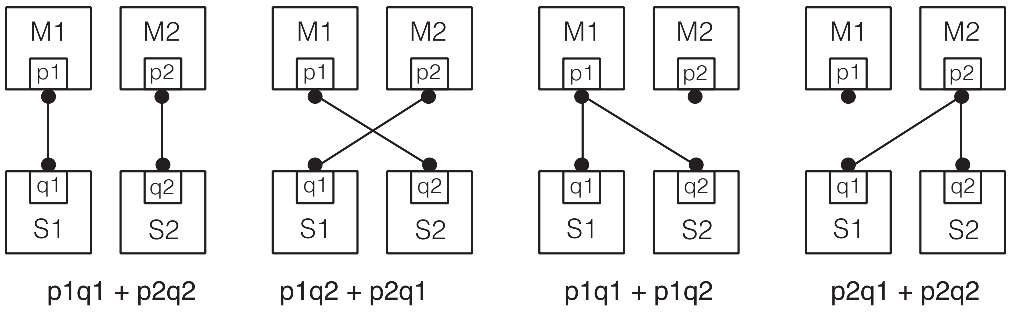


Figure 4.2 – Master/Slave architectures

In particular, the meaning of an interaction formula ϕ in PCL is the set $2^{I_a} \setminus \{\emptyset\}$, with $I_a = \{a \in I(P) \,|\, a \models_i \phi\}$, of all configurations involving any number of interactions satisfying ϕ in PIL.

The semantics of intersection and implication can also be stated directly as follows:

$$\gamma \models f_1 \sqcap f_2\,, \quad \text{if } \gamma \models f_1 \text{ and } \gamma \models f_2, \tag{4.7}$$

$$\gamma \models f_1 \Rightarrow f_2\,, \quad \text{if } \gamma \not\models f_1 \text{ or } \gamma \models f_2. \tag{4.8}$$

We say that two formulas are equivalent $f_1 \equiv f_2$ iff, for all $\gamma \in C(P)$ such that $\gamma \neq \emptyset$, $\gamma \models f_1 \Leftrightarrow \gamma \models f_2$.

We denote by $|f| \overset{def}{=} \{\gamma \in C(P) \setminus \{\emptyset\} \,|\, \gamma \models f\}$ the characteristic configuration set of the formula f. Clearly $f_1 \equiv f_2$ iff $|f_1| = |f_2|$.

Proposition 4.1.1. *Equivalence $\equiv$ is a congruence w.r.t. all PCL operations.*

Proof. In order to prove the proposition, it is sufficient to show that for each binary operator *op* from the PCL grammar (4.1), the characteristic configuration set of the formula $f_1\, op\, f_2$ can be expressed as a function of characteristic configuration sets of f_1 and f_2. In other words, we have to exhibit a binary operator op' on sets, such that $|f_1\, op\, f_2| = op'(|f_1|, |f_2|)$. Similarly, we have to exhibit a unary operator on sets, expressing the characteristic configuration set of the formula $\neg\, f$ in terms of the characteristic configuration set of f.

Clearly, the set operators corresponding to $\neg$ and $\sqcup$ are, respectively, complementation with respect to $C(P) \setminus \{\emptyset\}$ and set union. For the coalescing operator $+$, it is easy to see that, defining

$$op'_+(X, Y) \overset{def}{=} \{\gamma_1 \cup \gamma_2 \,|\, \gamma_1 \in X, \gamma_2 \in Y\}\,,$$

we have $|f_1 + f_2| = op'_+(|f_1|, |f_2|)$. □

Example 4.1.2. The Master/Slave architecture style for two masters M_1, M_2 and two slaves S_1, S_2 with ports $m_1,\ m_2,\ s_1,\ s_2$, respectively, characterizes the four configurations of Figure 4.2 as the union:

$$\bigsqcup_{i,j\in\{1,2\}} (\phi_{1,i} + \phi_{2,j}),$$

where $\phi_{i,j} = s_i \wedge m_j \wedge \overline{s_{i'}} \wedge \overline{m_{j'}}$ for $i \neq i', j \neq j'$ are monomials defining a binary interaction between ports s_i and m_j, respectively.

This formula can be alternatively written as a coalescing of interactions for each slave:

$$(\phi_{1,1} \sqcup \phi_{1,2}) + (\phi_{2,1} \sqcup \phi_{2,2}).$$

Any configuration satisfying this formula consists of two parts, which satisfy, respectively, the left and the right terms of the coalescing operator. The left term requires either an interaction $\{s_1, m_1\}$ or an interaction $\{s_1, m_2\}$. Similarly, the right term requires exactly one interaction among $\{s_2, m_1\}$ and $\{s_2, m_2\}$. Therefore, there are four possible pairs of interactions corresponding to the four configurations of Figure 4.2.

4.1.1 Properties of PCL

In this section, we show that PCL is a conservative extension of PIL. We also present the key properties of PCL operators, which allow us to define a normal form (Section 4.1.2), a sound and complete axiomatisation of PCL (Section 4.1.3) and decision procedures for the equality and satisfaction of PCL formulas (Subsection 4.1.4).

Conservative extension

Notice that from the PCL semantics of interaction formulas, it follows immediately that PCL is a conservative extension of PIL. Below we extend the PIL conjunction and disjunction operators to PCL.

PCL intersection is a conservative extension of PIL conjunction.

Proposition 4.1.3. *$\phi_1 \wedge \phi_2 \equiv \phi_1 \sqcap \phi_2$, for any interaction formulas ϕ_1, ϕ_2.*

Proof. For any two interaction formulas ϕ_1 and ϕ_2, $\phi_1 \wedge \phi_2$ is also an interaction formula. Hence, by (4.3), $\gamma \models \phi_1 \wedge \phi_2$ iff $\gamma \subseteq \{a \,|\, a \models_i \phi_1 \wedge \phi_2\} = \{a \,|\, a \models_i \phi_1 \wedge a \models_i \phi_2\}$. By (4.7), $\gamma \models \phi_1 \sqcap \phi_2$ iff $\gamma \models \phi_1$ and $\gamma \models \phi_2$, that is $\gamma \subseteq \{a \,|\, a \models_i \phi_1\} \cap \{a \,|\, a \models_i \phi_2\} = \{a \,|\, a \models_i \phi_1 \wedge a \models_i \phi_2\}$. Since characteristic configuration sets of formulas coincide, $\phi_1 \wedge \phi_2 \equiv \phi_1 \sqcap \phi_2$. $\square$

Thus, conjunction and intersection coincide on interaction formulas. In the rest of the thesis, we use the same symbol $\wedge$ to denote both operators.

Disjunction can be conservatively extended to PCL with the following semantics: for any PCL formulas f_1 and f_2,

$$\gamma \models f_1 \vee f_2\,, \qquad \text{if } \gamma \models f_1 \sqcup f_2 \sqcup f_1 + f_2. \tag{4.9}$$

Proposition 4.1.4. *For any interaction formulas ϕ_1 and ϕ_2 and any $\gamma \in C(P)$ such that $\gamma \neq \emptyset$, we have $\gamma \models \phi_1 \vee \phi_2$ iff $\forall a \in \gamma, a \models_i \phi_1 \vee \phi_2$.*

Proof. The PCL semantics defines $\gamma \models \phi_1 \vee \phi_2$ if $\gamma \models \phi_1$ or $\gamma \models \phi_2$ or there exist γ_1 and γ_2, such that $\gamma = \gamma_1 \cup \gamma_2$, $\gamma_1 \models \phi_1$ and $\gamma_2 \models \phi_2$, where $\gamma \models \phi$ if for all $a \in \gamma$, $a \models_i \phi$. Thus, in all three cases all interactions in γ either satisfy ϕ_1 or ϕ_2 and consequently, for all $a \in \gamma$, $a \models_i \phi_1 \vee \phi_2$. Conversely, if γ consists of interactions a, such that $a \models_i \phi_1 \vee \phi_2$, these interactions can be split into two possibly empty sets γ_1 and γ_2 such that for all $a \in \gamma_j$, where $j \in [1, 2]$, $a \models_i \phi_j$. If one of these groups is empty then the second one contains all interactions and $\gamma \models \phi_j$. Otherwise, $\gamma_1 \models \phi_1$ and $\gamma_2 \models \phi_2$, where $\gamma_1 \cup \gamma_2 = \gamma$. In all cases $\gamma \models \phi_1 \vee \phi_2$. $\square$

Union, complementation and conjunction have the standard set-theoretic meaning.

Proposition 4.1.5. *The operators $\sqcup$, $\neg$, $\wedge$ satisfy the usual axioms of propositional logic.*

Proof. The proof is immediate from the semantics (4.5), (4.6) and (4.7). $\square$

The coalescing operator

Notice that coalescing $+$ combines configurations, as opposed to union $\sqcup$, which combines configuration sets. Coalescing has the following properties:

Proposition 4.1.6. *$+$ is associative, commutative and has an absorbing element $false \overset{def}{=} \neg true$.*

Proof. The proof is immediate from the semantics (4.4). $\square$

Coalescing distributes over union, as shown in the following proposition:

Proposition 4.1.7. *For any formulas f, f_1, f_2, the following distributivity result holds:*

$$f + (f_1 \sqcup f_2) \equiv f + f_1 \sqcup f + f_2\,.$$

Proof. If $\gamma \models f + (f_1 \sqcup f_2)$ then there exist γ_1 and γ_2, such that $\gamma_1 \cup \gamma_2 = \gamma$, $\gamma_1 \models f$ and $\gamma_2 \models f_1 \sqcup f_2$. If $\gamma_2 \models f_1$ then $\gamma \models f + f_1$. Otherwise, $\gamma_2 \models f_2$ and $\gamma \models f + f_2$. Combining these two cases we obtain $\gamma \models f + f_1 \sqcup f + f_2$.

If $\gamma \models f + f_1 \sqcup f + f_2$ then either $\gamma \models f + f_1$ or $\gamma \models f + f_2$. In the first case there exist γ_1 and γ_2, such that $\gamma_1 \cup \gamma_2 = \gamma$, $\gamma_1 \models f$ and $\gamma_2 \models f_1$. Since $\gamma_2 \models f_1$ implies $\gamma_2 \models f_1 \sqcup f_2$, $\gamma \models f + (f_1 \sqcup f_2)$. The second case is similar. □

Associativity of coalescing and union, together with the distributivity of coalescing over union, immediately imply the following generalisation of the extended semantics of disjunction (4.9).

Corollary 4.1.8. *For any set of formulas* $\{f_i\}_{i \in I}$*, we have*

$$\bigvee_{i \in I} f_i \equiv \bigsqcup_{\emptyset \neq J \subseteq I} \sum_{j \in J} f_j \,,$$

where $\sum_{j \in J} f_j$ *denotes the coalescing of formulas* f_j*, for all* $j \in J$*.*

Example 4.1.9. A configuration γ satisfying the formula $f = f_1 \vee f_2 \vee f_3$ can be partitioned into $\gamma = \gamma_1 \cup \gamma_2 \cup \gamma_3$, such that $\gamma_i \models f_i$. However, by the semantics of disjunction, some γ_i can be empty. On the contrary, the semantics of coalescing requires all elements of such partition to be non-empty. Hence, in order to rewrite f without the disjunction operator, we take the union of all possible coalescings of f_1, f_2 and f_3. Thus, we have $f \equiv f_1 \sqcup f_2 \sqcup f_3 \sqcup (f_1 + f_2) \sqcup (f_1 + f_3) \sqcup (f_2 + f_3) \sqcup (f_1 + f_2 + f_3)$.

The following proposition shows distributivity results involving disjunction. In particular, it shows that disjunction distributes over union and coalescing distributes over disjunction.

Proposition 4.1.10. *For any formulas f, f_1, f_2, the following distributivity results hold:*

1. $f \vee (f_1 \sqcup f_2) \equiv (f \vee f_1) \sqcup (f \vee f_2)$,
2. $f + (f_1 \vee f_2) \equiv (f + f_1) \vee (f + f_2)$.

Proof. We have

$$\begin{aligned} f \vee (f_1 \sqcup f_2) &\equiv f \sqcup (f_1 \sqcup f_2) \sqcup f + (f_1 \sqcup f_2) \\ &\equiv f \sqcup f_1 \sqcup f + f_1 \sqcup f \sqcup f_2 \sqcup f + f_2 \equiv (f \vee f_1) \sqcup (f \vee f_2) \end{aligned}$$

and

$$\begin{aligned} f + (f_1 \vee f_2) &\equiv f + (f_1 \sqcup f_2 \sqcup f_1 + f_2) \\ &\equiv f + f_1 \sqcup f + f_2 \sqcup f + f_1 + f_2 \equiv (f + f_1) \vee (f + f_2) \,. \end{aligned}$$

□

The following example shows that coalescing does not distribute over conjunction.

Example 4.1.11. Let $P = \{p, q\}$ and consider $f = p \sqcup q$, $f_1 = p$ and $f_2 = q$. We then have $(f + f_1) \wedge (f + f_2) = ((p \sqcup q) + p) \wedge ((p \sqcup q) + q)$ and $f + (f_1 \wedge f_2) = (p \sqcup q) + (p \wedge q)$. The configuration $\{\{p\}, \{q\}\}$ satisfies the former, but not the latter.

Proposition 4.1.12. *For any formulas f, f_1, f_2, the following implication is true:*

$$f + (f_1 \wedge f_2) \Rightarrow (f + f_1) \wedge (f + f_2)\,.$$

Proof. If $\gamma \models f + (f_1 \wedge f_2)$ then there exist γ_1 and γ_2, such that $\gamma = \gamma_1 \cup \gamma_2$, $\gamma_1 \models f$, $\gamma_2 \models f_1$ and $\gamma_2 \models f_2$. Hence, we have both $\gamma \models f + f_1$ and $\gamma \models f + f_2$. $\square$

In general, neither conjunction distributes over coalescing nor coalescing over conjunction. To provide more distributivity results, we introduce the following classes of PCL formulas.

Definition 4.1.13.

- A formula f is *downward-closed* iff $\gamma \models f$ implies $\forall \gamma_1 \subseteq \gamma, \gamma_1 \models f$.
- A formula f is *upward-closed* iff $\gamma \models f$ implies $\forall \gamma_1 \supseteq \gamma, \gamma_1 \models f$.
- A formula f is $\cup$-*closed* iff $\gamma_1 \models f$ and $\gamma_2 \models f$ implies $\gamma_1 \cup \gamma_2 \models f$.

Example 4.1.14.

- $p \sqcup q$ is downward-closed,
- $\neg (p \sqcup q)$ is upward-closed,
- $p \vee q$ is $\cup$-closed.

The following propositions show properties of these classes and their relations.

Proposition 4.1.15. *If f and g are downward- (resp. upward-) closed then $f \sqcup g$ and $f \wedge g$ are also downward- (resp. upward-) closed.*

Proof. If $\gamma \models f \sqcup g$ then $\gamma \models f$ or $\gamma \models g$. If $\gamma \models f$ then $\forall \gamma_1 \subseteq \gamma\ \gamma_1 \models f$. Thus, $\gamma_1 \models f \sqcup g$. The case $\gamma \models g$ is similar.

If $\gamma \models f \wedge g$ then $\gamma \models f$ and $\gamma \models g$. If $\gamma \models f$ then $\forall \gamma_1 \subseteq \gamma\ \gamma_1 \models f$ and similarly for g. Thus, $\gamma_1 \models f \wedge g$.

If $\gamma \models f \sqcup g$ then $\gamma \models f$ or $\gamma \models g$. If $\gamma \models f$ then $\forall \gamma_1 \supseteq \gamma\ \gamma_1 \models f$. Thus, $\gamma_1 \models f \sqcup g$. The case $\gamma \models g$ is similar.

If $\gamma \models f \wedge g$ then $\gamma \models f$ and $\gamma \models g$. If $\gamma \models f$ then $\forall \gamma_1 \supseteq \gamma\ \gamma_1 \models f$ and similarly for g. Thus, $\gamma_1 \models f \wedge g$. $\square$

Proposition 4.1.16. *For any formula f, the formula $f + true$ is upward-closed.*

Proof. Let $\gamma \models f + true$. There exists $\gamma_1 \subseteq \gamma$ such that $\gamma_1 \models f$. For any $\gamma_2 \supseteq \gamma$ holds $\gamma_2 \supseteq \gamma_1$ and $\gamma_2 \models f + true$, since $true$ is satisfied by any configuration. □

Proposition 4.1.17. *If f is upward-closed then $f \equiv f + true$.*

Proof. If $\gamma \models f$ then $\gamma \cup \gamma = \gamma \models f + true$.
If $\gamma \models f + true$ then there exists $\gamma_1 \subseteq \gamma$ such that $\gamma_1 \models f$. Since f is upward-closed, for any $\gamma \supseteq \gamma_1$, holds $\gamma \models f$. □

Proposition 4.1.18. *If f and g are $\cup$-closed then $f + g$ is also $\cup$-closed.*

Proof. If $\gamma_1 \models f + g$ and $\gamma_2 \models f + g$ then there exist $\gamma_{1,1}$, $\gamma_{1,2}$, $\gamma_{2,1}$ and $\gamma_{2,2}$, such that $\gamma_i = \gamma_{i,1} \cup \gamma_{i,2}$, $\gamma_{i,1} \models f$ and $\gamma_{i,2} \models g$ for $i \in \{1, 2\}$. Since f and g are $\cup$-closed, $\gamma_{1,1} \cup \gamma_{2,1} \models f$ and $\gamma_{1,2} \cup \gamma_{2,2} \models g$ and consequently, $\gamma_1 \cup \gamma_2 \models f + g$. □

The following proposition shows that the complement of a downward-closed formula is an upward-closed formula.

Proposition 4.1.19. *A formula f is downward-closed iff the formula $\neg f$ is upward-closed.*

Proof. Assume that f is downward-closed and $\neg f$ is not upward-closed. The latter means that there exist γ_1 and $\gamma_2 \supseteq \gamma_1$ such that $\gamma_1 \models \neg f$ and $\gamma_2 \not\models \neg f$. This is equivalent to $\gamma_1 \not\models f$ and $\gamma_2 \models f$, which contradicts the fact that f is downward-closed.

Conversely, assume that $\neg f$ is upward-closed and f is not downward-closed. The latter means that there exist γ_1 and $\gamma_2 \subseteq \gamma_1$ such that $\gamma_1 \models f$ and $\gamma_2 \not\models f$. This is equivalent to $\gamma_1 \not\models \neg f$ and $\gamma_2 \models \neg f$, which contradicts the fact that $\neg f$ is upward-closed. □

Proposition 4.1.20. *A formula is $\cup$-closed and downward-closed iff it is an interaction formula.*

Proof. Let ϕ be an interaction formula. Consider two configurations $\gamma_1 \models \phi$ and $\gamma_2 \models \phi$. Any $\gamma' \subseteq \gamma_1$ contains only interactions from γ_1, thus, $\gamma' \models \phi$. For all $a \in \gamma_1 \cup \gamma_2$ holds $a \models_i \phi$, consequently $\gamma_1 \cup \gamma_2 \models \phi$. This shows that ϕ is $\cup$-closed and downward-closed.

Conversely, suppose that f is a $\cup$-closed and downward-closed formula and consider its characteristic configuration set $|f| = \{\gamma \in C(P) \setminus \{\emptyset\} \,|\, \gamma \models f\}$. Let $I = \bigcup_{\gamma \in |f|} \gamma$ be the set of all interactions belonging to configurations satisfying f. Since f is downward-closed, $\{a\} \models f$ for any $a \in I$. By the definition of $\cup$-closed formulas, the union of models is also a model. Thus, $\gamma \models f$, for any $\emptyset \neq \gamma \subseteq I$. Consequently, $|f| = \{\gamma \subseteq I \,|\, \gamma \neq \emptyset\}$ and $f = \bigvee_{a \in I} m_a$, where m_a denotes the characteristic monomial of the interaction a. □

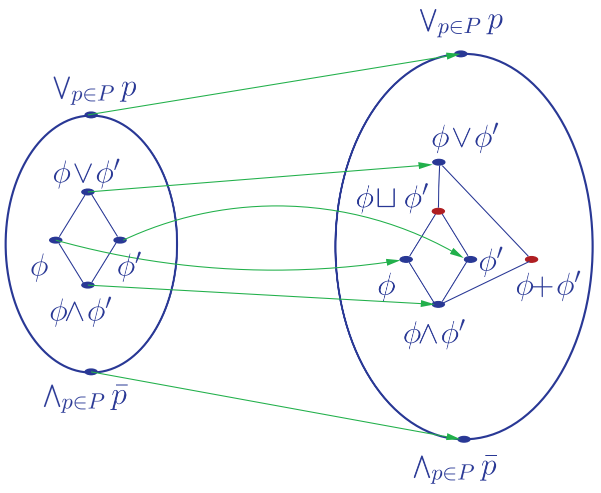


Figure 4.3 – Correspondence between the lattices of PIL and PCL

Thus, interaction formulas are represented by formulas that are both downward-closed and $\cup$-closed. Figure 4.3 shows the correspondence between the PIL lattice and the PCL lattice. Notice that, in general, $\phi \sqcup \phi'$ is not $\cup$-closed and $\phi + \phi'$ is not downward-closed. For example, for $P = \{p, q\}$, $f_1 = p\overline{q} \sqcup \overline{p}\, q$ is not $\cup$-closed, since $\{\{p\}\}$ and $\{\{q\}\}$ are models of f_1 but $\{\{p\}, \{q\}\}$ is not a model of f_1. Similarly, $f_2 = p\overline{q} + \overline{p}\, q$ is not downward-closed, since $\{\{p\}, \{q\}\}$ is a model of f_2 but neither $\{\{p\}\}$ nor $\{\{q\}\}$ is.

As shown before, conjunction does not distribute over coalescing. Nevertheless, it distributes for interaction formulas as shown in the following proposition.

Proposition 4.1.21. *For any formulas f_1, f_2 and interaction formula ϕ, we have:*

$$\phi \wedge (f_1 + f_2) \equiv (\phi \wedge f_1) + (\phi \wedge f_2)\,.$$

Proof. If γ is a configuration satisfying $\phi \wedge (f_1 + f_2)$ then $\gamma \models \phi$ and there exist γ_1, γ_2, such that $\gamma = \gamma_1 \cup \gamma_2$, $\gamma_1 \models f_1$ and $\gamma_2 \models f_2$. Since ϕ is an interaction formula, it is also downward-closed (Proposition 4.1.20). Thus, $\gamma \models \phi$ implies $\gamma_1 \models \phi$ and $\gamma_2 \models \phi$. Consequently, $\gamma_1 \models \phi \wedge f_1$ and $\gamma_2 \models \phi \wedge f_2$.

Conversely, if γ is a configuration satisfying $(\phi \wedge f_1) + (\phi \wedge f_2)$ then $\gamma = \gamma_1 \cup \gamma_2$ such that $\gamma_1 \models f_1, \gamma_1 \models \phi, \gamma_2 \models f_2$ and $\gamma_2 \models \phi$. Since ϕ is $\cup$-closed, $\gamma \models \phi$ and consequently, $\gamma \models \phi \wedge (f_1 + f_2)$. ☐

Notice that coalescing is not idempotent in general, as it is shown in the following example.

Example 4.1.22. $(p \sqcup q) + (p \sqcup q) \not\equiv p \sqcup q$. The configuration $\{\{p\}, \{q\}\}$ satisfies $(p \sqcup q) + (p \sqcup q)$, but it does not satisfy $p \sqcup q$.

Nevertheless, coalescing is idempotent on $\cup$-closed formulas.

Proposition 4.1.23. *$f + f \equiv f$ for any $\cup$-closed formula f.*

Proof. The implication $\gamma \models f \Rightarrow \gamma \models f + f$ for any γ is trivial.

Conversely, consider a configuration $\gamma \models f + f$. By the semantics of coalescing, there exist γ_1, γ_2, such that $\gamma = \gamma_1 \cup \gamma_2$, $\gamma_1 \models f$ and $\gamma_2 \models f$. Since f is $\cup$-closed, $\gamma_1 \cup \gamma_2 \models f$. Consequently, $\gamma \models f$. □

The closure operator

Coalescing with *true* presents a particular interest for writing specifications, since they allow adding any set of interactions to the configurations satisfying f. Notice that *true* is not a neutral element of coalescing: only the implication $f \Rightarrow f + \mathit{true}$ holds in general.

Definition 4.1.24. For any formula f, the *closure operator* $\sim$ is defined by putting $\sim f \overset{def}{=} f + \mathit{true}$. We give $\sim$ the same binding power as $\neg$.

Although closure is not a primitive operator of PCL, it is easy to see that the semantics of closure can be directly defined by putting $\gamma \models \sim f$ iff exists $\gamma_1 \subseteq \gamma$ such that $\gamma_1 \models f$.

Example 4.1.25. For $P = \{p, q, r\}$ the formula f characterizing all the configurations such that p must interact with both q and r, is $f = pq + pr + \mathit{true} = \sim(pq + qr)$. Notice that the only constraint imposed by the formula f is that configurations that satisfy it must contain an interaction pqr or both interactions pq and qr. Configurations satisfying f can contain any additional interactions.

Proposition 4.1.26. *$\sim\sim f \equiv \sim f$ for any formula f.*

Proof. $\sim\sim f \equiv \sim f + \mathit{true} \equiv f + \mathit{true} + \mathit{true} \equiv f + \mathit{true} \equiv \sim f$. □

Notice that, as an immediate corollary of Proposition 4.1.17, the closure of any formula is upward-closed. The following proposition shows that $\sim f$ is the smallest upward-closed formula greater than f in the lattice of PCL formulas ordered by implication.

Proposition 4.1.27. *For any formula f, holds $f \Rightarrow \sim f$. Furthermore, for any upward-closed formula f', such that $f \Rightarrow f'$, holds $\sim f \Rightarrow f'$.*

Proof. $f \Rightarrow \sim f$ follows directly from the semantics of the $\sim$ operator. Assume that there exists an upward-closed f', such that $f \Rightarrow f'$, and a configuration γ, such that $\gamma \models \sim f$ and $\gamma \not\models f'$. Since $\gamma \models \sim f$ there exists $\gamma_1 \subseteq \gamma$ such that $\gamma_1 \models f$. Since $f \Rightarrow f'$, we have $\gamma_1 \models f'$. The formula f' is upward-closed, therefore $\gamma_1 \models f'$ implies $\gamma \models f'$, which contradicts our assumption. □

The closure operator can be interpreted as a modal operator with existential quantification. The formula $\sim f$ characterizes configurations γ, such that there *exists* a sub-configuration of γ satisfying f. Thus, $\sim f$ means "possible f". Dually $\neg \sim \neg f$ means "always f" in the following sense: if a configuration γ satisfies $\neg \sim \neg f$, *all* sub-configurations of γ satisfy f.

Corollary 4.1.28. *For any formula f, holds $\neg \sim \neg f \Rightarrow f$. Furthermore, for any downward-closed formula f', such that $f' \Rightarrow f$, holds $f' \Rightarrow \neg \sim \neg f$.*

Proof. By 4.1.27, for any formula f, we have $\neg f \Rightarrow \sim \neg f$, which immediately implies $\neg \sim \neg f \Rightarrow f$. For any downward-closed f', such that $f' \Rightarrow f$, we observe that, by Proposition 4.1.19, $\neg f'$ is upward-closed. Hence, by Proposition 4.1.27, $\sim \neg f \Rightarrow \neg f'$ and, consequently, $f' \Rightarrow \neg \sim \neg f$. □

Clearly, if f is downward-closed then $\neg \sim \neg f \equiv f$. However, this is not true in general. Consider $f = m_a + m_b$, where m_a and m_b are characteristic monomials of interactions a and b, respectively. The only configuration satisfying f is $\gamma = \{a, b\}$. In particular, none of the sub-configurations $\{a\}, \{b\} \subset \gamma$ satisfies f. Thus, $\neg \sim \neg (m_a + m_b) \equiv \mathit{false}$.

Proposition 4.1.29. *For any formulas f_1, f_2, the following distributivity results hold:*

1. $\sim(f_1 \sqcup f_2) \equiv \sim f_1 \sqcup \sim f_2 \equiv \sim(f_1 \vee f_2)$,
2. $\sim f_1 + \sim f_2 \equiv \sim(f_1 + f_2) \equiv \sim f_1 \wedge \sim f_2$.

Proof. We have the following equalities:

$$\begin{aligned}
\sim(f_1 \sqcup f_2) &\equiv (f_1 \sqcup f_2) + \mathit{true} \equiv f_1 + \mathit{true} \sqcup f_2 + \mathit{true} \equiv \sim f_1 \sqcup \sim f_2\,, \\
\sim(f_1 \vee f_2) &\equiv f_1 + \mathit{true} \sqcup f_2 + \mathit{true} \sqcup f_1 + f_2 + \mathit{true} \\
&\equiv f_1 + \mathit{true} \sqcup f_2 + \mathit{true} \equiv \sim f_1 \sqcup \sim f_2\,, \\
\sim f_1 + \sim f_2 &\equiv f_1 + \mathit{true} + f_2 + \mathit{true} \equiv f_1 + f_2 + \mathit{true} \equiv \sim(f_1 + f_2)\,, \\
\sim f_1 \wedge \sim f_2 &\equiv (f_1 + \mathit{true}) \wedge (f_2 + \mathit{true}) \equiv f_1 + f_2 + \mathit{true} \equiv \sim(f_1 + f_2)\,.
\end{aligned}$$

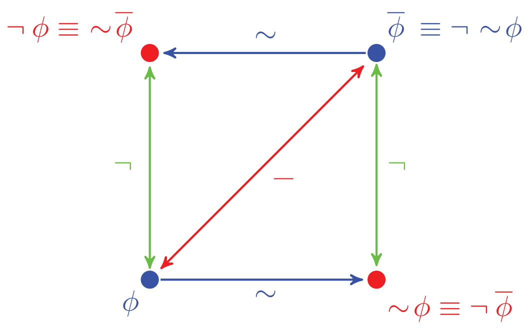
□

The complementation operator

The following results allow us to address the relation between complementation and negation.

Lemma 4.1.30. *For any interaction formula ϕ, the following two formulas are equivalent:*

$$\phi \sqcup \overline{\phi} \sqcup (\overline{\phi} + \phi) \equiv \mathit{true}\,. \tag{4.10}$$



$\neg\,\phi \equiv \sim\overline{\phi}$
$\sim$
$\overline{\phi} \equiv \neg \sim\phi$
$\neg$
$-$
$\neg$
ϕ
$\sim$
$\sim\phi \equiv \neg\,\overline{\phi}$

Figure 4.4 – Correspondence between negation and complementation of interaction formulas

Proof. The proof is immediate from Corollary 4.1.8 and the fact that $\phi \vee \overline{\phi} \equiv \mathit{true}$, for any interaction formula ϕ. □

Notice that the three terms in the left-hand side of (4.10) are mutually disjoint.

Proposition 4.1.31. *For any interaction formula ϕ, holds $\neg\,\phi \equiv \sim\overline{\phi}$.*

Proof. By Lemma 4.1.30, we have $\neg\,\phi \equiv \overline{\phi} \sqcup (\phi + \overline{\phi}) \equiv \overline{\phi} + \mathit{true} \equiv \sim\overline{\phi}$. □

In particular, this means that complementation can also be interpreted as a modality. Proposition 4.1.31 shows that the complementation of f represents all configurations that contain $\overline{\phi}$. Equivalences $\neg\,\overline{\phi} \equiv \sim\phi$, $\neg \sim\phi \equiv \overline{\phi}$, $\neg \sim\overline{\phi} \equiv \phi$ and $\sim\neg\,\phi \equiv \neg\,\phi$, for interaction formulas ϕ, are direct corollaries of Proposition 4.1.31 and, for the latter, Proposition 4.1.26. Figure 4.4 depicts the relations between complementation and negation of the interaction formulas.

Complementation of coalescings of interaction formulas

The following lemma expresses coalescing through extended disjunction. Coalescing is more restrictive than extended disjunction requiring the existence of sub-configurations that satisfy all operands.

Proposition 4.1.32. *For any formulas f, g, we have:*

$$f + g \equiv \sim f \wedge \sim g \wedge (f \vee g) .$$

Proof. By (4.9) and Proposition 4.1.29, we have

$$\sim f \wedge \sim g \wedge (f \vee g) \equiv \sim(f + g) \wedge (f \sqcup g \sqcup f + g) .$$

Notice that $\gamma \models \sim(f+g) \wedge f$ iff $\gamma \models f$ and there exists $\gamma_1 \subseteq \gamma$ such that $\gamma_1 \models g$. Thus, $\sim(f+g) \wedge f \equiv f + (f \wedge g)$. By applying a similar transformation to g, we obtain

$$\sim(f+g) \wedge (f \sqcup g \sqcup f+g) \equiv \big(f+(f \wedge g)\big) \sqcup \big(g+(f \wedge g)\big) \sqcup (f+g) \equiv f+g\,,$$

where the last equality is an immediate consequence of the fact that $f \wedge g \Rightarrow f$ and $f \wedge g \Rightarrow g$. ☐

Proposition 4.1.33. *For any interaction formulas ϕ, ψ, the following two formulas are equivalent:*

$$\neg\,(\phi+\psi) \equiv \overline{\phi} \sqcup \overline{\psi} \sqcup \sim(\overline{\phi} \wedge \overline{\psi})\,.$$

Proof. By Proposition 4.1.32 $\phi+\psi \equiv \sim\phi \wedge \sim\psi \wedge (\phi \vee \psi)$. Thus, $\neg\,(\phi+\psi) \equiv \neg\,(\sim\phi \wedge \sim\psi \wedge (\phi \vee \psi)) \equiv \neg\sim\phi \sqcup \neg\sim\psi \sqcup \neg\,(\phi \vee \psi)$. Since ϕ, ψ and $\phi \vee \psi$ are interaction formulas, the application of Proposition 4.1.31 gives $\neg\,(\phi+\psi) \equiv \overline{\phi} \sqcup \overline{\psi} \sqcup \sim(\overline{\phi} \wedge \overline{\psi})$ ☐

Proposition 4.1.33 allows the elimination of complementation as shown in the following example.

Example 4.1.34. Consider a formula $f = \neg\,(pq+pr)$ and a configuration $\gamma \models f$. The PCL semantics requires that γ cannot be split into two non-empty parts $\gamma_1 \models pq$ and $\gamma_2 \models pr$. This can happen in two cases: 1) there exists $a \in \gamma$ such that a does not satisfy neither pq nor pr; 2) one of the monomials is not satisfied by any interaction in γ. The former case can be expressed as $\sim(\overline{pq}\ \overline{pr})$ and the latter as $\overline{pq} \sqcup \overline{pr}$. The union of these formulas gives the equivalence $\neg\,(pq+pr) \equiv \overline{pq} \sqcup \overline{pr} \sqcup \sim(\overline{pq}\ \overline{pr})$.

Lemma 4.1.32 and Proposition 4.1.33 can be generalized as follows:

Proposition 4.1.35. *For any set of formulas F, we have:*

$$\sum_{f\in F} f \equiv \bigwedge_{f\in F} \sim f \wedge \bigvee_{f\in F} f\,.$$

Proposition 4.1.36. *For any set of interaction formulas Φ, the following two formulas are equivalent:*

$$\neg \sum_{\phi\in\Phi} \phi \equiv \bigsqcup_{\phi\in\Phi} \overline{\phi} \sqcup \sim \bigwedge_{\phi\in\Phi} \overline{\phi}\,.$$

Proofs of Propositions 4.1.35 and 4.1.36 are similar to the proofs of Propositions 4.1.32 and 4.1.33, respectively.

Conjunction of coalescings of interaction formulas

Conjunction of coalescings of interaction formulas can be eliminated by using the following distributivity result to push it down within the formula tree.

Proposition 4.1.37. *If Φ and Ψ are sets of interaction formulas, then*

$$\sum_{\phi\in\Phi}\phi \wedge \sum_{\psi\in\Psi}\psi \;\equiv\; \sum_{\xi\in\Phi\cup\Psi}\Big(\xi \wedge \bigvee_{(\phi,\psi)\in\Phi\times\Psi}(\phi\wedge\psi)\Big).$$

Proof. Notice that

$$\sum_{\phi\in\Phi}\phi \wedge \sum_{\psi\in\Psi}\psi \;\equiv\; \neg\neg\Big(\sum_{\phi\in\Phi}\phi \wedge \sum_{\psi\in\Psi}\psi\Big) \equiv \neg\Big(\neg\sum_{\phi\in\Phi}\phi \sqcup \neg\sum_{\psi\in\Psi}\psi\Big).$$

By Proposition 4.1.36, this can be further transformed into

$$\neg\left(\bigsqcup_{\phi\in\Phi}\overline{\phi} \sqcup \sim\bigwedge_{\phi\in\Phi}\overline{\phi} \sqcup \bigsqcup_{\psi\in\Psi}\overline{\psi} \sqcup \sim\bigwedge_{\psi\in\Psi}\overline{\psi}\right)$$
$$\equiv \neg\left(\bigsqcup_{\xi\in\Phi\cup\Psi}\overline{\xi} \sqcup \sim\bigwedge_{\phi\in\Phi}\overline{\phi} \sqcup \sim\bigwedge_{\psi\in\Psi}\overline{\psi}\right),$$

which we further transform by applying twice the De Morgan's law (once for complementation and union and once for negation and disjunction) and Proposition 4.1.31:

$$\bigwedge_{\xi\in\Phi\cup\Psi}\neg\overline{\xi} \wedge \neg\Big(\sim\bigwedge_{\phi\in\Phi}\overline{\phi}\Big) \wedge \neg\Big(\sim\bigwedge_{\psi\in\Psi}\overline{\psi}\Big) \;\equiv\; \bigwedge_{\xi\in\Phi\cup\Psi}\sim\xi \wedge \overline{\bigwedge_{\phi\in\Phi}\overline{\phi}} \wedge \overline{\bigwedge_{\psi\in\Psi}\overline{\psi}}\;.$$

By Proposition 4.1.29 and another application of De Morgan's law, we obtain

$$\sim\sum_{\xi\in\Phi\cup\Psi}\xi \wedge \bigvee_{\phi\in\Phi}\phi \wedge \bigvee_{\psi\in\Psi}\psi \equiv \sim\sum_{\xi\in\Phi\cup\Psi}\xi \wedge \bigvee_{(\phi,\psi)\in\Phi\times\Psi}(\phi\wedge\psi)\,.$$

Let γ be a configuration satisfying the formula in the right-hand side of this equation. By (4.7), any interaction $a\in\gamma$ satisfies the second conjunct in this formula. Hence, there exists a pair $(\phi,\psi)\in\Phi\times\Psi$, such that $a\models_i \phi\wedge\psi$ and, a fortiori, there exists $\xi\in\Phi\cup\Psi$, such that $a\models_i\xi$. Thus, the closure operator in the first conjunct of this formula can be discarded. Finally, by Proposition 4.1.21, we have

$$\Big(\sum_{\xi\in\Phi\cup\Psi}\xi\Big) \wedge \bigvee_{(\phi,\psi)\in\Phi\times\Psi}(\phi\wedge\psi) \;\equiv\; \sum_{\xi\in\Phi\cup\Psi}\Big(\xi \wedge \bigvee_{(\phi,\psi)\in\Phi\times\Psi}(\phi\wedge\psi)\Big).$$

□

Example 4.1.38. Consider a formula $f = (\phi_1+\phi_2)\wedge(\phi_3+\phi_4)$, where ϕ_1, ϕ_2, ϕ_3 and ϕ_4 are interaction formulas, and a configuration $\gamma\models f$. The semantics requires that there exists two partitions of γ: $\gamma=\gamma_1\cup\gamma_2$ and $\gamma=\gamma_3\cup\gamma_4$, such that $\gamma_i\models\phi_i$ for $i\in[1,4]$. Considering an intersection $\gamma_{i,j}=\gamma_i\cap\gamma_j$ we have $\gamma_{i,j}\models\phi_i\wedge\phi_j$. Thus, $\gamma=\bigcup\gamma_{i,j}$ satisfies $\phi_1\phi_3\vee\phi_1\phi_4\vee\phi_2\phi_3\vee\phi_2\phi_4$ even if some $\gamma_{i,j}$ are empty. Nevertheless, disjunction allows configurations such that no interaction satisfy one of the disjunction terms and consequently some ϕ_i. A coalescing of ϕ_i allows only configurations such that each ϕ_i is satisfied by at least

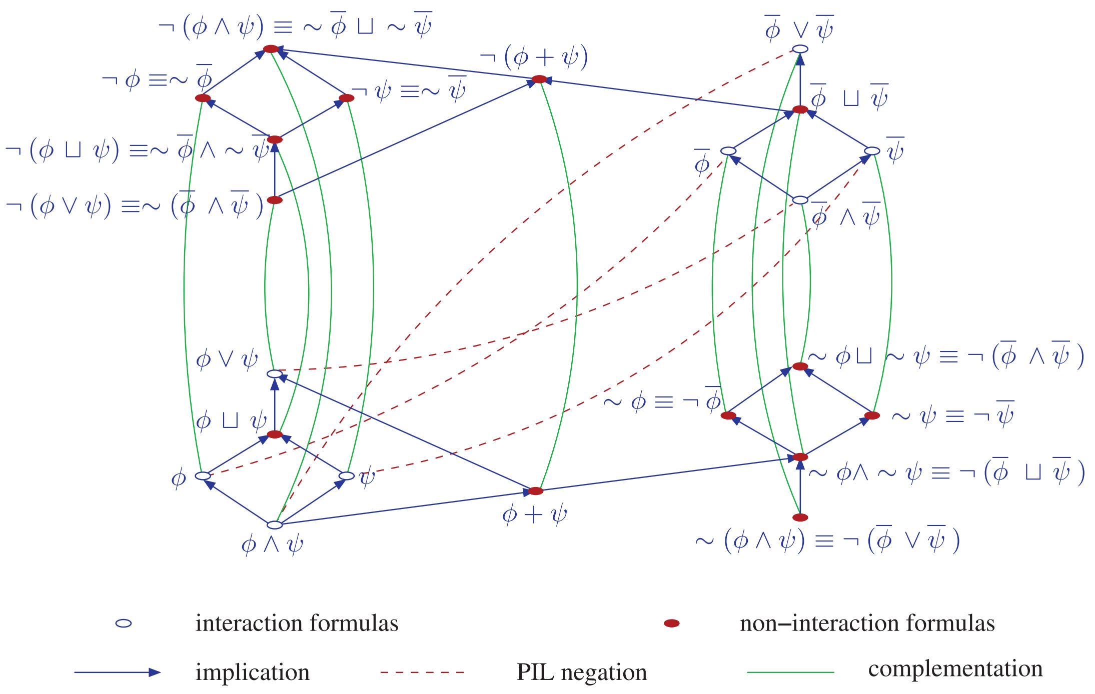

$\neg(\phi\wedge\psi)\equiv\sim\overline{\phi}\sqcup\sim\overline{\psi}$
$\neg\phi\equiv\sim\overline{\phi}$
$\neg\psi\equiv\sim\overline{\psi}$
$\neg(\phi\sqcup\psi)\equiv\sim\overline{\phi}\wedge\sim\overline{\psi}$
$\neg(\phi\vee\psi)\equiv\sim(\overline{\phi}\wedge\overline{\psi})$
$\neg(\phi+\psi)$
$\overline{\phi}\vee\overline{\psi}$
$\overline{\phi}\sqcup\overline{\psi}$
$\overline{\phi}$
$\overline{\psi}$
$\overline{\phi}\wedge\overline{\psi}$
$\phi\vee\psi$
$\phi\sqcup\psi$
ϕ
ψ
$\phi\wedge\psi$
$\phi+\psi$
$\sim\phi\sqcup\sim\psi\equiv\neg(\overline{\phi}\wedge\overline{\psi})$
$\sim\phi\equiv\neg\overline{\phi}$
$\sim\psi\equiv\neg\overline{\psi}$
$\sim\phi\wedge\sim\psi\equiv\neg(\overline{\phi}\sqcup\overline{\psi})$
$\sim(\phi\wedge\psi)\equiv\neg(\overline{\phi}\vee\overline{\psi})$
interaction formulas
non–interaction formulas
implication
PIL negation
complementation

Figure 4.5 – PCL lattice.

one interaction. Thus, the conjunction of these formulas gives the equivalent representation:

$$
\begin{aligned}
f &\equiv (\phi_1\phi_3 \vee \phi_1\phi_4 \vee \phi_2\phi_3 \vee \phi_2\phi_4) \wedge (\phi_1 + \phi_2 + \phi_3 + \phi_4)\\
&= \phi_1 \wedge (\phi_1\phi_3 \vee \phi_1\phi_4 \vee \phi_2\phi_3 \vee \phi_2\phi_4) + \phi_2 \wedge (\phi_1\phi_3 \vee \phi_1\phi_4 \vee \phi_2\phi_3 \vee \phi_2\phi_4)\\
&+ \phi_3 \wedge (\phi_1\phi_3 \vee \phi_1\phi_4 \vee \phi_2\phi_3 \vee \phi_2\phi_4) + \phi_4 \wedge (\phi_1\phi_3 \vee \phi_1\phi_4 \vee \phi_2\phi_3 \vee \phi_2\phi_4)\,.
\end{aligned}
$$

The PCL lattice

The PCL lattice is illustrated in Figure 4.5. The circle nodes represent interaction formulas, whereas the red dot nodes represent all other formulas. Notice that the PCL lattice has two sub-lattices generated by monomials:

- through disjunction and negation (isomorphic to the PIL lattice);
- through union and complementation (disjunction is not expressible).

Notice that coalescing cannot be expressed in any of these two sub-lattices. Although some formulas involving the closure operator can be expressed in the second sub-lattice, e.g. $\sim\overline{\phi}\ \equiv \neg\,\phi$, in general this is not the case, e.g. the formulas $\sim(\overline{\phi}\ \wedge\ \overline{\psi}\,)$ and $\sim\phi\sqcup\sim\psi$ are not part of either sub-lattice. However, the closure operator is expressible by taking as generators the interaction formulas:

Proposition 4.1.39. *The lattice generated by interaction formulas through union and complementation is closed under the closure operator $\sim$.*

Proof. We must prove that, for any formula f in this lattice, the formula $\sim f$ is also in the lattice.

Since union and complementation satisfy the usual axioms of propositional logic, f can be represented in the equivalent of the disjunction normal form:

$$f \equiv \bigsqcup_{i\in I}\Big(\bigwedge_{k\in K_i}\phi_k \wedge \bigwedge_{j\in J_i}\neg\,\phi_j\Big)\,,$$

where all ϕ_j and ϕ_k are interaction formulas. Furthermore, since the conjunction of interaction formulas $\bigwedge_{k\in K_i}\phi_k$ is also an interaction formula, we can assume, without loss of generality, that all K_i are singleton sets and

$$f \equiv \bigsqcup_{i\in I}\Big(\phi_i \wedge \bigwedge_{j\in J_i}\neg\,\phi_j\Big)\,.$$

Applying the closure operator, we then have

$$\begin{aligned}
\sim f &\equiv \sim\bigsqcup_{i\in I}\Big(\phi_i \wedge \bigwedge_{j\in J_i}\neg\,\phi_j\Big) \\
&\equiv \bigsqcup_{i\in I}\sim\Big(\phi_i \wedge \bigwedge_{j\in J_i}\neg\,\phi_j\Big) && \text{// by Proposition 4.1.29} \\
&\equiv \bigsqcup_{i\in I}\sim\Big(\phi_i \wedge \sim\Big(\sum_{j\in J_i}\overline{\phi_j}\,\Big)\Big) && \text{// by Propositions 4.1.31 and 4.1.29} \\
&\equiv \bigsqcup_{i\in I}\sim\Big(\phi_i + \sum_{j\in J_i}(\phi_i \wedge \overline{\phi_j}\,)\Big) && \text{// by Proposition 4.1.21} \\
&\equiv \bigsqcup_{i\in I}\Big(\sim\phi_i \wedge \bigwedge_{j\in J_i}\sim(\phi_i \wedge \overline{\phi_j}\,)\Big) && \text{// by Proposition 4.1.29} \\
&\equiv \bigsqcup_{i\in I}\Big(\neg\,\overline{\phi_i}\ \wedge \bigwedge_{j\in J_i}\neg\,\overline{\phi_i \wedge \overline{\phi_j}}\,\Big) && \text{// by Proposition 4.1.31} \\
&\equiv \bigsqcup_{i\in I}\neg\,\Big(\overline{\phi_i}\ \sqcup \bigsqcup_{j\in J_i}\overline{\phi_i \wedge \overline{\phi_j}}\,\Big)\,.
\end{aligned}$$

Since, for all i and j, both $\overline{\phi_i}$ and $\overline{\phi_i \wedge \overline{\phi_j}}$ are interaction formulas, we conclude that $\sim f$ belongs to the lattice generated by interaction formulas through union and complementation. □

4.1.2 Normal form and axiomatisation of PCL formulas

To simplify the presentation, we assume in this subsection that disjunction can appear only within interaction formulas, i.e. we do not consider the extension (4.9) of the disjunction

1. $$\frac{g \wedge \bigsqcup_{i\in I} f_i}{\bigsqcup_{i\in I} g \wedge f_i}$$ (Proposition 4.1.5)

2. $$\frac{g + \bigsqcup_{i\in I} f_i}{\bigsqcup_{i\in I} f_i + g}$$ (Proposition 4.1.7)

3. $$\frac{\neg \bigsqcup_{i\in I} f_i}{\bigwedge_{i\in I} \neg f_i}$$ (Proposition 4.1.5)

4. $$\frac{\neg \sum_{\phi\in\Phi} \phi\,, \quad \text{all } \phi \text{ are interaction formulas}}{\bigsqcup_{\phi\in\Phi} \overline{\phi} \sqcup \sim\Big(\bigwedge_{\phi\in\Phi} \overline{\phi}\Big)}$$ (Proposition 4.1.36)

5. $$\frac{\sum_{\phi\in\Phi} \phi \wedge \sum_{\psi\in\Psi} \psi\,, \quad \text{all } \phi \in \Phi \text{ and } \psi \in \Psi \text{ are interaction formulas}}{\sum_{\xi\in\Phi\cup\Psi} \Big(\xi \wedge \bigvee_{(\phi,\psi)\in\Phi\times\Psi} (\phi \wedge \psi)\Big)}$$ (Proposition 4.1.37)

Figure 4.6 – Rewriting system for computing the normal form by the procedure in Figure 4.7

operator to general PCL formulas. We prove that any PCL formula can be expressed in the following normal form: $\bigsqcup_{i\in I} \sum_{j\in J_i} \bigvee_{k\in K_{i,j}} m_{i,j,k}$, where all $m_{i,j,k}$ are monomials. This normal form can be obtained using the rewriting system given in Figure 4.6 and usual Boolean transformations for interaction formulas. Notice that no two rules can be simultaneously applicable to the same node. Normal form of a formula is computed by applying the procedure in Figure 4.7 to the root of its Abstract Syntax Tree (AST).

An application of a rule to a node of an AST modifies only the subtree rooted at this node. In order to simplify the reasoning, we impose the following additional constraint on the order of application of the rules from the rewriting system in Figure 4.6.

Constraint 4.1.40. We require that any rule be applied to a node n only if no rule can be applied to any other node in the subtree of n.

Remark 4.1.41. We extend Constraint 4.1.40 to include usual Boolean transformations. Hence, at every step of the process, all interaction sub-formulas are maintained in Disjunctive Normal Form.

```
procedure normalise (node)
  if (node is an interaction formula)
    transform node into DNF;
    return;
  endif

  foreach child of node do
    normalise(child);
  od

  if (no rule applicable to node)
    return;
  else
    apply rule to node;
    normalise(node);
  endif
end
```

Figure 4.7 – Procedure for computing the normal form using the rewriting system of Figure 4.6

Example 4.1.42. The following example illustrates the normalization process:

$$\begin{aligned}
(pq \sqcup r) \wedge (pr + \neg\, q) \equiv\ & (pq \sqcup r) \\
& \wedge\ (pr + (\overline{q}\ \sqcup \overline{q}\ + \mathit{true})) && // \text{ rule 4 with } \Phi = \{q\} \\
\equiv\ & (pq \sqcup r) \wedge (pr + \overline{q}\ + \mathit{true}) && // \text{ absorption laws} \\
\equiv\ & (pq \wedge (pr + \overline{q}\ + \mathit{true})) \\
& \sqcup\ (r \wedge (pr + \overline{q}\ + \mathit{true})) && // \text{ rule 1} \\
\equiv\ & ((pq \wedge pr) + (pq \wedge \overline{q}\,) + (pq \wedge \mathit{true})) \\
& \sqcup\ ((r \wedge pr) + (r \wedge \overline{q}\,) + (r \wedge \mathit{true})) && // \text{ rule 5} \\
\equiv\ & (pqr + \mathit{false} + pq) \sqcup (pr + r\overline{q}\ + r) && // \text{ Boolean laws} \\
\equiv\ & pr + r\overline{q}\ + r\,. && // \text{ absorption and identity laws}
\end{aligned}$$

The first step removes the complementation. Then the application of distributivity rules pushes conjunction down in the expression tree of the formula, to the level of monomials. Finally, the formula is simplified, by observing that $false$ is the absorbing element of coalescing and the identity of union.

Theorem 4.1.43. *Under Constraint 4.1.40, the rewriting system in Figure 4.6 has the following properties:*

1. *The rewriting system is terminating and confluent.*
2. *For any formula f' derived from a formula f by the application of rewriting rules, we have $f' \equiv f$.*

3. Any irreducible formula is in the normal form $\bigsqcup_{i \in I} \sum_{j \in J_i} \bigvee_{k \in K_{i,j}} m_{i,j,k}$.

Proof. 1. In order to prove that the rewriting system is terminating, we define a ranking function on the AST of a formula, with leaves representing interaction sub-formulas. First, we introduce the following notations:

- Denote $p(n)$ the number of nodes in the subtree with the root n.
- Denote $d(n)$ the depth of the node n in the AST of the formula.
- Let $N = \sum_n p(n)^{p(n)}$, where the sum is taken over all $\neg$-nodes.
- Let $C = \sum_n p(n)^{p(n)}$, where the sum is taken over all $\wedge$-nodes.
- Let $U = \sum_n d(n)$, where the sum is taken over all $\sqcup$-nodes.
- Denote A the number of $+$-nodes in the AST of the formula.

The ranking function associates a tuple to a tree $\langle N, C, U, A \rangle$. We use lexicographical order to compare the values of the function, i.e. $\langle a_1, a_2, a_3, a_4 \rangle < \langle b_1, b_2, b_3, b_4 \rangle$ iff there exists $i \leq 4$ such that $a_j = b_j$, for all $j < i$, and $a_i < b_i$. We show that application of each rewriting rule strictly reduces the value of the ranking function.

- Rule 1 does not change N and reduces C. Let n be the $\wedge$-node, to which we apply the Rule 1. For each $\wedge$-node n', generated by the application of the rule, we have $p(n') < p(n)$. The number of generated $\wedge$-nodes n' is less than $p(n)$. Hence, $p(n)^{p(n)} > p(n) * p(n')^{p(n')}$, which implies that the value of C decreases after the application of the rule. Although, application of Rule 1 increases the value of U, the ranking function decreases by the definition of the lexicographical order.
- Application of Rule 2 increases A, but decreases U as it transforms a non-empty set of $\sqcup$-nodes into one with smaller depth. This rule does not change the values of N or C.
- Application of Rule 3 decreases N. A $\neg$-node with value $p(n)^{p(n)}$ is transformed into less than $p(n)$ nodes of value less than $p(n')^{p(n')}$ with $p(n') < p(n)$.
- Application of Rule 4 decreases N. It transforms a $\neg$-node into a union of conjunctions and coalescing.
- Application of Rule 5 decreases C and does not change N. It transforms a $\wedge$-node into a coalescing of interaction formulas.
- Application of usual Boolean transformations makes modifications only inside leaves, thus this rule does not affect the function value.

Since all rewriting rules decrease the rank of the tree and each value is a tuple of finite natural numbers, any sequence of applications of rewriting rules is finite.

Notice that applications of rules in different subtrees do not interfere and the order of rule applications between subtrees does not affect the resulting formula. This observation, together with Constraint 4.1.40, guarantees the confluence of the rewriting system.

2. Since all rewriting rules in Figure 4.6 preserve equality, the formula obtained by application of these rules is necessarily equal to the initial one.

3. Let f be an irreducible formula and let T be an AST of f. Any node of T can be represented by the expression $x \to (n_1, \dots, n_k)$, where $x \in \{\sqcup, +, \neg, \wedge\}$ is an operator and $(n_1, \dots, n_k)$ is the list of child nodes. We call such a node $x \to (n_1, \dots, n_k)$ an *x-node.* Notice that, since all operators of the Configuration Logic are associative, an x-node can always be merged with its immediate child x-nodes: let $n_1 = x \to (m_1, \dots, m_l)$, then $x \to (n_1, \dots, n_k)$ can be substituted by $x \to (m_1, \dots, m_l, n_2, \dots, n_k)$ without changing the meaning of the formula (similarly for all n_i). Henceforth, we assume that no child node of an x-node is an x-node.

Let n be a $\neg$-node in T, such that none of the nodes in the sub-tree rooted in n is a $\neg$-node. Let n' be a child node of n. Since Rules 3 and 4 cannot be applied to n, the node n' is neither a $\sqcup$-node, nor a node representing an interaction formula. Hence, n' corresponds to a conjunction or a coalescing of PCL formulas, among which at least one is not an interaction formula. Notice that in the subtree rooted at n' there are neither $\neg$-nodes by the assumption that n is the deepest one nor $\sqcup$-nodes since Rules 1, 2 and 3 cannot be applied. Let n'' be the deepest coalescing node in the subtree rooted at n'. Children of n'' are interaction formulas as subtrees rooted at n'' cannot contain $\neg$-, $\sqcup$- or $+$-nodes. The parent node of n'' cannot be a negation or a union since they cannot appear in the subtree rooted at n', it is not a coalescing due to the form of AST and it is not a conjunction since Rule 5 is not applicable. This contradicts to the assumption that there are $\neg$-nodes in the AST.

Since none of the Rules 1, 2 and 3 are applicable, a $\sqcup$-node can only be the root of the AST of f. Hence, since Rule 5 is not applicable and there are no $\neg$-nodes in the AST of f, a $+$-node can only be the root or a child of the $\sqcup$-node. Furthermore, for the same reason, the children of a $+$-node can only be interaction formulas.

Since all interaction sub-formulas are in their DNF forms (see Remark 4.1.41), we conclude that f is in normal form. $\square$

A *full monomial* is a monomial, which involves all ports, i.e. $m = \bigwedge_{p \in P_+} p \wedge \bigwedge_{p \in P_-} \overline{p}$ such that $P = P_+ \cup P_-$ and $P_+ \cap P_- = \emptyset$. We define a *full normal form* as $\bigsqcup_{i \in I} \sum_{j \in J} m_{i,j,k}$, where $m_{i,j,k}$ are full monomials. We show that any formula has an equivalent full normal form.

Lemma 4.1.44. *A formula $f = \sum_{i \in I} m_i$, where m_i are full monomials, is satisfied by* *exactly one configuration $\gamma = \{a_i\}_{i \in I}$, where a_i is an interaction corresponding to the full monomial m_i: $m_i = \bigwedge_{p \in a_i} p \wedge \bigwedge_{p \notin a_i} \overline{p}$.*

Proof. Since m_i is a full monomial, there exists exactly one valuation of ports such that the monomial evaluates to true, i.e. there exists exactly one interaction a_i such that $a_i \models_i m_i$.

$\gamma \models \sum_{i \in I} m_i$ iff there exists $\{\gamma_i\}_{i \in I}$ such that $\gamma = \bigcup_{i \in I} \gamma_i$ and, for all $i \in I$, $\gamma_i \models m_i$. For each m_i there exists only one interaction and consequently only one configuration $\gamma_i \models m_i$. Thus, there exists exactly one γ, such that $\gamma \models f$. ☐

Theorem 4.1.45. *Any formula f has a unique full normal form.*

Proof. By Theorem 4.1.43 any formula f can be rewritten as a formula $f' \equiv f$ in normal form. In f', any non-full monomial can be transformed into a disjunction of full monomials, which, by Corollary 4.1.8, can be further transformed into a union of coalesced full monomials. The application of Proposition 4.1.7 leads to the full normal form. Uniqueness is a corollary of Lemma 4.1.44. ☐

Example 4.1.46. Let $P = \{p, q, r\}$ and consider the normal form formula $pr + r\overline{q}$. It can be transformed into full normal form as follows:

$$\begin{aligned} pr + r\overline{q} \; &\equiv (pqr \sqcup p\overline{q}\,r \sqcup pqr + p\overline{q}\,r) + (p\overline{q}\,r \sqcup \overline{p}\;\overline{q}\,r \sqcup p\overline{q}\,r + \overline{p}\;\overline{q}\,r) \\ &\equiv (pqr + p\overline{q}\,r) \sqcup (pqr + \overline{p}\;\overline{q}\,r) \sqcup (pqr + p\overline{q}\,r + \overline{p}\;\overline{q}\,r) \sqcup p\overline{q}\,r \sqcup (p\overline{q}\,r + \overline{p}\;\overline{q}\,r) \sqcup (p\overline{q}\,r + \overline{p}\;\overline{q}\,r) \\ &\qquad \sqcup (pqr + p\overline{q}\,r) \sqcup (pqr + p\overline{q}\,r + \overline{p}\;\overline{q}\,r) \sqcup (pqr + p\overline{q}\,r + \overline{p}\;\overline{q}\,r). \end{aligned}$$

4.1.3 Soundness and completeness

Axioms. PCL operators satisfy the following axioms:

1. The PIL axioms for interaction formulas.
2. The usual axioms of propositional logic for $\sqcup$, $\wedge$, $\neg$.
3. $+$ is associative, commutative and has an absorbing element *false*.
4. For any formulas f, f_1 and f_2, holds $f + (f_1 \sqcup f_2) \equiv f + f_1 \sqcup f + f_2$.
5. For any sets of interaction formulas Φ and Ψ, holds
$$\sum_{\phi \in \Phi} \phi \wedge \sum_{\psi \in \Psi} \psi \; \equiv \sum_{\xi \in \Phi \cup \Psi} \Big(\xi \wedge \bigvee_{(\phi, \psi) \in \Phi \times \Psi} (\phi \wedge \psi) \Big).$$
6. For any set of interaction formulas Φ, holds
$$\neg \Big(\sum_{\phi \in \Phi} \phi \Big) \equiv \bigsqcup_{\phi \in \Phi} \overline{\phi} \sqcup \sim\! \Big(\bigwedge_{\phi \in \Phi} \overline{\phi} \Big) .$$

Theorem 4.1.47. *The above axiomatization is sound and complete for the equality in PCL.*

Proof. Soundness of all the above axioms has been proved in previous sections. Completeness is an immediate consequence of the existence of a unique full normal form. □

4.1.4 Checking satisfaction of PCL formulas

We provide a method for checking that a configuration of the form $\gamma = \{a_1, \ldots, a_n\}$ satisfies a formula f. Without loss of generality, we assume that the formula is in normal form $f = \bigsqcup_{i \in I} \sum_{j \in J_i} \bigvee_{k \in K_{i,j}} m_{i,j,k}$. We have to check that γ satisfies at least one of the terms $\sum_{j \in J_i} \bigvee_{k \in K_{i,j}} m_{i,j,k}$, for some $i \in I$. Algorithm 1 performs this verification for one term (index i is omitted).

Algorithm 1: Algorithm for checking satisfaction of formulas

```
Data: A sub-formula $f = \sum_{j \in J} \bigvee_{k \in K_j} m_{j,k}$ and a configuration $\gamma = \{a_1, \ldots, a_n\}$.
Result: Returns true if $\gamma \models f$, otherwise it returns $false$.
begin
    $J' = \emptyset$;
    $l = 1$;
    $b = true$;

    while ($l \leq n$ and $b$) do
        $X = \{j \in J \mid a_l \models_i \bigvee_{k \in K_j} m_{j,k}\}$;
        if $X \neq \emptyset$ then
            $J' = J' \cup X$;
        else
            $b = false$;
        $l = l + 1$;
    return $J' = J$;
```

We have to check the validity of the following two statements: 1) each interaction satisfies at least one interaction formula and 2) each interaction formula is satisfied by at least one interaction. The algorithm iterates through the interactions, checking the first part and memorising the satisfied interaction formulas. After the iteration stops, it checks whether all interaction formulas were satisfied by at least one interaction. Configuration γ satisfies the formula f iff the disjunction of the results of Algorithm 1, for all terms of the union, evaluates to true.

An alternative method for checking satisfaction of a formula f by a configuration γ is based on the existence of a normal form and the completeness theorem.

Consider a formula f and a configuration $\gamma = \{a_1, ..., a_n\}$. In order to decide whether $\gamma \models f$, we associate with γ a characteristic formula $\varphi_\gamma = m_1 + \cdots + m_n$, where $m_i = \bigwedge_{p \in a_i} p \wedge \bigwedge_{p \notin a_i} \overline{p}$ are characteristic monomials of the respective interactions a_i. Notice that φ_γ has exactly one model γ (Lemma 4.1.44). If formulas φ_γ and $\neg f$ have no common models then γ is a model of f. Thus, $\gamma \models f$ iff $\varphi_\gamma \wedge \neg f \equiv false$. This latter equality can be decided by verifying whether all terms of the normal form of $\varphi_\gamma \wedge \neg f$ are equal to $false$. Recall that the terms of a formula in normal form are coalescings of interaction formulas. Therefore, for a term to be equal to $false$, it is sufficient that one of its participating interaction formulas be equal to $false$. Finally, as in Boolean logics, a disjunction of monomials is equal to $false$ iff all monomials contain one of the variables at least twice in opposite (positive and negative) forms.

Example 4.1.48. Let $P = \{p, q, r\}$ and consider $f = p\,q + r\,\overline{p}$ and $\gamma = \{\{p, q, r\}, \{q, r\}\}$. In order to decide whether $\gamma \models f$, we first apply Algorithm 1. This algorithm iterates through the interactions of γ and monomials of f: $\{p, q, r\}$ satisfies $p\,q$, whereas $\{q, r\}$ satisfies $r\,\overline{p}$. For both interactions the sets of monomials are not empty and all monomials were visited. Hence, $\gamma \models f$.

Alternatively, we consider the characteristic formula $\varphi_\gamma = p\,q\,r + q\,r\,\overline{p}$ and check whether $\varphi_\gamma \wedge \neg f = (p\,q\,r + q\,r\,\overline{p}) \wedge \neg (p\,q + r\,\overline{p}) \equiv false$. We have:

$$\begin{aligned}
&(pqr + qr\overline{p}) \wedge \neg (pq + r\overline{p}) \\
&\quad \equiv (pqr + qr\overline{p}) \wedge \Big((\overline{p} \vee \overline{q}) \sqcup (\overline{r} \vee p) \sqcup \sim((\overline{p} \vee \overline{q}) \wedge (\overline{r} \vee p))\Big) \\
&\quad \equiv \Big((pqr \wedge (\overline{p} \vee \overline{q})) + (qr\overline{p} \wedge (\overline{p} \vee \overline{q}))\Big) \sqcup \Big((pqr \wedge (\overline{r} \vee p)) + (qr\overline{p} \wedge (\overline{r} \vee p))\Big) \\
&\qquad \sqcup \Big((pqr + qr\overline{p}) \wedge ((\overline{p} \vee \overline{q}) \wedge (\overline{r} \vee p) + true)\Big) \\
&\quad \equiv (false + \overline{p}\,qr) \sqcup (pqr + false) \sqcup \Big((pqr + qr\overline{p}) \wedge ((\overline{p}\,\overline{r} \vee \overline{q}\,\overline{r} \vee p\,\overline{q}) + true)\Big) \\
&\quad \equiv false \sqcup false \sqcup \Big((pqr + qr\overline{p}) \wedge ((\overline{p}\,\overline{r} \vee \overline{q}\,\overline{r} \vee p\,\overline{q}) + true)\Big) \\
&\quad \equiv (pqr \vee qr\overline{p}) \wedge pqr + (pqr \vee qr\overline{p}) \wedge qr\overline{p} \\
&\qquad + (pqr \vee qr\overline{p}) \wedge (\overline{p}\,\overline{r} \vee \overline{q}\,\overline{r} \vee p\,\overline{q}) + (pqr \vee qr\overline{p}) \wedge true \\
&\quad \equiv pqr + qr\overline{p} + false + (pqr \vee qr\overline{p}) \equiv false\,.
\end{aligned}$$

4.2 First and second order extensions of PCL

PCL is defined for a given set of components and a given set of ports. On the contrary, architecture styles are defined for arbitrary number of components. In order to specify architecture styles, we introduce types of components and quantification over component variables. We make the following assumptions:

- A finite set of component types $\mathcal{T} = \{T_1, \ldots, T_n\}$ is given. Instances of a component type have the same interface and behaviour. We write $c : T$ to denote a component c of type T. Additionally, we denote C_T the set of all the components of type T. Finally, we assume the existence of the universal component type U, such that any component or component set is of this type. Thus, C_U represents all the components of a model.

- The interface of each component type has a distinct set of ports. We write $c.p$ to denote the port p of component c and $c.P$ to denote the set of ports of component c.

4.2.1 First-order configuration logic

Syntax. The language of the formulas of the first-order configuration logic is an extension of the PCL language by allowing Boolean expressions on component variables, existential quantification and a specific coalescing quantifier $\Sigma c{:}T$.

$$F ::= \mathit{true} \mid \phi \mid \exists c{:}T\big(\Phi(c)\big).F \mid \Sigma c{:}T\big(\Phi(c)\big).F \mid F \sqcup F \mid \neg\, F \mid F + F\,,$$

where ϕ is an interaction formula, c is a component variable and $\Phi(c)$ is some set-theoretic predicate on c (omitted when $\Phi = \mathit{true}$).

Additionally, we define the usual notation for universal quantifier:

$$\forall c{:}T\big(\Phi(c)\big).F \overset{def}{=} \neg\, \exists c{:}T\big(\Phi(c)\big).\neg\, F.$$

Semantics. The semantics is defined for closed formulas, where, for each variable in the formula, there is a quantifier over this variable in an upper nesting level. We assume that a finite set of component types $\mathcal{T} = \{T_1, \dots, T_n\}$ is given. Models are pairs $\langle B, \gamma\rangle$, where B is a set of component instances of types from $\mathcal{T}$ and γ is a configuration on the set of ports P of these components. For quantifier-free closed formulas the semantics is the same as for PCL formulas. For closed formulas with quantifiers the satisfaction relation is defined by the following rules:

$$\begin{aligned}
&\langle B,\gamma\rangle \models \exists c{:}T\,\big(\Phi(c)\big)\ .\ F\,, &&\text{iff}\quad \gamma \models \bigsqcup_{c'{:}T\in B\,\wedge\,\Phi(c')} F[c'/c]\,,\\
&\langle B,\gamma\rangle \models \Sigma c{:}T\,\big(\Phi(c)\big)\ .\ F\,, &&\\
&\qquad\text{iff}\quad \{c' : T\in B \mid \Phi(c')\} \neq \emptyset\ \wedge\ \gamma \models \sum_{c'{:}T\in B\,\wedge\,\Phi(c')} F[c'/c]\,, &&
\end{aligned}$$

where $c' : T$ ranges over all component instances of type $T \in \mathcal{T}$ satisfying Φ and $F[c'/c]$ is obtained by replacing all occurrences of c in F by c'.

For a more concise representation of formulas, we introduce the following additional notation:

$$\begin{aligned}
\natural(c_1.p_1, \dots, c_n.p_n) \overset{def}{=}\ & \bigwedge_{i=1}^{n} c_i.p_i\ \wedge \bigwedge_{i=1}^{n}\ \bigwedge_{p\in c_i.P\setminus\{p_i\}} \overline{c_i.p}\\
& \wedge \bigwedge_{T\in\mathcal{T}} \left(\forall c{:}T\big(c \notin \{c_1,\dots,c_n\}\big).\ \bigwedge_{p\in c.P} \overline{c.p}\right).
\end{aligned}$$

The $\sharp(c_1.p_1, \dots, c_n.p_n)$ notation expresses an exact interaction, i.e. all ports in the arguments must participate in the interaction and all other ports of the system cannot participate in the interaction. If $\langle B, \gamma\rangle$ is a model, it can be shown that:

$$\langle B, \gamma\rangle \models \sharp(c_1.p_1, c_2.p_2, \dots, c_n.p_n) \quad \text{iff } c_1, c_2, \dots, c_n \in B \text{ and } \gamma = \big\{\{c_1.p_1,\ c_2.p_2, \dots, c_n.p_n\}\big\} .$$

The following three examples illustrate the specification of simple interactions.

Example 4.2.1 (Single interaction)**.** Assume that there is only one type of components T with a single port p. We characterize models with a single interaction $\{c_1.p, c_2.p\}$.

The formulas $c_1.p\ c_2.p$ and $\sim(c_1.p\ c_2.p)$ do not ensure the presence of interaction $\{c_1.p, c_2.p\}$, since the model with $\gamma = \big\{\{c_1.p, c_2.p, c_3.p\}\big\}$ satisfies these formulas. The correct specification can be expressed by a monomial, which contains all the negated ports that are not included in the interaction:

$$c_1.p \wedge c_2.p \wedge \forall c{:}T\big(c \notin \{c_1, c_2\}\big).\ \overline{c.p}\ . \tag{4.11}$$

This formula is can be equivalently rewritten using the $\sharp$ notation introduced above: $\sharp(c_1.p,\ c_2.p)$.

Example 4.2.2 (No interaction of arity greater than two)**.** Assume again that all components are of type T with a single port p. We want to express the property that all interactions involve at most two ports.

If we have three components c_1, c_2, c_3 the formula $\overline{c_1.p\ c_2.p\ c_3.p}$ forbids interactions involving all of the components. The desired specification is obtained by requiring that this formula holds for any triple of components:

$$\forall c_1 : T.\ \forall c_2 : T(c_1 \neq c_2).\ \forall c_3{:}T\big(c_3 \notin \{c_1, c_2\}\big).(\overline{c_1.p\ c_2.p\ c_3.p}\,).$$

Example 4.2.3 (Binary interactions)**.** Let us further restrict the specification of Example 4.2.2, so that all binary interactions among components are included. The monomial $c_1.p\ c_2.p$ represents all interactions involving ports $c_1.p$ and $c_2.p$. In order to allow other interactions, we take the closure of this formula, i.e. $\sim(c_1.p\ c_2.p)$. To obtain the required specification, we conjunct that of Example 4.2.2 with $\sim(c_1.p\ c_2.p)$ for each pair of components as follows:

$$\forall c_1 : T.\ \forall c_2 : T(c_1 \neq c_2).\ \sim(c_1.p\ c_2.p) \\ \wedge\ \forall c_1 : T.\ \forall c_2 : T(c_1 \neq c_2).\ \forall c_3{:}T\big(c_3 \notin \{c_1, c_2\}\big).(\overline{c_1.p\ c_2.p\ c_3.p}\,)\,.$$

The following examples illustrate the specification of architecture styles and patterns.

Example 4.2.4. The Star architecture style, illustrated in Figure 4.8, is defined for a set of components of the same type. One central component s is connected to every other

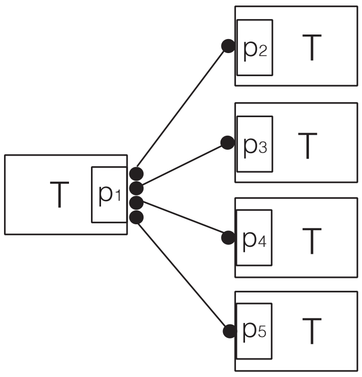


Figure 4.8 – A Star architecture

component through a binary interaction and there are no other interactions. It can be specified as follows:

$$\exists s{:}T.\ \forall c{:}T(c \neq s).\ \Big(\sim(c.p\ s.p)\ \wedge\ \forall c'{:}T(c' \notin \{c,s\}).\ (\overline{c'.p\ c.p}\,)\Big) \wedge \neg\big(\exists c{:}T.\ \sim\sharp(c.p)\big)\,. \tag{4.12}$$

The three conjuncts of this formula express, respectively, the properties: 1) any component is connected to the center; 2) components other than the center are not connected among themselves; and 3) unary interactions are forbidden.

Notice that the semantics of the first conjunct in (4.12), $\forall c : T(c \neq s).\ \sim(c.p\ s.p)$, is a conjunction of closure formulas. In this conjunct, the closure operator also allows interactions in addition to the ones explicitly defined. Therefore, to correctly specify this style, we forbid all other interactions by using the second and third conjuncts of the specification. A simpler alternative specification uses the Σ quantifier:

$$\exists s{:}T.\ \Sigma c{:}T(c \neq s).\ \sharp(c.p,\ s.p)\,. \tag{4.13}$$

The $\sharp$ notation requires interactions to be binary and the Σ quantifier allows configurations that contain only interactions satisfying $\sharp(c.p,\ s.p)$, for some c. Thus, contrary to (4.12), we do not need to explicitly forbid unary interactions and connections between non-center components.

Example 4.2.5. The Pipes and Filters architecture style [44] involves two types of components, P and F, each having two ports *in* and *out*. Each input (resp. output) of a filter is connected to an output (resp. input) of a single pipe. The output of any pipe can be connected to at most one filter. One possible configuration is shown in Figure 4.9.

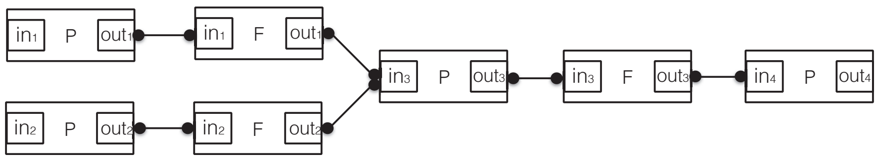


Figure 4.9 – A Pipes and Filters architecture

This style can be specified as follows:

$$\begin{aligned}
&\forall f\!:\!F.\ \exists p\!:\!P.\ \sim\!(f.in\ p.out) \wedge \forall p'\!:\!P(p \neq p').\ \big(\,\overline{f.in\ p'.out}\,\big) && (4.14)\\
\wedge\ &\forall f\!:\!F.\ \exists p\!:\!P.\ \sim\!(f.out\ p.in) \wedge \forall p'\!:\!P(p \neq p').\ \big(\,\overline{f.out\ p'.in}\,\big) && (4.15)\\
\wedge\ &\forall p\!:\!P.\ \exists f\!:\!F.\ \forall f'\!:\!F(f \neq f').\ \big(\,\overline{p.out\ f'.in}\,\big) && (4.16)\\
\wedge\ &\forall p\!:\!P.\ \Big(\overline{p.in\,p.out} \wedge \forall p'\!:\!P(p \neq p').\ \big(\overline{p.in\,p'.in} \wedge \overline{p.in\,p'.out}\,\big)\Big) && (4.17)\\
\wedge\ &\forall f\!:\!F.\ \Big(\overline{f.in\,f.out} \wedge \forall f'\!:\!F(f \neq f').\ \big(\overline{f.in\,f'.in} \wedge \overline{f.in\,f'.out}\,\big)\Big)\,. && (4.18)
\end{aligned}$$

The first conjunct (4.14) requires that the input of each filter be connected to the output of a single pipe. The second conjunct (4.15) requires that the output of each filter be connected to the input of a single pipe. The third conjunct (4.16) requires that the output of a pipe be connected to at most one filter. Finally, the fourth and fifth conjuncts (4.17) and (4.18) require that pipes only be connected to filters and vice-versa.

Notice that (4.14) and (4.15) in Example 4.2.5 can be simplified by introducing the additional notation for "exists unique":

$$\exists! c\!:\!T\big(\Phi(c)\big).\ F(c) \overset{def}{=} \exists c\!:\!T\big(\Phi(c)\big).\ F(c) \wedge \forall c'\!:\!T\big(c \neq c' \wedge \Phi(c')\big).\ \neg F(c')\,. \qquad (4.19)$$

Using this notation, (4.14) and (4.15) can be rewritten, respectively, as

$$\forall f\!:\!F.\ \exists! p\!:\!P.\ \sim\!(f.in\ p.out) \qquad \text{and} \qquad \forall f\!:\!F.\ \exists! p\!:\!P.\ \sim\!(f.out\ p.in)\,.$$

Example 4.2.6. In the Blackboard architecture style [32], a blackboard component of type B holds data[1] that may be updated by a group of knowledge sources of type S. A controller of type C enforces mutual exclusion of write access. Figure 4.10 depicts a model with three knowledge sources. We provide specifications of models composed of: 1) a single blackboard b with two ports *sh* (share) and *ctrl* (control); 2) a single controller c with a port *ctrl*; and 3) a set of knowledge sources with a port *acc* (access). No knowledge can be shared without taking control of the blackboard through the *ctrl* port.

[1]We omit the data representation in this example, since only the fact that the data is updated is relevant and not the data itself.

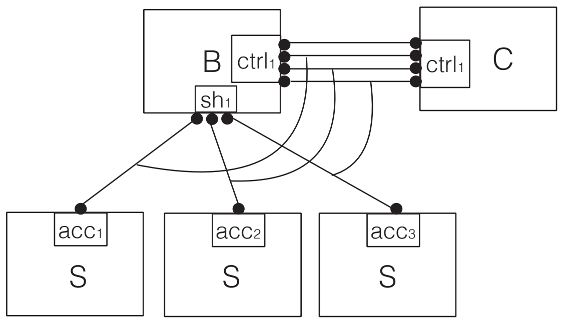


Figure 4.10 – A Blackboard architecture

The Blackboard architecture style can be specified as follows:

$$b.ctrl \wedge c.ctrl \wedge \sim\Big(\Sigma s\!:\!S.\ (s.acc\ b.sh)\Big) \wedge \Big(\forall s_1 : S.\ \forall s_2 : S(s_1 \neq s_2).\ (\overline{s_1.acc\ s_2.acc}\,)\Big)\,.$$

The first two conjuncts require that the control ports of blackboard and controller components participate in all interactions. The third conjunct requires that all knowledge sources be connected to the blackboard. The last conjunct requires that there be no interactions involving two or more knowledge sources.

Example 4.2.7. The Request/Response pattern involves Clients and Services. It is defined as follows [35]:

“Request/Response begins when the client establishes a connection to the service. Once a connection has been established, the client sends its request and waits for a response. The service processes the request as soon as it is received and returns a response over the same connection. This sequence of client-service activities is considered to be synchronous because the activities occur in a coordinated and strictly ordered sequence. Once the client submits a request, it cannot continue until the service provides a response."

From this informal description we can infer the following. There are two types of components: a client *Cl* and a service *S*. Clients have three ports: *Cl.con*, *Cl.req* and *Cl.rec* that correspond to the connect, request and receive actions defined in the pattern, respectively. Service components have two ports *S.get* for receiving a request and *S.send* for sending a reply to the client that raised a request.

We use a coordinator of type *C* to enforce the properties: 1) only one client can be connected at a time to a service; and 2) a client has to connect to the service before sending a request. A unique coordinator is needed per service and therefore, the number of coordinators must match the numbers of services. There can be arbitrarily many clients. Each coordinator has three ports *con*, *get* and *dsc* that correspond to connect, get a request and disconnect actions. Notice that the behaviour of a coordinator is cyclic involving the

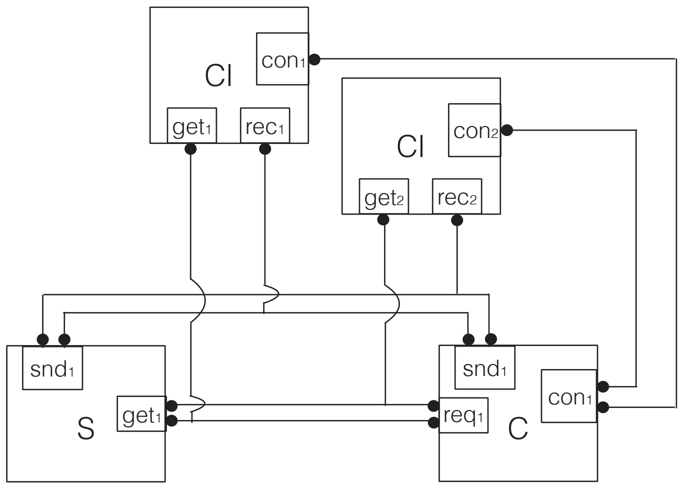


Figure 4.11 – A Request/Response architecture

sequence $con \to get \to dsc \to con$. The Request/Reply pattern is illustrated in Figure 4.11.

This pattern can be specified as follows:

$$
\begin{aligned}
&\Sigma cl{:}Cl.\ \Sigma s{:}S.\ \exists c{:}C.\ \Big(\sharp(cl.con,\ c.con) + \sharp(cl.req,\ s.get,\ c.get) \\
&\qquad\qquad\qquad\qquad\qquad\qquad + \sharp(cl.rec,\ s.send,\ c.dsc)\Big) \\
\wedge\ &\Sigma cl{:}Cl.\ \Sigma c{:}C.\ \exists s{:}S.\ \Big(\sharp(cl.con,\ c.con) + \sharp(cl.req,\ s.get,\ c.get) \\
&\qquad\qquad\qquad\qquad\qquad\qquad + \sharp(cl.rec,\ s.send,\ c.dsc)\Big)\,.
\end{aligned}
$$

Notice that the $\exists$ quantifier has the semantics of union. Coalescing distributes over union. Therefore, the meaning of the nested existential quantifier in the first conjunct is several configurations, where in each configuration a service is connected to a single coordinator.

The property *"a unique coordinator is needed per service"* is enforced by the formula as follows: 1) the first conjunct requires that each service be connected to a single coordinator; and 2) the second conjunct requires that each coordinator be connected to a single service.

Example 4.2.8. The Repository architecture style [31] consists of a repository component r with a port p and a set of data-accessor components of type A with ports q. We provide below a list of increasingly strong properties that may be used to characterize this style:

1. The basic property *"there exists a single repository and all interactions involve it"* is specified as follows:

$$
SingleRepo \overset{def}{=} \exists r{:}R.\ (r.p) \wedge \forall r{:}R.\ \forall r'{:}R.\ (r = r')\,,
$$

where the subterm $\forall r{:}R.\ \forall r'{:}R.\ (r = r')$ can be expressed in the logic as $\forall r{:}R.\ \forall r' :$

$R(r' \neq r).\ false.$

2. The additional property *"there are some data-accessors and any data-accessor must be connected to the repository"* is enforced by extending the formula as follows:

$$\mathit{DataAccessors} \overset{def}{=} \mathit{SingleRepo} \wedge \exists a\!:\!A.\ \mathit{true} \wedge \forall a\!:\!A.\ \exists r\!:\!R.\ \sim\!(r.p\ a.q)\,.$$

4.2.2 First-order configuration logic with ordered components

We present an extension of the first-order logic in which components are ordered and thus, we can define constraints based on component order. As a result, several architecture styles, for instance the Ring and Linear architecture styles, that are not expressible in the first-order logic can now be expressed [67]. Another difference with the first-order logic presented in Section 4.2.1 is that we quantify over port variables instead of component variables. This modification allows us to shorten formulas. It does not have any impact on the expressiveness of the logic since the order of ports and the order of components coincide. We write $T.p$ to denote the port p of the component type T and $T.P$ to denote the interface of T.

Syntax. The syntax of the first-order logic with ordered components is defined by:

$$F \quad ::= \quad \mathit{true} \;\;|\;\; \phi \;\;|\;\; F \sqcup F \;\;|\;\; \neg\, F \;\;|\;\; F + F \;\;|\;\; \wp x[i]\!:\!T.p\,(\Phi(x[i],i)).F \quad ,$$

where ϕ is an interaction formula; $x[i]$, called *port variable*, refers to the i^{th} instance of the generic port $T.p$; $\Phi(x[i],i)$ is a predicate based on set-theoretic operations on ports and arithmetic operations on indices (omitted when $\Phi = \mathit{true}$) and $\wp \in \{\exists, \Sigma, \bigsqcup\}$ is a quantifier, corresponding, respectively, to union, coalescing and disjunction operators.

Semantics. The semantics is defined for closed formulas. Models are architectures $\langle \mathcal{B}, \gamma \rangle$, where $\mathcal{B}$ is a set of component instances of types from $\mathcal{T}$ and γ is a configuration on the set of ports P of these components.

We denote n_T the number of components of type T. Within each component type, components are ordered linearly, i.e. they can be represented by an array with the index ranging from 1 to n_T. Port instances inherit the same ordering: if a component type T has a generic port $T.p$, then its i^{th} instance, denoted p_i,[2] belongs to the i^{th} component instance of type T.

For quantifier-free closed formulas, the semantics is the same as for PCL formulas. For

[2]To keep full information, the i^{th} instance of a generic port $T.p$ should be denoted $(T.p)_i$. However, since the type T will always be clear from the context, we omit it to simplify the notation.

quantifiers, the satisfaction relation is defined as follows:

$$\langle \mathcal{B}, \gamma \rangle \models \exists x[i] \mathbin{:} T.p \left(\Phi(x[i], i)\right) \,.\, F\,, \qquad \text{iff} \quad \gamma \models \bigsqcup_{\substack{j \in [1, n_T] \\ \text{s.t. } \Phi(p_j, j)}} F[p_j/x[i]]\,,$$

$$\langle \mathcal{B}, \gamma \rangle \models \Sigma x[i] \mathbin{:} T.p \left(\Phi(x[i], i)\right) \,.\, F\,,$$
$$\text{iff} \quad \{j \in [1, n_T] \,|\, \Phi(p_j, j)\} \neq \emptyset \ \wedge \ \gamma \models \sum_{\substack{j \in [1, n_T] \\ \text{s.t. } \Phi(p_j, j)}} F[p_j/x[i]]\,,$$

$$\langle \mathcal{B}, \gamma \rangle \models \text{Ɑ} x[i] \mathbin{:} T.p \left(\Phi(x[i], i)\right) \,.\, F\,, \qquad \text{iff} \quad \gamma \models \bigvee_{\substack{j \in [1, n_T] \\ \text{s.t. } \Phi(p_j, j)}} F[p_j/x[i]]\,,$$

where j ranges over all indices of component instances of type $T \in \mathcal{T}$, such that p_j and j satisfy Φ; and $F[p_j/x[i]]$ is the formula obtained by replacing all occurrences of the port variable $x[i]$ in F by the port instance p_j.

For the sake of clarity, we introduce the following additional notations:

- The universal quantifier:

$$\forall x[i] \mathbin{:} T.p \left(\Phi(x[i], i)\right) \,.\, F \ \stackrel{def}{=} \ \neg \Big(\exists x[i] \mathbin{:} T.p \left(\Phi(x[i], i)\right) \,.\, \neg F\Big)\,.$$

- Quantification over components:

$$\wp x[i] \mathbin{:} T \left(\Phi(i)\right) \,.\, F \ \stackrel{def}{=} \ \wp x[i] \mathbin{:} T.p \left(\Phi(i)\right) \,.\, F\,,$$

  for an arbitrary $p \in T.P$ and $\wp \in \{\exists, \Sigma, \text{Ɑ}, \forall\}$. Here, the predicate Φ does not depend on $x[i]$, since this variable refers to a component and not to a port.

- For an arbitrary set of port instances $S \subseteq U.P$ (not necessarily instances of the same generic port):

$$\sharp S \ \stackrel{def}{=} \ \bigwedge_{s \in S} s \ \wedge \forall x[i] \mathbin{:} U.P\big(x[i] \notin S\big) \,.\, \overline{x[i]}\ .$$

The following examples illustrate the specification of architecture styles.

Example 4.2.9. The Repository architecture style, also presented in Example 4.2.8, consists of one repository component of type R with a port p and a set of data-accessor components of type A with ports q. We require that any data-accessor component must be connected to

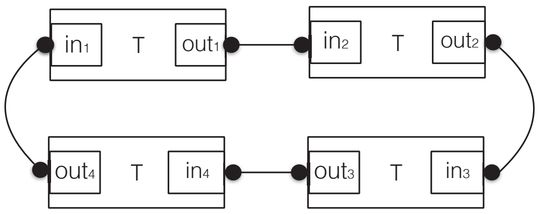


Figure 4.12 – A Ring architecture

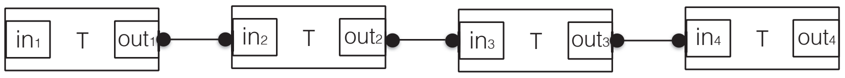


Figure 4.13 – A Linear architecture

the repository. The style can be specified as follows:

$$\forall r[i]\!:\!R\ .\ \forall r'[j]\!:\!R\ (i \neq j)\ .\ false\ \wedge\ \exists a[i]\!:\!A\ .\ true \\ \wedge\ \exists x[i]\!:\!R.p\ .\ \Big(x[i]\ \wedge\ \forall y[j]\!:\!A.q\ .\ \sim(x[i]\ y[j])\Big)\,.$$

This formula additionally enforces the existence of a single repository and at least one data-accessor.

Example 4.2.10. Consider the Request/Response pattern described in Example 4.2.7. The specification of this pattern can be simplified by requiring connections between pairs of services and coordinators with equal indices. The Request/Response pattern can be specified in the first-order configuration logic with ordered components as follows:

$$\Sigma cl[i]\!:\!Cl.\ \Sigma s[j]\!:\!S.\ \exists c[k]\!:\!C(k=j).\ \Big(\sharp\{cl[i].con,\ c[k].con\} \\ +\ \sharp\{cl[i].req,\ s[j].get,\ c[k].get\} + \sharp\{cl[i].rec,\ s[j].send,\ c[k].dsc\}\Big)\,.$$

In the following two examples of this section, we consider systems consisting of components of a single type T with two generic ports *in* and *out*. We assume that every interaction has at least one *in* port and at least one *out* port. Alternatively, this assumption can be enforced by the constraint

$$\neg\ \Big(\forall x[i]\!:\!T.out\ .\ \overline{x[i]}\Big) \wedge \neg\ \Big(\forall x[i]\!:\!T.in\ .\ \overline{x[i]}\Big)\ .$$

Example 4.2.11. The Ring architecture style (Example 4.2.15), can be specified as follows:

$$\Sigma x[i]\!:\!T.out\ .\ \Sigma y[j]\!:\!T.in\ (j = i+1 \mod n_T)\ .\ \sharp\{x[i], y[j]\}.$$

The constraint allows only interactions between neighbouring components.

Example 4.2.12. The Linear architecture style, shown in Figure 4.13, involves serially connected components. It is similar to the Ring architecture style: the difference being that in the Linear architecture style, there are two distinguished components that are the ends of the line such that the input of the first component and the output of the last component are not connected.

$$\Sigma x[i]\!:\!T.out \ . \ \Sigma y[j]\!:\!T.in \ (j = i+1) \ . \ \sharp\{x[i], y[j]\} \, .$$

The formula is similar to the specification of the Ring architecture style. Taking equality instead of the modular one in the constraint forbids the interaction between the first and the last component.

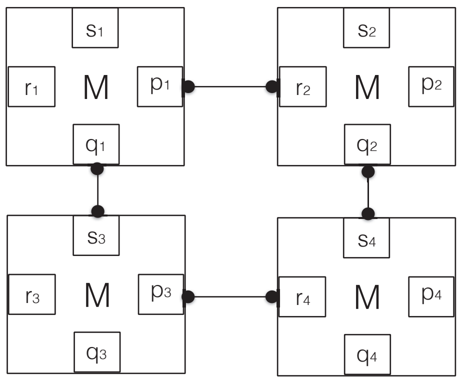


Figure 4.14 – A Square Grid architecture

Example 4.2.13. The Square Grid architecture style, shown in Figure 4.14, involves n^2 components of type T, each with four ports p, q, r and s. Adjacent components are connected through ports p and r in each row of the grid and through ports q and s in each column. It can be specified as follows:

$$\Sigma c[i]\!:\!T \ . \ \Sigma c[j]\!:\!T \ (j = i+1 \wedge i \neq 0 \bmod n) \ . \ \sharp(c[i].p, c[j].q) \\ + \Sigma c[i]\!:\!T \ . \ \Sigma c[j]\!:\!T \ (j = i+n) \ . \ \sharp(c[i].r, c[j].t) \, ,$$

The formula is based on the specification of the Linear architecture style. It requires components be arranged in n horizontal and n vertical lines of length n.

4.2.3 Second-order configuration logic

Properties stating that two components are connected through a chain of interactions, are essential for architecture style specification. In [60, 67], it is shown that transitive closure, necessary to specify such reachability properties, cannot be expressed in first-order logic (without or with ordered components). This motivates the introduction of the second-order configuration logic with quantification over sets of components.

This logic extends the first-order logic with variables ranging over component sets. We write $C\!:\!T$ to express the fact that all components belonging to C are of type T.

Syntax. The syntax of the second-order configuration logic is defined by:

$$\begin{aligned} S ::= {} & \mathit{true} \mid \phi \mid \exists c{:}T(\Phi(c)).S \mid \Sigma c{:}T(\Phi(c)).S \mid S \sqcup S \mid \neg\, S \mid S + S \\ & \mid \exists C : T(\Psi(C)).S \mid \Sigma C : T(\Psi(C)).S\,, \end{aligned}$$

where ϕ is an interaction formula, c is a component variable, C is a component set variable and $\Phi(c)$, $\Psi(C)$ are some set-theoretic predicates (omitted when *true*). Additionally, we define the usual notation for universal quantifier:

$$\forall C{:}T(\Psi(C)).S \overset{def}{=} \neg\, \exists C{:}T(\Psi(C)).\neg\, S.$$

Semantics. The semantics is defined for closed formulas, where, for each variable in the formula, there is a quantifier over this variable in an upper nesting level. Models are pairs $\langle B, \gamma\rangle$, where B is a set of component instances of types from $\mathcal{T}$ and γ is a configuration on the set of ports P of these components. The meaning of quantifier-free formulas or formulas with quantification only over component variables is as for first-order logic. We define the meaning of quantifiers over component set variables as follows:

$$\begin{aligned} & \langle B, \gamma\rangle \models \exists C{:}T\left(\Psi(C)\right)\,.\,S\,, && \text{iff} \quad \gamma \models \bigsqcup_{C'{:}T \in B \,\wedge\, \Psi(C')} S[C'/C]\,, \\ & \langle B, \gamma\rangle \models \Sigma C{:}T\left(\Psi(C)\right)\,.\,S\,, && \\ & \qquad \text{iff} \quad \{C' : T \in B \mid \Psi(C')\} \neq \emptyset \ \wedge\ \gamma \models \sum_{C'{:}T \in B \,\wedge\, \Psi(C')} S[C'/C]\,, && \end{aligned}$$

where $C'{:}T$ ranges over all sets of components of type T that satisfy Ψ.

In the examples of this subsection, we consider systems consisting of components of a single type T with two ports *in* and *out*. We assume that every interaction has at least one *in* port and at least one *out* port. Alternatively, this assumption can be enforced by the constraint $\neg\,(\forall c{:}T.\ \overline{c.out}\,) \wedge \neg\,(\forall c{:}T.\ \overline{c.in}\,)$.

Example 4.2.14. The property that the graph, formed by components belonging to a set C and interactions among their ports, is connected can be expressed as follows:

$$\begin{aligned} \mathit{Connected}(C) \overset{def}{=} {} & \forall C'{:}T(C' \subsetneq C). \\ & \Big(\exists c'{:}T(c' \in C').\ \exists c{:}T(c \in C \setminus C').\ \sim\!(c.in\ c'.out) \sqcup \sim\!(c'.in\ c.out)\Big)\,. \end{aligned}$$

In particular, the formula requires that for any subset C' of C there exist an interaction that involves a component that belongs to C' and a component that belongs to $C \setminus C'$. This property cannot be expressed in the first-order logic and in the first-order logic with ordered components.

We show how the Ring, Linear and Square Grid architecture styles, previously shown in Examples 4.2.11, 4.2.12, 4.2.13, respectively, are specified with the *Connected* predicate.

Example 4.2.15. The component connection graph respects the Ring architecture style (Figure 4.12) if the following predicate is satisfied:

$$\begin{aligned}&\mathit{Connected}(U) \ \wedge\ \Sigma c{:}T.\ \exists c'{:}T(c \neq c').\ \sharp(c.\mathit{in},\ c'.\mathit{out})\\ &\qquad\qquad\qquad\qquad\wedge\ \Sigma c{:}T.\ \exists c'{:}T(c \neq c').\ \sharp(c.\mathit{out},\ c'.\mathit{in})\,,\end{aligned}$$

The constraint *Connected*(U) is used to ensure that all components form a single ring, rather than several disconnected ones. The second and third conjuncts require that each input port be connected to a unique output port.

Example 4.2.16. Linear architectures (Figure 4.13) involve serially connected components. The following formula requires that the components in C form a linear architecture:

$$\begin{aligned}\mathit{Linear}(C,\mathit{out},\mathit{in}) &\overset{def}{=} \mathit{Connected}(U)\ \wedge\ \exists c_1{:}T.\ \exists c_2{:}T(c_2 \neq c_1).\\ &\Big(\Sigma c{:}T(c \neq c_1).\ \exists c'{:}T(c' \not\in \{c, c_2\}).\ \sharp(c.\mathit{in},\ c'.\mathit{out})\\ &\qquad\wedge\ \Sigma c{:}T(c \neq c_2).\ \exists c'{:}T(c' \not\in \{c, c_1\}).\ \sharp(c.\mathit{out},\ c'.\mathit{in})\Big)\,.\end{aligned}$$

Example 4.2.17. The Square Grid architecture style, shown in Figure 4.14, can be expressed as follows:

$$\begin{aligned}&\Big(\forall c{:}T.\ \ \big(\overline{c.p}\ \sqcup \exists c'{:}T(c\neq c').\,\sharp(c.p, c'.r) + \overline{c.p}\,\big)\\ &\qquad\wedge\big(\overline{c.q}\ \sqcup \exists c'{:}T(c\neq c').\,\sharp(c.q, c'.s) + \overline{c.q}\,\big)\\ &\qquad\wedge\big(\overline{c.r}\ \sqcup \exists c'{:}T(c\neq c').\,\sharp(c.r, c'.p) + \overline{c.r}\,\big)\\ &\qquad\wedge\big(\overline{c.s}\ \sqcup \exists c'{:}T(c\neq c').\,\sharp(c.s, c'.q) + \overline{c.s}\,\big)\Big)\\ &\bigwedge\Big(\forall c{:}T.\,\exists C{:}T(c\in C).\,\mathit{MaxLinear}(C,p,r)\\ &\qquad\qquad\wedge\exists C'{:}T(C'\cap C=\{c\}\wedge|C'|=|C|).\,\mathit{MaxLinear}(C',q,s)\Big)\\ &\bigwedge\Big(\forall c_1 : T.\forall c_2 : T(c_1\neq c_2).\forall c_3{:}T(c_3\not\in\{c_1,c_2\}).\\ &\quad\sim(c_1.p\,c_2.r + c_1.q\,c_3.s) \Rightarrow \exists c_4{:}T\big(c_4\not\in\{c_1,c_2,c_3\}\big).\ \sim(c_2.q\,c_4.s + c_3.p\,c_4.r)\\ &\quad\wedge\sim(c_1.q\,c_2.s + c_1.r\,c_3.p) \Rightarrow \exists c_4{:}T\big(c_4\not\in\{c_1,c_2,c_3\}\big).\ \sim(c_2.r\,c_4.p + c_3.q\,c_4.s)\\ &\quad\wedge\sim(c_1.r\,c_2.p + c_1.s\,c_3.q) \Rightarrow \exists c_4{:}T\big(c_4\not\in\{c_1,c_2,c_3\}\big).\ \sim(c_2.s\,c_4.q + c_3.r\,c_4.p)\\ &\quad\wedge\sim(c_1.s\,c_2.q + c_1.p\,c_3.r) \Rightarrow \exists c_4{:}T\big(c_4\not\in\{c_1,c_2,c_3\}\big).\ \sim(c_2.p\,c_4.r + c_3.s\,c_4.q)\Big)\\ &\bigwedge\ \mathit{Connected}(T),\end{aligned}$$

where

$$MaxLinear(C, p_1, p_2) \overset{def}{=} Linear(C, p_1, p_2) \ \wedge \forall C' : T(C \subset C').\neg\ Linear(C', p_1, p_2).$$

The four big conjuncts represent, respectively, the following constraints:

1. Each port participates in at most one interaction.
2. Each component belongs in one row and one column of equal sizes. The conjunction with the first constraint ensures that, for any two components, the rows (columns) in which they belong either coincide or do not intersect.
3. If two components are connected to a third one and all three components do not belong in the same row or column then there exists a fourth component that is connected to the first two. The conjunction with the second constraint ensures that given two adjacent components that belong in the same row (column), all other components that belong in the columns (rows) of the first two components are pairwise connected.
4. Components form a single grid instead of several ones. Notice that it is not possible to distinguish a single grid from several small ones in the first-order logic and thus, this architecture style cannot be expressed in first-order logic.

4.3 Related work

An architecture style typically specifies a design vocabulary, constraints on how that vocabulary is used and semantic assumptions about that vocabulary [42]. Constraints may be about the allowed interactions between components, e.g. strong synchronization between components. Semantic assumptions concern the behaviour of the involved components, e.g. loss-less channel, server etc.

A plethora of approaches exist for characterizing architecture styles. For instance, patterns are very commonly used for this purpose. Patterns in [35, 51] incorporate explicit constructs for architecture modelling. Nonetheless, they lack formal semantics and they are not amenable to analysis.

Among the formal approaches for representing and analysing architecture descriptions, we distinguish two main categories:

- *Extensional approaches*, where one explicitly defines every object that is needed for the specification, i.e. the connections inducing interactions among the components (see the specification (4.13) of the Star pattern). All connections, other than the ones specified, are excluded. Most ADLs, for instance SOFA [58], Wright [4], XCD [81], adopt this approach.

- *Intentional approaches*, where one does not explicitly specify all the connections among the components, but these are derived from a set of logical constraints, formulating the intentions of the designer (see the specification (4.12) of the Star pattern). In this case specifications are conjunctions of logical formulas.

The proposed framework encompasses both approaches. It allows the description of individual interactions, e.g. by using interaction formulas. It also allows specification of configuration sets, e.g. by using formulas of the form $\sim f$.

A large body of literature, originating in [49, 64], studies the use of graph grammars and transformations [86] to define software architectures. Although this work focuses mainly on dynamic reconfiguration of architectures, e.g. [28, 62, 63], graph grammars can be used to extensionally define architecture styles: a style admits all the configurations that can be derived by its defining grammar. The main limitations, outlined already in [64], are the following: 1) the difficulty of understanding the architecture style defined by a grammar; 2) the fact that the restriction to context-free grammars precludes the specification of certain styles (e.g. trees with unbounded number of components or interactions, square grids); 3) the impossibility of combining several styles in a homogeneous manner. To some extent, the latter two are addressed, respectively, by considering synchronised hyperedge replacement [41], context-sensitive grammars [39, 106] and architecture views [84]. Our approach avoids these problems. Combining the extensional and intentional approaches allows intuitive specification of architecture styles. The higher-order extensions of PCL allow imposing global constraints necessary to specify styles that are not expressible by context-free graph grammars. Finally, the combination of several architecture styles is defined by the conjunction of the corresponding PCL formulas.

The proposed framework has similarities, but also significant differences, with the use of Alloy [55] and OCL [100] for intentional specification of architecture styles, respectively, in ACME and Darwin [43, 45] and in UML [24]. Our approach achieves a strong semantic integration between architectures and architecture styles. Moreover, configuration logic allows a fine characterization of the coordination structure by using n-ary connectivity predicates. On the contrary, the connectivity primitives in [43, 45] are binary predicates and cannot tightly characterize coordination structures involving multiparty interaction. To specify an n-ary interaction, these approaches require an additional entity connected by n binary links with the interacting ports. Since the behaviour of such entities is not part of the architecture style, it is impossible to distinguish, e.g., between an n-ary synchronisation and a sequence of n binary ones.

4.4 Summary

We presented configuration logics for the specification of architecture styles. Configuration logic formulas characterize interaction configurations between instances of typed components. Configuration logic is a powerset extension of interaction logic used to describe architectures.

It is integrated in a unified semantic framework which is equipped with a decision procedure for checking that a given architecture model meets given style requirements. Quantification over components and sets of components allows the genericity needed for architecture styles. We have illustrated through multiple examples that configuration logics are a powerful tool for modelling architecture styles.

5 Architecture diagrams

We propose a different avenue to architecture style specification based on *architecture diagrams*, which is a graphical language rooted on rigorous semantics. We focus on the specification of generic coordination mechanisms based on the concept of *connector*. Notice that in configuration logics, we use interactions to specify component coordination, whereas in architecture diagrams we use connectors. In this chapter, we use connectors that are not hierarchical and do not contain any triggers (Section 2.3.2). As a result, each connector defines exactly one interaction and thus, we can use the words connector and interaction interchangeably.

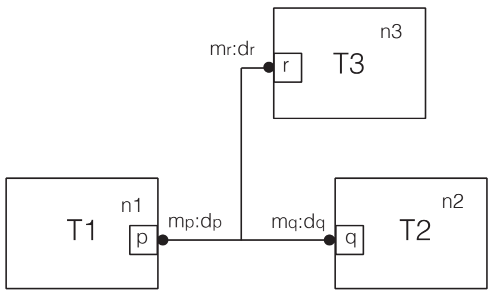


Figure 5.1 – An architecture diagram

An architecture diagram consists of a set of *component types*, a *cardinality function* and a set of *connector motifs*. Instances of a component type have the same interface and behaviour. The interface of a component type is characterised by a set of *port types*. The cardinality function associates each component type with its *cardinality*, i.e. number of instances. Figure 5.1 shows an architecture diagram consisting of three component types T_1, T_2 and T_3 with n_1, n_2 and n_3 instances and port types p, q and r, respectively. Instantiated components have *port instances* p_i, q_j, r_k for i, j, k belonging to the intervals $[1, n_1]$, $[1, n_2]$, $[1, n_3]$, respectively.

Connector motifs are non-empty sets of port types that must interact. Each port type p in the connector motif has two constraints represented as a pair $m : d$. Multiplicity m is the number of port instances p_i that are involved in the connectors. Degree d specifies the number of connectors in which each port instance is involved. A connector motif defines a set of configurations, where a configuration is a set of connectors. The architecture diagram in Figure 5.1 has a single connector motif involving port types p, q and r.

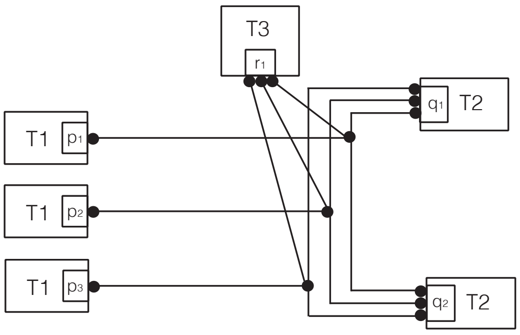


Figure 5.2 – Architecture conforming to the diagram in Figure 5.1

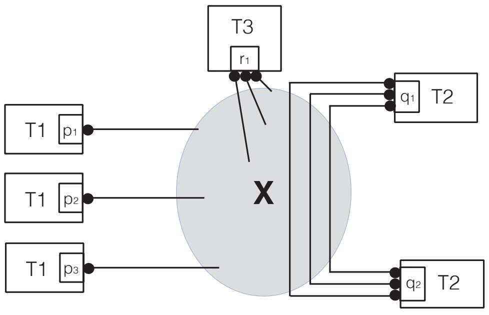


Figure 5.3 – Composition for the diagram in Figure 5.1

Figure 5.2 shows the unique architecture obtained from the diagram in Figure 5.1 by taking $n_1 = 3$, $m_p = 1$, $d_p = 1$; $n_2 = 2$, $m_q = 2$, $d_q = 3$; $n_3 = 1$, $m_r = 1$, $d_r = 3$. This is the result of composition of constraints for port types p, q and r as depicted in Figure 5.3. For port p, we have three instances and as both the multiplicity and the degree are equal to 1, each port p_i has a single connector lead. For port q, we have two instances and as the multiplicity is 2, we have connectors involving q_1 and q_2 and their total number is equal to 3 to meet the degree constraint. Finally, for port r, we have a single instance r_1 that has three connector leads to satisfy the degree constraint.

The semantics of an architecture diagram consisting of a set of connector motifs $\{\Gamma_i\}_{i\in[1..k]}$ is defined as follows. The meaning of each connector motif Γ_i is a set of configurations $\{\gamma_{i,j}\}_{j\in J_i}$. The architecture diagram specifies all the architectures characterised by configurations of connectors of the form: $\gamma_{1,j_1} \cup \cdots \cup \gamma_{n,j_n}$, where the indices $j_i \in J_i$. In other words, the configuration of an architecture conforming to the diagram is obtained by taking the union of all sub-configurations corresponding to each connector motif.

$$\begin{tikzcd}
 & \text{simple} \arrow[dl] \arrow[dr] & \\
\text{index} \arrow[dr] & & \text{interval} \arrow[dl] \\
 & \text{general} &
\end{tikzcd}$$

Figure 5.4 – The four classes of architecture diagrams

We propose four classes of architecture diagrams, namely *simple*, *interval*, *index* and *general* architecture diagrams. Figure 5.4 shows the expressiveness hierarchy of the four classes.

5.1 Simple architecture diagrams

We consider that a component interface is defined by its set of ports, which are used for interaction with other components. Thus, a component type T has a set of port types $T.P$.

5.1.1 Syntax and semantics

A *simple architecture diagram* $\langle \mathcal{T}, n, \mathcal{C} \rangle$ consists of:

- a set of *component types* $\mathcal{T} = \{T_1, \ldots, T_k\}$;
- an associated *cardinality* function $n : \mathcal{T} \to \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers (to simplify the notation, we will abbreviate $n(T_i)$ to n_i);
- a set of *connector motifs* $\mathcal{C} = \{\Gamma_1, \ldots, \Gamma_l\}$ of the form $\Gamma = (a, \{m_p : d_p\}_{p \in a})$, where $\emptyset \neq a \subset \bigcup_{i=1}^{k} T_i.P$ is a generic connector and $m_p, d_p \in \mathbb{N}$ (with $m_p > 0$) are the *multiplicity* and *degree* associated to port type $p \in a$.

Figure 5.5 shows the graphical representation of an architecture diagram with a single connector motif. It defines different architecture styles, for different values of the multiplicity, degree and cardinality parameters.

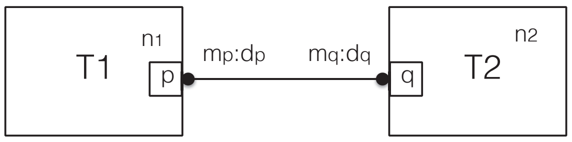


Figure 5.5 – A simple architecture diagram

An *architecture* is a pair $\langle \mathcal{B}, \gamma \rangle$, where $\mathcal{B}$ is an ordered set of components and γ is a *configuration*, i.e. a set of connectors among the ports of components in $\mathcal{B}$. We define a connector as a set of ports that must interact. For a component $B \in \mathcal{B}$ and a component type T, we say that B *is of type* T if the ports of B are in a bijective correspondence with the port types in T. Let $B_1, \ldots, B_n$ be all the components of type T in $\mathcal{B}$. For a port type $p \in T.P$, we denote the corresponding port instances by $p_1, \ldots, p_n$ and its associated cardinality by $n_p = n(T)$.

Semantics. An architecture $\langle \mathcal{B}, \gamma \rangle$ *conforms* to a diagram $\langle \mathcal{T}, n, \mathcal{C} \rangle$ if, for each $i \in [1, k]$, the number of components of type T_i in $\mathcal{B}$ is equal to n_i and γ can be partitioned into disjoint sets $\gamma_1, \ldots, \gamma_l$, such that, for each connector motif $\Gamma_i = (a, \{m_p : d_p\}_{p \in a}) \in \mathcal{C}$ and each $p \in a$,

1. there are exactly m_p instances of p in each connector in γ_i and
2. each instance of p is involved in exactly d_p connectors in γ_i.

We assume that, for any two connector motifs $\Gamma_i = (a, \{m_p^i : d_p^i\}_{p\in a})$ (for $i = 1, 2$) with the same set of port types a, there exists $p \in a$, such that $m_p^1 \neq m_p^2$. Two connector motifs with the same set of port types and multiplicities $\Gamma_i = (a, \{m_p : d_p^i\}_{p\in a})$ (for $i = 1, 2$) can be replaced by a single connector motif $\Gamma = (a, \{m_p : d_p^1 + d_p^2\}_{p\in a})$, which imposes a weaker constraint. We consider that the aforementioned assumption does not have significant impact on the expressiveness of the formalism. On the contrary, it greatly simplifies semantics and analysis. In particular, it ensures that for any configuration γ there exists at most one partition into disjoint sets $\gamma_1, \dots, \gamma_l$, which correspond to different connector motifs. In other words, a connector cannot correspond to two different connector motifs. This greatly simplifies function `VerifyMultiplicity` in Subsection 5.5. Furthermore, it also simplifies the consistency conditions presented in Subsection 5.1.2. Notice that this assumption allows connector motifs with the same set of port types but different multiplicities.

Multiplicity constrains the number of instances of the port type that must participate in a connector, whereas degree constrains the number of connectors attached to any instance of the port type. Consider the two diagrams shown in Figures 5.6 and 5.7. They have the same set of component types and cardinalities. Nevertheless, their multiplicities and degrees differ, resulting in different architectures. Architectures conforming to the two diagrams are also shown in Figures 5.6 and 5.7.

In Figure 5.6, the multiplicity of the port type p is 1 and the multiplicity of the port type q is 3, thus, any connector must involve one instance of p and three instances of q. Since there are only three instances of q, any connector must involve all of them. The degrees of both port types are 1, so each port is involved in exactly one connector. Thus, the diagram defines a single architecture with one quaternary connector.

In Figure 5.7, the multiplicities of both port types p and q is 1. Thus, all connectors are binary and involve one instance of p and one instance of q. The degree of p is 3, thus three connectors are attached to each instance of p. Thus, the architecture diagram defines a single architecture with three binary connectors.

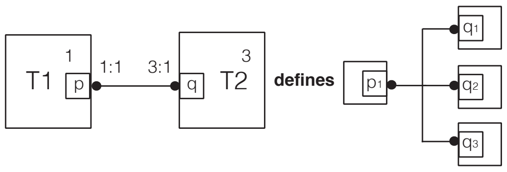


Figure 5.6 – Quaternary synchronisation

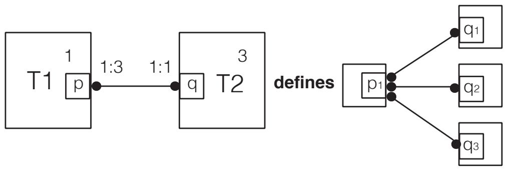


Figure 5.7 – Binary synchronisation

Notice that component instances of an architecture that conforms to a simple architecture diagram are not ordered and thus, the following property follows directly from the semantics.

Property 1. *Consider an architecture $A = \langle \mathcal{B}, \gamma \rangle$, containing two component instances B_1, B_2 of type T, that conforms to a simple architecture diagram $\mathcal{AD}$. Then, an architecture A' obtained by renaming B_1 to B_2 and B_2 to B_1 also conforms to $\mathcal{AD}$.*

5.1.2 Consistency conditions

Notice that there exist diagrams that do not define any architecture such as the diagram shown in Figure 5.8. Since the multiplicity is 1 for both port types p and q, a conforming architecture must include only binary connectors involving one instance of p and one instance of q. Additionally, since the degree of both p and q is 1, each port instance must be involved in exactly one connector. However, the cardinalities impose that there be three connectors attached to the instances of p, but only two connectors attached to the instances of q. Both requirements cannot be satisfied simultaneously and therefore, no architecture conforms to this diagram.

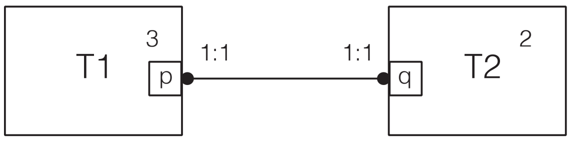


Figure 5.8 – An inconsistent diagram

Consider a connector motif $\Gamma = (a, \{m_p : d_p\}_{p \in a})$ in a diagram $\langle \mathcal{T}, n, \mathcal{C} \rangle$ and a port type $p \in a$, such that $p \in T.P$, for some $T \in \mathcal{T}$. We denote $s_p = n_p \cdot d_p / m_p$ the *matching factor of* p.

A *regular configuration of* p is a multiset of connectors, such that 1) each connector involves m_p instances of p and no other ports and 2) each instance of the port p is involved in exactly d_p connectors. Notice the difference between a configuration and a regular configuration of p: the former defines a set of connectors, while the latter defines a multiset of sub-connectors involving only instances of the port type p. Considering the diagram in Figure 5.1 and the architecture in Figure 5.2 the only regular configuration of r is the multiset $\{r_1,\ r_1,\ r_1\}$. The three copies of the singleton sub-connector r_1 are then fused with sub-connectors $p_i q_1 q_2$ $(i = 1, 2, 3)$, resulting in a configuration with three distinct connectors.

Lemma 5.1.1. *Each regular configuration of a port p has exactly s_p connectors.*

Proof. 1) We have s_p connectors and each connector is a set of m_p port instances. Thus, the sum of connectors' sizes is $s_p \cdot m_p$; 2) Connectors consist of ports and each port instance is involved in d_p connectors. The total number of ports in connectors is $n_p \cdot d_p$ where n_p is the number of port instances. Thus $s_p \cdot m_p = n_p \cdot d_p$ or $s_p = n_p \cdot d_p / m_p$. □

Notice that, for the diagram of Figure 5.8, we have $s_p = 3$, while $s_q = 2$. To form connectors, each sub-connector from a regular configuration of p must be fused with exactly one sub-connector from a regular configuration of q, and vice-versa. Since the sizes of such regular configurations are different by the above lemma, there is no architecture conforming to this diagram.

Proposition 5.1.2 provides the necessary and sufficient conditions for a simple architecture diagram to be consistent, i.e. to have at least one conforming architecture. The multiplicity of a port type must not exceed the number of component instances that contain this port. The matching factors of all ports participating in the same connector motif must be equal integers. Finally, since the number of distinct connectors of a connector motif is bounded and equal to $\prod_{q\in a}\binom{n_q}{m_q}$, there must be enough connectors to build a configuration. Notice that because of the assumption that we made in Subsection 5.1.1 this condition can be applied independently to each connector motif. Since, by the semantics of diagrams, connector motifs correspond to disjoint sets of connectors, these conditions are applied separately to each connector motif.

Proposition 5.1.2. *A simple architecture diagram has a conforming architecture iff, for each connector motif* $\Gamma = (a, \{m_p : d_p\}_{p\in a})$ *and each* $p \in a$*, we have*

1. $m_p \leq n_p$,
2. $\forall q \in a,\ s_p = s_q \in \mathbb{N}$,
3. $s_p \leq \prod_{q\in a}\binom{n_q}{m_q}$.

Proof. This proposition is a special case of Proposition 5.2.5. □

5.1.3 Synthesis of configurations

The synthesis procedure for each connector motif consists of the following two steps: 1) we find regular configurations for each port type; 2) we fuse these regular configurations generating global configurations specified by the connector motif.

Regular configurations of a port type We start with an example illustrating the steps of the synthesis procedure for a port p.

Example 5.1.3. Consider a port p with $n_p = 4$ and $m_p = 2$. There are 6 connectors of multiplicity 2: p_1p_2, p_1p_3, p_1p_4, p_2p_3, p_2p_4, p_3p_4. They correspond to the set of edges of a complete graph with vertices p_1, p_2, p_3, p_4. The regular configurations of p for $d_p = 1, 2, 3$, where each edge appears at most once (i.e. sets of connectors) are shown in Figure 5.9.

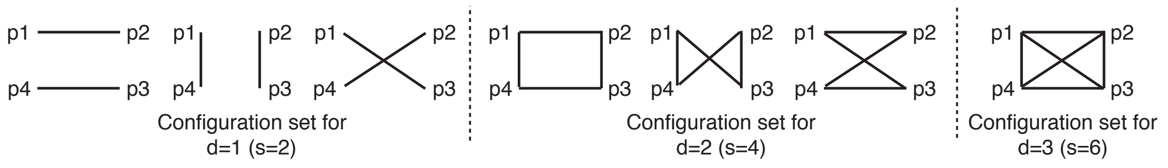

p1
p2
p3
p4
Configuration set for d=1 (s=2)
Configuration set for d=2 (s=4)
Configuration set for d=3 (s=6)

Figure 5.9 – Regular configurations of p with $n_p = 4$, $m_p = 2$

We provide below an equational characterisation of all the regular configurations (multisets) of a given port type p with given n_p, m_p, and d_p. For the n_p port instances, $p_1, \ldots, p_n$,

Table 5.1 – Vector representation of regular configurations

$d_p = 1$	$[100001], [010010], [001100].$
$d_p = 2$	$[110011], [101101], [011110],$ $[200002], [020020], [002200].$
$d_p = 3$	$[111111], [210012], [201102], [120021], [021120],$ $[012210], [102201], [300003], [030030], [003300].$

we have a set $\{a_i\}_{i\in[1,w]}$ of different connectors, where $w = \binom{n_p}{m_p}$, to which we associate a column vector of non-negative integer variables $X = [x_1, \ldots, x_w]^T$.

Consider the Example 5.1.3 and variables $x_1, \ldots, x_6$ representing the number of occurrences in a regular configuration of the connectors p_1p_2, p_1p_3, p_1p_4, p_2p_3, p_2p_4, p_3p_4, respectively. All the regular configurations (i.e. multisets of connectors), for $d_p = 1, 2, 3$, represented as vectors of the form $[x_1, \ldots, x_6]$ are listed in Table 5.1. Notice that vectors for $d_p > 1$ can be obtained as linear combinations of the vectors describing configuration sets for $d_p = 1$.

Then, for the port p we define an $n_p \times w$ incidence matrix $G = [g_{i,j}]_{n_p\times w}$ with $g_{i,j} = 1$ if $p_i \in a_j$ and $g_{i,j} = 0$ otherwise. The following equation holds: $GX = D$, where $D = [d_p, \ldots, d_p]$ (d_p repeated n_p times). Any non-negative integer solution of this equation defines a regular configuration of p. For Example 5.1.3, the equations are:

$$\begin{cases} x_1 + x_2 + x_3 = d, \\ x_1 + x_4 + x_5 = d, \\ x_2 + x_4 + x_6 = d, \\ x_3 + x_5 + x_6 = d\,, \end{cases} \quad \text{which is equivalent to} \quad \begin{cases} x_1 + x_2 + x_3 = d, \\ x_3 = x_4, \\ x_2 = x_5, \\ x_1 = x_6\,. \end{cases} \tag{5.1}$$

Notice that the vectors of Table 5.1 are solutions of (5.1).

Configurations of a connector motif Let $\Gamma = (a, \{m_p : d_p\}_{p\in a})$ be a connector motif such that all port types of $a = \{p^1, \ldots, p^v\}$ have the same integer matching factor s. For each $p^j \in a$, let $\gamma^j = \{a_i^j\}_{i\in[1,s]}$ be a regular configuration of p^j. For arbitrary permutations π_j of $[1, s]$, a set $\{a_i^1 \cup \bigcup_{j=2}^{v} a^j_{\pi_j(i)}\}_{i\in[1,s]}$ is a configuration specified by the connector motif.

In order to provide an equational characterisation of the connector motif, we consider, for each $j \in [1, v]$, a corresponding solution vector X^j of equations $G^jX^j = D^j$ characterising the regular configurations of p^j. We denote by w^j the dimension of the vector X^j.

In order to characterise the configurations of connectors conforming to Γ, we consider, for each configuration, the v-dimensional matrix $E = [e_{i_1,\ldots,i_v}]_{w^1\times\cdots\times w^v}$ of 0-1 variables, such that $e_{i_1,\dots,i_v} = 1$ if the connector $a^1_{i_1} \cup \dots \cup a^v_{i_v}$ belongs to the configuration and 0 otherwise. By definition, the sum of all elements in E is equal to s. Moreover, the following equations hold:

$$\begin{cases} x^1_i = \Sigma_{i_2,i_3\dots,i_v}\ e_{i,i_2,\dots,i_v}\,, & \text{for } i \in [1, w^1], \\ x^2_i = \Sigma_{i_1,i_3,\dots,i_v}\ e_{i_1,i,\dots,i_v}\,, & \text{for } i \in [1, w^2], \\ \quad\vdots & \\ x^v_i = \Sigma_{i_1,i_2,\dots,i_{v-1}}\ e_{i_1,\dots,i_{v-1},i}\,, & \text{for } i \in [1, w^v]. \end{cases} \tag{5.2}$$

For instance, for a fixed $i \in [1, w^1]$, all $e_{i,i_2,\dots,i_v}$ describe all connectors that contain a^1_i. The regular configuration γ^1 is characterised by X^1, enforcing that a^1_i belongs to x^1_i connectors. The system of linear equations (5.2), combined with the systems of linear equations $G^j X^j = D^j$, for $j \in [1, v]$, fully characterises the configurations of Γ. They can be used to synthesise architectures from architecture diagrams.

Example 5.1.4. Consider a diagram $(\{T_1, T_2\}, n, \{\Gamma\})$, where $T_1 = \{p\}$, $T_2 = \{q\}$, $n(T_1) = n(T_2) = 4$ and $\Gamma = (pq, \{(m_p : d_p, m_q : d_q)\})$ with $m_p = 2$, $m_q = 3$. The corresponding equations $G_p X = D_p$, $G_q Y = D_q$ can be rewritten as

$$\begin{cases} x_1 + x_2 + x_3 = d_p, \\ x_3 = x_4,\ x_2 = x_5,\ x_1 = x_6, \end{cases} \quad \text{and} \quad \begin{cases} 3y_1 = d_q, \\ y_1 = y_2 = y_3 = y_4. \end{cases} \tag{5.3}$$

Together with the constraints $x_i = \Sigma_j e_{i,j}$ and $y_j = \Sigma_i e_{i,j}$, for $E = [e_{i,j}]_{6\times 4}$, equations (5.3) completely characterise all the configurations conforming to Γ.

The same methodology can be used to synthesise configurations with additional constraints. To impose that some specific connectors must be included, whereas other specific connectors must be excluded from the configurations, the corresponding variables in the matrix E are given fixed values: 1 (resp. 0) if the connector must be included (resp. excluded) from the configurations. The rest of the synthesis procedure remains the same.

Example 5.1.5. Figure 5.10 shows the architecture diagram from Example 5.1.4, with $d_p = 2$ and $d_q = 3$. We want to synthesise the configurations of this diagram with the following additional constraints: connectors $p_1p_2q_1q_2q_3$ and $p_1p_3q_2q_3q_4$ must be included, whereas connector $p_2p_4q_1q_2q_4$ must be excluded from the synthesised configurations.

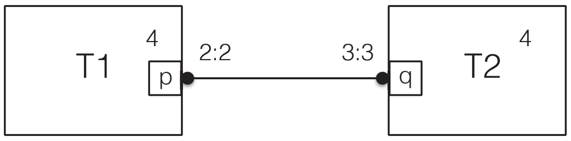


Figure 5.10 – Architecture diagram of Example 5.1.5

First, we compute the vectors X and Y that represent the regular configurations of port types p and q, respectively. Variables $x_1, \dots, x_6$ represent the number of occurrences in a configuration of the connectors p_1p_2, p_1p_3, p_1p_4, p_2p_3, p_2p_4, p_3p_4, respectively. Variables

$y_1, \dots, y_4$ represent the number of occurrences in a configuration of the connectors $q_1q_2q_3$, $q_1q_2q_4$, $q_1q_3q_4$, $q_2q_3q_4$, respectively. Vector X can take one of the following values for $d_p = 2$: [110011], [101101], [011110], [200002], [020020] or [002200] (Example 5.1.3). Regular configurations of q are characterised by the equations $3y_1 = d$ and $y_1 = y_2 = y_3 = y_4$ (Example 5.1.4). For $d = 3$ there is a single solution $Y = [1111]$.

We now consider the matrix E, where we fix $e_{1,1} = e_{2,4} = 1$ and $e_{5,2} = 0$ to impose the additional synthesis constraints:

$$E = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{array} \begin{array}{c} \begin{array}{cccc} y_1 & y_2 & y_3 & y_4 \end{array} \\ \begin{pmatrix} 1 & e_{1,2} & e_{1,3} & e_{1,4} \\ e_{2,1} & e_{2,2} & e_{2,3} & 1 \\ e_{3,1} & e_{3,2} & e_{3,3} & e_{3,4} \\ e_{4,1} & e_{4,2} & e_{4,3} & e_{4,4} \\ e_{5,1} & 0 & e_{5,3} & e_{5,4} \\ e_{6,1} & e_{6,2} & e_{6,3} & e_{6,4} \end{pmatrix} \end{array}$$

Since, for all $i \in [1, 6]$, we have $x_i = \Sigma_j\, e_{i,j}$, we observe that $x_1, x_2 \geq 1$. The only valuation of X that satisfies this constraint is [110011]; as mentioned above, the only possible valuation of Y is [1111]. The sum of rows 3 and 4 of E is 0, so all their elements must be 0s. The sum of rows 1 and 2 as well as the sum of columns 1 and 4 is 1. Since there exists already an element with value 1, all other elements in these rows and columns must be 0s. This gives us the following multidimensional matrix:

$$E = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{array} \begin{array}{c} \begin{array}{cccc} y_1 & y_2 & y_3 & y_4 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & e_{5,3} & 0 \\ 0 & e_{6,2} & e_{6,3} & 0 \end{pmatrix} \end{array}$$

The sums of the remaining rows and columns give us the correct values of the other three elements. The complete solution is the following:

$$E = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{array} \begin{array}{c} \begin{array}{cccc} y_1 & y_2 & y_3 & y_4 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array},$$

which corresponds to the configuration $\{p_1p_2q_1q_2q_3,\ p_1p_3q_2q_3q_4,\ p_2p_4q_1q_3q_4,\ p_3p_4q_1q_2q_4\}$.

5.1.4 Architecture style specification examples

Example 5.1.6. The Star architecture style consists of a single center component of type $T_1 = \{p\}$ and n_2 components of type $T_2 = \{q\}$. The central component is connected to every other component by a binary connector and there are no other connectors. The diagram in Figure 5.11 graphically describes the Star architecture style. The specification of the Star style with configuration logics was presented in Example 4.2.4.

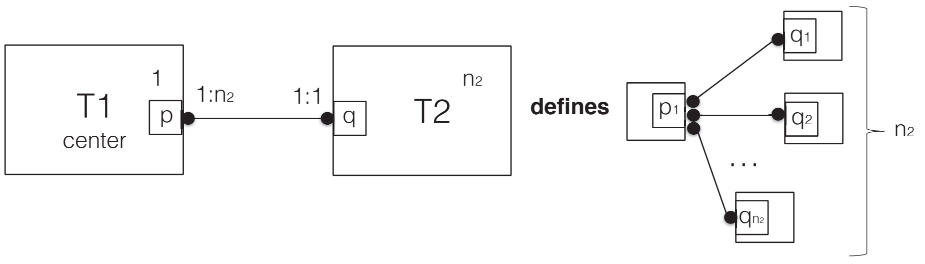


Figure 5.11 – Star architecture style with architecture diagrams

Example 5.1.7. We now consider the multi-star extension of the Star architecture style, with n center components of type T_1, each connected to d components of type T_2 by binary connectors. As in Example 5.1.6, there are no other connectors. The diagram of Figure 5.12 graphically describes this architecture style.

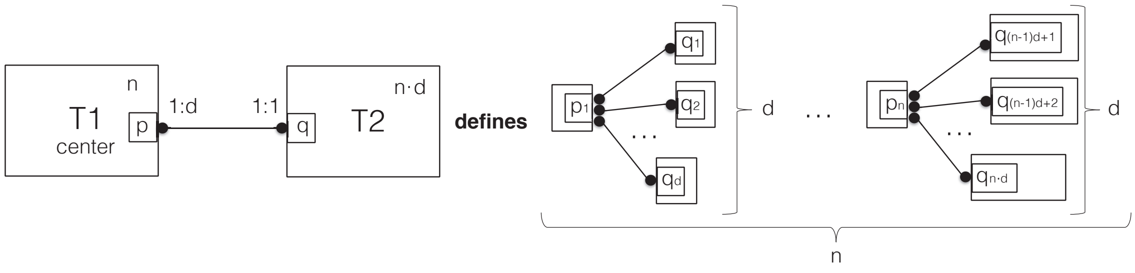


Figure 5.12 – Multi-star architecture style with architecture diagrams

5.2 Interval architecture diagrams

To further enhance the expressiveness of diagrams we introduce interval architecture diagrams where the cardinality, multiplicity and degree parameters can be intervals. With simple architecture diagrams, we cannot express properties such as *"component instances of type T are optional"*. For instance, let us consider the example in Figure 5.13[1], that shows four Master/Slave architectures involving two masters and two slaves. In this example, one of the masters might be optional, i.e. it might not interact with any slaves. As illustrated in

[1]This example was previously shown in Chapter 4, Figure 4.2.

Figure 5.13, in the first two architectures, each master interacts with one slave, however, in the last two architectures, a single master interacts with both slaves while the other master does not interact with any slaves. In other words, the degree of the port type m varies from 0 to 2 and cannot be represented by an integer.

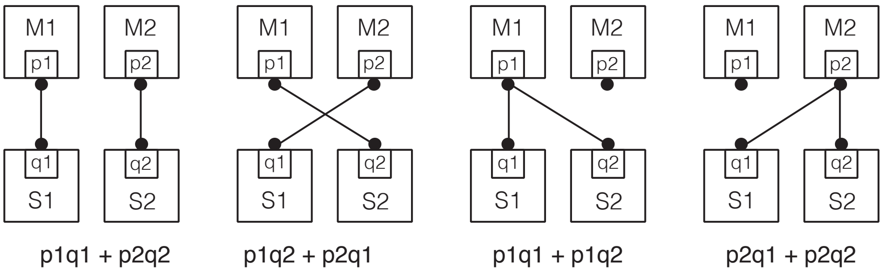


Figure 5.13 – Master/Slave architectures

5.2.1 Syntax and semantics

An *interval architecture diagram* $\langle \mathcal{T}, n, \mathcal{C}\rangle$ consists of:

- a set of *component types* $\mathcal{T} = \{T_1, \ldots, T_k\}$;
- a *cardinality* function $n : \mathcal{T} \to \mathbb{N}^2$, associating, to each $T_i \in \mathcal{T}$, an interval $n(T_i) = [n_i^l, n_i^u] \subset \mathbb{N}$ (thus, $n_i^l \leq n_i^u$);
- a set of *connector motifs* $\mathcal{C} = \{\Gamma_1, \ldots, \Gamma_l\}$ of the form $\Gamma = \Big(a, \{ty[m_p^l, m_p^u] : ty[d_p^l, d_p^u]\}_{p \in a}\Big)$, where $\emptyset \neq a \subset \bigcup_{i=1}^k T_i.P$ is a generic connector and $ty[m_p^l, m_p^u], ty[d_p^l, d_p^u]$, with $[m_p^l, m_p^u]$, $[d_p^l, d_p^u] \subset \mathbb{N}$ non-empty intervals and $ty \in \{mc, sc\}$ (mc means "multiple choice", whereas sc means "single choice"), are, respectively, *multiplicity* and *degree* constraints associated to $p \in a$.

Semantics. An architecture $\langle \mathcal{B}, \gamma\rangle$ *conforms* to an interval architecture diagram $\langle \mathcal{T}, n, \mathcal{C}\rangle$ if, for each $i \in [1, k]$, the number of components of type T_i in $\mathcal{B}$ lies in $[n_i^l, n_i^u]$ and γ can be partitioned into disjoint sets $\gamma_1, \ldots, \gamma_l$, such that for each connector motif $\Gamma_i = (a, \{ty[m_p^l, m_p^u] : ty[d_p^l, d_p^u]\}_{p \in a}) \in \mathcal{C}$ and each $p \in a$:

1. there are $m_p \in [m_p^l, m_p^u]$ instances of p in each connector in γ_i; in case of a single choice interval, the number of instances of p is equal in all connectors in γ_i;
2. each instance of p is involved in $d_p \in [d_p^l, d_p^u]$ connectors in γ_i; in case of a single choice interval, the number of connectors involving an instance of p is the same for all instances of p.

In other words, each port type p has an associated pair of intervals defining its multiplicity and degree. The interval attributes specify whether these constraints are uniformly applied

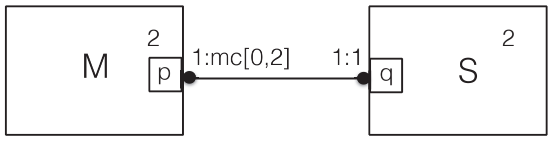


Figure 5.14 – Architecture diagram that defines the set of architectures in Figure 5.13

or not. We write $sc[x, y]$ (single choice) to mean that the same multiplicity or degree is applied to each port instance of p. We write $mc[x, y]$ (multiple choice) to mean that different multiplicities or degrees can be applied to different port instances of p, provided that they lie in the interval.

We assume that, for any two connector motifs $\Gamma_i = (a, \{ty[m_p^l, m_p^u]_i : ty[d_p^l, d_p^u]_i\}_{p \in a})$ (for $i = 1, 2$) with the same set of port types a, there exists $p \in a$ such that $[m_p^l, m_p^u]_1 \cap [m_p^l, m_p^u]_2 = \emptyset$. Similarly to simple architecture diagrams, without significant impact on the expressiveness of the formalism, this assumption greatly simplifies semantics and analysis. In particular, it ensures that for any configuration γ there exists at most one partition into disjoint sets $\gamma_1, \dots, \gamma_l$, which correspond to different connector motifs, which simplifies function `VerifyMultiplicity` in Section 5.5. Furthermore, it simplifies the consistency conditions in Proposition 5.2.5 that can now be applied independently to each connector motif.

Notice that component instances of an architecture that conforms to an interval architecture diagram are not ordered and thus, the following property follows directly from the semantics.

Property 2. *Consider an architecture A, containing two component instances B_1, B_2 of type T, that conforms to an interval architecture diagram $\mathcal{AD}$. Then, an architecture A' is obtained by renaming B_1 to B_2 and B_2 to B_1 also conforms to $\mathcal{AD}$.*

Example 5.2.1. The diagram in Figure 5.14 defines the set of architectures shown in Figure 5.13. Notice that the degree of port type p is the multiple choice interval $mc[0, 2]$, since one master component may be connected to two slaves, while the other master may not be connected to any slaves. For the sake of simplicity, we represent intervals $[x, x]$, $mc[x, x]$ and $sc[x, x]$ as x.

Proposition 5.2.2. *Interval architecture diagrams are strictly more expressive than simple architecture diagrams.*

Proof. Any cardinality n of a simple architecture diagram can be represented as $[n, n]$. Any multiplicity m (resp. degree d) can be represented as $sc[m, m]$ (resp. $sc[d, d]$). Thus, any simple architecture diagram can be represented as an interval architecture diagram proving that interval architecture diagrams are at least as expressive as simple architecture diagrams. To show that they are strictly more expressive than simple diagrams, let us consider the diagram in Figure 5.14 and its conforming architectures shown in Figure 5.13. Suppose that it is possible to express the set of architectures shown in Figure 5.13 with a simple architecture diagram. The first two architectures imply that the degree of port type p would be equal to 1, whereas the port instance p_1 of the third architecture implies that the degree of port type p would be equal to 2. Since the degree of a port type in simple diagrams cannot have multiple values, this results in a contradiction. $\square$

5.2.2 Consistency conditions

Similarly to simple architecture diagrams, there are interval diagrams that do not define any architectures. Proposition 5.2.5 provides the necessary and sufficient conditions for the consistency of interval architecture diagrams. A connector cannot contain more port instances than there exist in the system. Thus, the lower bound of multiplicity should not exceed the maximal number of instances of the associated component type. For all port types of a connector motif, there should exist a common matching factor that does not exceed the maximum number of different connectors between these ports. These conditions are a generalisation of Proposition 5.1.2. Auxiliary Lemmas 5.2.3 and 5.2.4 are necessary for proving Proposition 5.2.5.

Lemma 5.2.3. *Consider a set of port types P and a set of s connectors over these ports. Assume that connector $k \in [1, s]$ contains $m_{k,p}$ port instances of $p \in P$ and a port instance $p_j \in p$ is an element of exactly $d_{p,j}$ connectors. The following equality holds $\forall p \in P$, $\sum_{k=1}^{s} m_{k,p} = \sum_{p_j \in p} d_{p,j}$.*

Proof. Consider a port type $p \in P$. Consider a bipartite graph $G = (U, V, E)$ defined by two disjoint sets of vertices: set of vertices U where each vertex corresponds to one port instance, and a set of vertices V where each vertex corresponds to one connector. The graph G has an edge between vertices $u \in U$ and $v \in V$ if the port associated to u is an element of the connector associated to v. A vertex $v_k \in V$ associated with connector k is adjacent to exactly $m_{k,p}$ vertices associated with ports p of the connector k. Therefore, the number of edges between U and V is equal to $\sum_{k=1}^{s} m_{k,p}$. Furthermore, the degree of a vertex $u_j \in U$ is equal to $d_{p,j}$ because p_j is an element of $d_{p,j}$ connectors. Therefore, the number of edges between U and V must also be equal to $\sum_{p_j \in p} d_{p,j}$. Combining this with our previous observation, we obtain that $\sum_{k=1}^{s} m_{k,p} = \sum_{p_j \in p} d_{p,j}$. The same reasoning can be applied to any port type $p \in P$ giving the same equality. $\square$

Lemma 5.2.4. *Let P be a set of port types with two associated parameters: n_p representing a number of port instances $p \in P$ and $[d_p^l, d_p^u]$ for $d_p^l \in \mathbb{N}, d_p^u \in \mathbb{N},, d_p^l \leq d_p^u$ representing the desired degree interval. Consider a set of s distinct connectors A over P, such that for a port $p \in P$, a connector $a \in A$ contains $m_{a,p}$ instances of p, where $m_{a,p} \leq n_p$, and $n_p \cdot d_p^l \leq \sum_{a \in A} m_{a,p} \leq n_p \cdot d_p^u$. Then it is possible to construct a set of s distinct connectors A' such that a connector a contains $m_{a,p}$ instances of p with the degree of an instance p_j being equal to $d_{p,j} \in [d_p^l, d_p^u]$.*

Proof. Let $d_{p,i}$ be a degree of the port p_i in A, i.e. $d_{p,i} = |\{a \in A | p_i \in a\}|$. Let us define a function $f : 2^A \to \mathbb{N}$ such that $f(A) = \Sigma_{p \in P} \Sigma_{p_i \in p} \min_{d_p \in [d_p^l, d_p^u]} |d_{p,i} - d_p|$. Function $f(A)$

achieves its minimal value $f(A) = 0$ if and only if the degree $d_{p,i} \in [d^l_p, d^u_p]$ for all ports. That is, if $f(A) = 0$, we construct A' by the assignment $A' = A$.

Suppose now that we have A for which $f(A) \neq 0$. Since $f(A) \neq 0$, this means that there is at least one port p_i such that $d_{p,i} < d^l_p$ or $d_{p,i} > d^u_p$. Without loss of generality we can assume the first case (the other case is symmetric). From Lemma 5.2.3, we know that $\sum_{k=1}^{s} m_{k,p} = \sum_{p_j \in p} d_{p,j}$. Thus, for at least one port $p_j \in p$ holds $d_{p,j} > d^l_p$.

Now, observe that for the two ports p_i and p_j, $d_{p,j} > d_{p,i}$. This means that we can redefine at least one connector by replacing port p_i with port p_j without having duplicated connectors (otherwise, by the pigeonhole principle port p_j would already be an element of two identical connectors). Consider a new set of s connectors A_{new} obtained by applying the port replacement procedure. Its value function is at most $f(A_{new}) \leq f(A) - 1 < f(A)$. Since initial value $f(A)$ is bounded, the consecutive application of the port replacement procedure eventually leads to set A_{new} for which $f(A_{new}) = 0$. Therefore, it is possible to construct set A'. $\square$

To simplify the presentation of Proposition 5.2.5 we use the following notion of choice function. Let $\mathcal{I}_T$ and $\mathcal{I}$ be the sets of, respectively, typed intervals and intervals, as in the definition of interval diagrams above. A function $g : \mathcal{I}_T \to \mathcal{I}$ is a *choice function* if it satisfies the following constraints:

$$g(ty[x,y]) = \begin{cases} [x,y], & \text{if } ty = mc, \\ [z,z], & \text{for some } z \in [x,y], \text{ if } ty = sc. \end{cases}$$

Proposition 5.2.5. *An interval architecture diagram $\langle \mathcal{T}, n, \mathcal{C} \rangle$ is consistent iff, for each $T \in \mathcal{T}$, there exists a cardinality $n_i \in [n^l_i, n^u_i]$ and, for each connector motif $(a, \{M_p : D_p\}_{p\in a}) \in \mathcal{C}$ and each $p \in a$, there exist choice functions g^m_p, g^d_p, such that, for $[m^l_p, m^u_p] = g^m_p(M_p)$ and $[d^l_p, d^u_p] = g^d_p(D_p)$ hold:*

1. *$m^l_p \leq n_p$, for all $p \in a$, (where $n_p = n_i$ for $p \in T.P$),*
2. *$(S \cap U \cap \mathbb{N}) \neq \emptyset$, where*
   (a) *$S = \bigcap_{p\in a} s_p$ with* $$s_p = \begin{cases} [\frac{n_p \cdot d^l_p}{m^u_p}, \frac{n_p \cdot d^u_p}{m^l_p}], & \text{if } m^l_p > 0, \\ [\frac{n_p \cdot d^l_p}{m^u_p}, \infty), & \text{if } m^l_p = 0, \end{cases}$$
   (b) *$U = [1, \prod_{p\in a} \sum_{m\in[m^l_p, m^u_p]} \binom{n_p}{m}]$.*

Proof. Necessity $\to$: Consider an architecture conforming to the diagram. Consider values $n_T \in [n^l_T, n^u_T]$ for each $T \in \mathcal{T}$ equal to the number of components of the corresponding type in the architecture. Consider a connector motif $\Gamma = (a, \{M_p : D_p\}_{p\in a}) \in \mathcal{C}$ and consider functions g_p^m, g_p^d for each port type $p \in a$ consistent with the architecture, i.e. if the multiplicity (degree) of p in the sub-configuration corresponding to the connector motif Γ in the architecture is equal to v and the multiplicity (degree) interval has type sc then the corresponding g_p^m (g_p^d) returns $[v, v]$.

Condition 1 is trivially obtained - $n_p < m_p^l$ cannot occur as the multiplicity of ports cannot be greater than the number of component instances. In order to show condition 2 we apply Lemma 5.2.3, $\forall p \in a$, $\sum_{k=1}^{s} m_{k,p} = \sum_{p_j \in p} d_{p,j}$, where s is the number of connectors in the architecture corresponding to Γ. The lower bound on the left hand side is $s \cdot m_p^l$, while the upper bound on the right hand side is $n_p \cdot d_p^u$: these two bounds give us $s \leq \frac{n_p \cdot d_p^u}{m_p^l}$ (if $m_p^l = 0$, $s \to \infty$). By inspecting the upper bound of the left hand side and the lower bound of the right hand side, we obtain $s \geq \frac{n_p \cdot d_p^l}{m_p^u}$. Thus, $s \in S$. For the set U, notice that $\prod_{p \in a} \sum_{m \in [m_p^l, m_p^u]} \binom{n_p}{m}$ is equal to the number of different ways one could connect ports in a, so that port $p \in a$ has n_p instances and connector k contains $m_{k,p} \in [m_p^l, m_p^u]$ ports $p_i \in p$. Therefore, $s \in U$, otherwise, by the pigeonhole principle there would exist duplicated connectors. Thus, $s \in S$ and $s \in U$ and $s \in \mathbb{N}$ so their intersection is not empty. This reasoning can be applied to any connector motif, proving the necessity of the consistency conditions.

Sufficiency $\leftarrow$: We prove this part by construction. Consider values n_T and functions g for which all conditions are satisfied. Consider a set of behaviours, such that each type $T \in \mathcal{T}$ has n_T instances. In order to construct an architecture we need only a set of connectors. We construct sets for each connector motif independently, taking their union in the final step. Consider a connector motif $\Gamma = (a, \{M_p : D_p\}_{p \in a}) \in \mathcal{C}$. Suppose that there are no degree constraints. As in the first part of the proof, we know that condition 2 implies that the number of connectors is bounded by $\prod_{p \in a} \sum_{m \in [m_p^l, m_p^u]} \binom{n_p}{m}$, where $m_p^l \leq n_p$, which is satisfied by condition 1. Since $\prod_{p \in a} \sum_{m \in [m_p^l, m_p^u]} \binom{n_p}{m}$ is the number of different ways one could connect ports in a, so that port type $p \in a$ has n_p instances and connector k contains $m_{k,p} \in [m_p^l, m_p^u]$ port instances of port type p, it follows that it is always possible to select s distinct connectors (distinct by the set of port instances they contain), where a port instance $p_i \in p$ is allowed to have a degree $d_{p,i} \notin [d_p^l, d_p^u]$. Now, consider one such set A of s connectors. Since condition 2 is satisfied, we know that $n_p \cdot d_p^l \leq \sum_{k \in A} m_{k,p} \leq n_p \cdot d_p^u$ for all $p \in a$, so we can apply the result of Lemma 5.2.4. More precisely, we can iterate over ports in a (in arbitrary order), and balance the degrees of port instances $p_i \in p$, achieving the degree $d_{p,i} \in [d_p^l, d_p^u]$. Since by Lemma 5.2.4 each iteration preserves the distinctness of connections, once the entire iterative procedure finishes, we obtain a set of s distinct connectors for which each port instance has the degree that takes values in $[d_p^l, d_p^u]$, and this holds for all $p \in a$. Considering such sets for each connector motif and taking their union, we obtain an architecture that conforms to the diagram. $\square$

5.2.3 Synthesis of configurations

The equational characterisation in Section 5.1.3 can be generalised, using systems of inequalities with some additional variables, to interval architecture diagrams. Below, we show how to characterise the configurations induced by n instances of a port type p with the associated degree interval $ty[d_p^l, d_p^u]$.

For a given multiplicity m, let $X = [x_1, \ldots, x_w]^T$ be the column vector of integer variables, corresponding to the set $\{a_i\}_{i \in [1,w]}$ (with $w = \binom{n}{m}$) of connectors of multiplicity m, involving port instances $p_1, \ldots, p_n$. Let G be the incidence matrix $G = [g_{i,j}]_{n \times w}$ with $g_{i,j} = 1$ if $p_i \in a_j$ and $g_{i,j} = 0$ otherwise. The configurations induced by the n instances of p are characterised by the equation $GX = D$, where $D = [d_1, \ldots, d_n]^T$ and the additional (in)equalities:

$$\begin{aligned} &d_1 = \cdots = d_n = d \text{ and } d_p^l \leq d \leq d_p^u, && \text{for } ty = sc, \\ &d_p^l \leq d_1 \leq d_p^u, \ldots, d_p^l \leq d_n \leq d_p^u, && \text{for } ty = mc. \end{aligned} \tag{5.4}$$

Example 5.2.6. As in Example 5.1.3, consider a port type p and $n = 4$, $m = 2$. For the degree interval $sc[1, 3]$, the corresponding constraints are $1 \leq d \leq 3$, $x_1 + x_2 + x_3 = d$, $x_4 = x_3$, $x_5 = x_2$, $x_6 = x_1$.

For the degree interval $mc[1, 3]$ the corresponding constraints are $1 \leq d_i \leq 3$, for $i \in [1, 4]$, $x_1 + x_2 + x_3 = d_1$, $x_1 + x_4 + x_5 = d_2$, $x_2 + x_4 + x_6 = d_3$, $x_3 + x_5 + x_6 = d_4$. By solving this system we get:

$$\begin{cases} 0 \leq x_i \text{ for } i \in [1..6], \\ x_6 \leq 3, \\ x_5 \leq 3 - x_6, \\ x_4 \leq 3 - x_6, \\ x_4 \leq 3 - x_5, \\ 1 - x_5 - x_6 \leq x_3 \leq 3 - x_5 - x_6, \\ 1 - x_4 - x_6 \leq x_2 \leq 3 - x_4 - x_6, \\ x_2 \leq 3 - x_3, \\ 1 - x_4 - x_5 \leq x_1 \leq 3 - x_4 - x_5, \\ 1 - x_2 - x_3 \leq x_1 \leq 3 - x_2 - x_3. \end{cases}$$

Suppose that the multiplicity of p in the motif is given by an interval $ty[m_p^l, m_p^u]$. Contrary to the degree, multiplicity does not appear explicitly as a variable in the constraints. Instead, it influences the number and nature of elements in both the matrix G and vector X.

Therefore, for single choice (i.e. $ty = sc$), the configurations induced by n instances of p are characterised by the disjunction of the instantiations of the system of equalities combining $G_m X_m = D$ with (5.4), for $m \in [m_p^l, m_p^u]$.

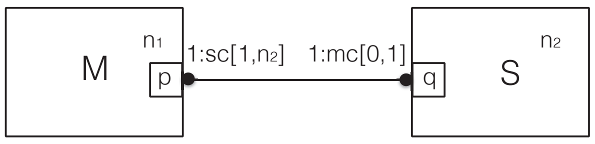


Figure 5.15 – Master/Slave architecture style with architecture diagrams

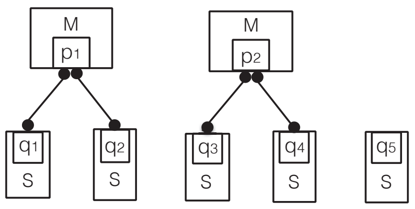


Figure 5.16 – A Master/Slave architecture

For multiple choice (i.e. $ty = mc$), all the configurations are characterised by the system combining (5.4) with

$$\sum_{m \in [m_p^l, m_p^u]} (G_m X_m) = D\,.$$

Notice that the above modifications to accommodate for interval-defined multiplicity are orthogonal to those in (5.4), accommodating for interval-defined degree. Similarly to the single-choice case for multiplicity, for interval-defined cardinality, the configurations are characterised by taking the disjunction of the characterisations for all values $n \in [n_p^l, n_p^u]$.

Based on the above characterisation for the configurations of one port type, global configurations can be characterised by systems of linear constraints in the same manner as for simple architecture diagrams.

5.2.4 Architecture style specification examples

Example 5.2.7. The diagram in Figure 5.15 describes a Master/Slave architecture style. We require that each slave interact with at most one master and that each master be connected to the same number of slaves.

Multiplicities of both port types p and q are equal to 1, allowing only binary connectors between a master and a slave. The single choice degree of port type p ensures that all port instances are connected to the same number of connectors which is a number in $[1, n_2]$. The multiple choice degree of port type q ensures that all port instances are connected to at most one master. A conforming architecture for $n_1 = 2$ and $n_2 = 5$ is shown in Figure 5.16.

Example 5.2.8. The diagram in Figure 5.17 describes the Repository architecture style involving a single instance of a component of type R and an arbitrary number of data-accessor components of type A. We require that any data-accessor component be connected to the repository. In Figure 5.17, we also show conforming architectures for 3 data-accessors. The

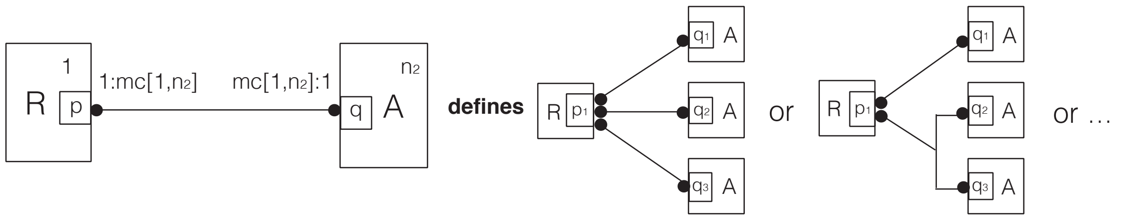


Figure 5.17 – Repository architecture style with architecture diagrams

Repository style was specified with configuration logics in Example 4.2.8.

Example 5.2.9. The Pipes and Filters architecture style [44] involves two types of components, P and F, each having two ports *in* and *out*. Each input (resp. output) of a filter is connected to an output (resp. input) of a single pipe. The output of any pipe can be connected to at most one filter. Figure 5.18 graphically describes the Pipes and Filters architecture style. The Pipes and Filters style was specified with configuration logics in Example 4.2.5.

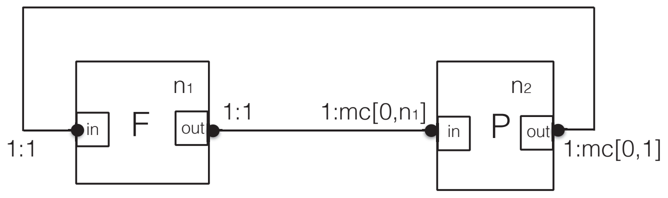


Figure 5.18 – Pipes and Filters architecture style with architecture diagrams

Example 5.2.10. The Map-Reduce architecture style [36] allows processing large datasets, such as those found in search engines and social networking sites. Figure 5.19 graphically describes the Map-Reduce architecture style. A conforming architecture for $n_1 = 3$ and $n_2 = 2$ is shown in Figure 5.20.

A large dataset is split into smaller datasets and stored in the global filesystem (GFS). The $Master$ is responsible for coordinating and distributing the smaller datasets from the GFS to each of the map worker components (MW). The port *in* of each MW is connected to the *Mcontrol* and *read* ports of the $Master$ and the GFS, respectively. Each MW processes the datasets and writes the result to its dedicated local filesystem (LFS) through a binary connector between their *out* and *write* ports. The connector is binary since no MW is allowed to read the output of another MW. Each reduce worker (*RW*) reads the results from multiple LFS as instructed by the $Master$ component. To this end, the *in* port of each *RW* is connected to the *Rcontrol* and *read* ports of the $Master$ and some LFS, respectively. Then, each *RW* combines the results to produce a final result written back to the GFS through a binary connector between their *out* and *write* ports. No *RW* is allowed to read the output of another *RW*.

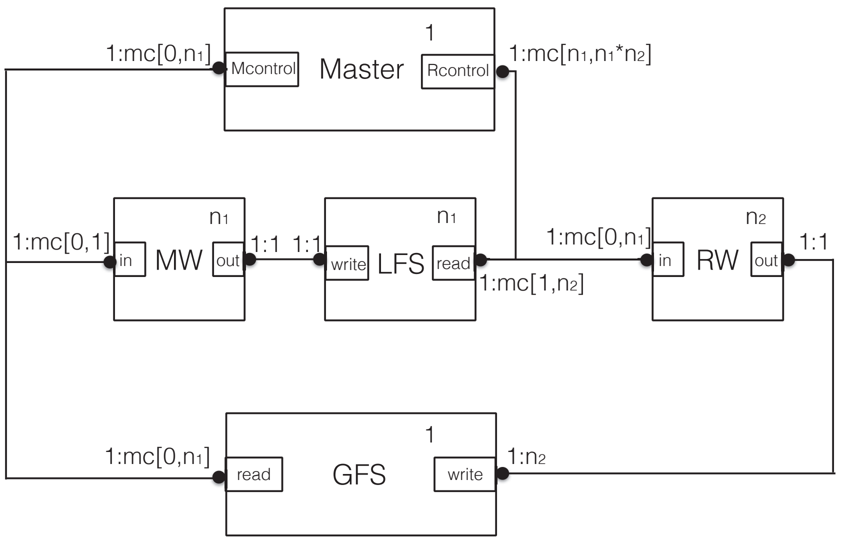


Figure 5.19 – Map-Reduce architecture style with architecture diagrams

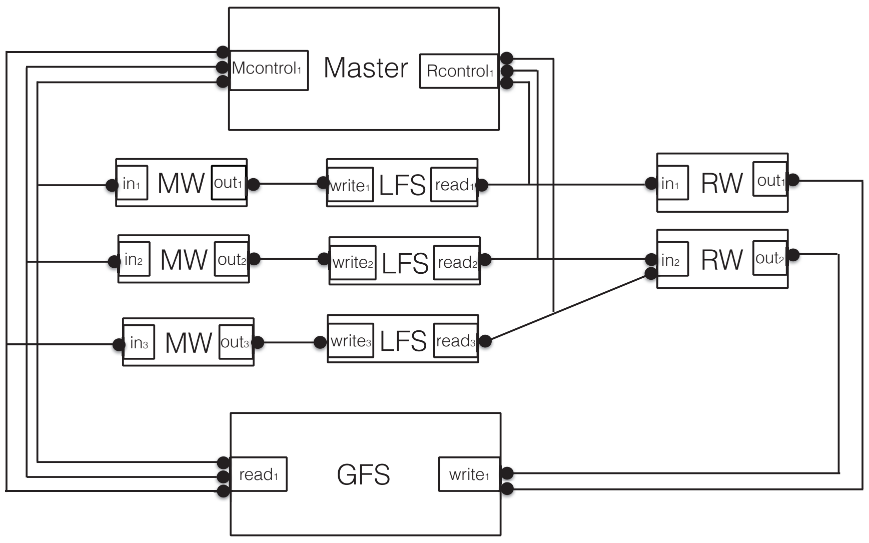


Figure 5.20 – A Map-Reduce architecture

5.3 Index architecture diagrams

We extend simple architecture diagrams with index variables on ports. Arithmetic predicates over the set of index variables are associated with connector motifs and restrict the involvement of port instances in the same connector.

5.3.1 Syntax and semantics

An *index architecture diagram* $\langle \mathcal{T}, n, \mathcal{C} \rangle$ consists of:

- a set of *component types* $\mathcal{T} = \{T_1, \ldots, T_k\}$;
- an associated *cardinality* function $n : \mathcal{T} \to \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers (again, we abbreviate $n(T_i)$ to n_i);
- a set of *connector motifs* $\mathcal{C} = \{\Gamma_1, \ldots, \Gamma_l\}$ of the form $\Gamma = (a, \{m_p : d_p\}_{p \in a}, \{I_p\}_{p \in a}, \Phi(\mathcal{I}))$, where
  - $\emptyset \neq a \subset \bigcup_{i=1}^{k} T_i.P$ is a generic connector,
  - $m_p, d_p \in \mathbb{N}$ (with $m_p > 0$) are the *multiplicity* and *degree* associated to port type $p \in a$,
  - I_p (with $|I_p| = m_p$) is the set of index variables associated to p,
  - $\Phi(\mathcal{I})$ is a predicate based on arithmetic operations over the set of index variables $\mathcal{I} = \bigcup_{p \in a} I_p$.

To maintain consistency with the notations used in configuration logics and simple architecture diagrams, we use the notation $p[I]$ to associate the set of index variables I with a port type p in the graphical representation of index architecture diagrams.

Semantics. An architecture $\langle B, \gamma \rangle$ *conforms* to a diagram $\langle \mathcal{T}, n, \mathcal{C} \rangle$ if, for each $i \in [1, k]$, the number of components of type T_i in B is equal to n_i and γ can be partitioned into disjoint sets $\gamma_1, \ldots, \gamma_l$, such that, for each connector motif $\Gamma_i = (a, \{m_p : d_p\}_{p \in a}, \{I_p\}_{p \in a}, \Phi(\mathcal{I})) \in \mathcal{C}$ and each $p \in a$,

1. there are exactly m_p instances of p in each connector in γ_i;
2. each instance of p is involved in exactly d_p connectors in γ_i.
3. each connector imposes a valuation of index variables $\mathcal{I}$ that satisfies the associated predicate $\Phi(\mathcal{I})$.

Similarly to simple diagrams we assume that, for any two connector motifs $\Gamma_i = (a, \{m_p^i : d_p^i\}_{p \in a}, \Phi(\mathcal{I}))$ (for $i = 1, 2$) with the same set of port types a, there exists $p \in a$, such that $m_p^1 \neq m_p^2$.

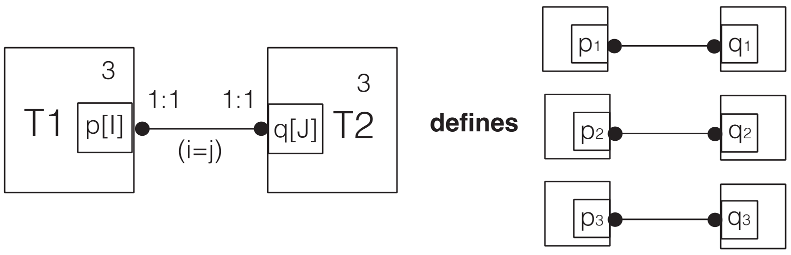


Figure 5.21 – An index architecture diagram

Consider the diagram shown in Figure 5.21. Notice that, since the multiplicities of both port types in the connector motif are equal to 1, the associated sets of index variables can be unambiguously assumed to be $I = \{i\}$ and $J = \{j\}$. The predicate $(i = j)$ imposes the restriction that components can be connected only if their indices are equal. For instance, the configuration $\{p_1q_2, p_2q_3, p_3q_1\}$ does not conform to the diagram shown in Figure 5.21.

Since simple architecture diagrams do not have mechanisms to restrict possible configurations based on the order of components, we obtain the following result.

Proposition 5.3.1. *Index architecture diagrams are strictly more expressive than simple architecture diagrams.*

Proof. Any simple architecture diagram can be represented as an index architecture diagram if the associated predicate $\Phi(\mathcal{I})$ of each connector is equal to true. Thus, index architecture diagrams are at least as expressive as simple architecture diagrams. To show that they are strictly more expressive than simple diagrams, let us consider the diagram in Figure 5.21. Suppose there exists a simple diagram, whereof the only conforming architecture is that in the right-hand side of Figure 5.21. By Property 1, it has to be that the set of conforming architectures of this diagram also contains another architecture that differs from the one in Figure 5.21 by having connectors between $p_1 - q_2$ and $p_2 - q_1$ instead of connectors between $p_1 - q_1$ and $p_2 - q_2$. Thus, the cardinality of the set of conforming architectures of this diagram is strictly greater than 1, which proves that no simple architecture diagram can define exactly the architecture shown in Figure 5.21. $\square$

5.3.2 Architecture style specification examples

For the sake of simplicity, in the following examples we omit the set of indices of port types if there is no associated predicate.

Example 5.3.2. The Request/Response pattern is graphically specified in Figure 5.22. There are two types of components: a client Cl and a service S. A coordinator of type C enforces the properties: 1) only one client can be connected at a time to a service; and 2) a client has to connect to the service before sending a request. A unique coordinator is needed per service and therefore, the number of coordinators must match the numbers of

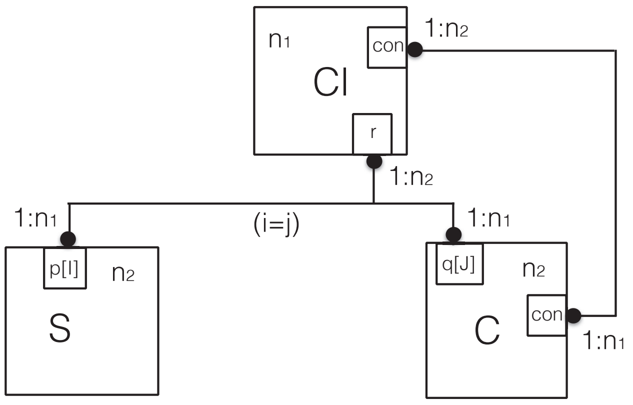


Figure 5.22 – Request/Response style with architecture diagrams

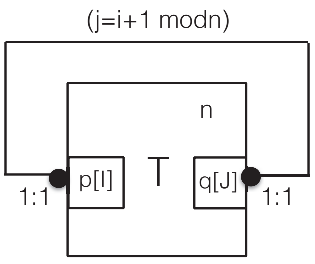


Figure 5.23 – Ring style with architecture diagrams

services. There can be arbitrarily many clients. The Request/Reply pattern is illustrated in Figure 4.11. The property *"a unique coordinator is needed per service"* is enforced by choosing pairs of services and coordinators with equal indices. The equivalent specification with configuration logics was presented in Example 4.2.7.

Example 5.3.3. The Ring architecture style is graphically specified in Figure 5.23. The predicate $(j = i + 1 \mod n)$ ensures that all components are connected in a single ring. Equivalent specifications with configuration logics were presented in Example 4.2.15 and Example 4.2.11.

5.4 General architecture diagrams

General architecture diagrams combine both extensions of simple architecture diagrams, i.e. indices and intervals and additionally, introduce arithmetic predicates on multiplicities and degrees. The former are associated with connector motifs, while the latter are associated with component types.

5.4.1 Syntax and Semantics

A *general architecture diagram* $\langle \mathcal{T}, n, \mathcal{C}, \mathcal{D} \rangle$ consists of:

- a set of *component types* $\mathcal{T} = \{T_1, \dots, T_k\}$;
- a *cardinality* function $n : \mathcal{T} \to \mathbb{N}^2$, associating, to each $T_i \in \mathcal{T}$, an interval $n(T_i) = [n_i^l, n_i^u] \subset \mathbb{N}$ (thus, $n_i^l \leq n_i^u$);
- a set of *connector motifs* $\mathcal{C} = \{\Gamma_1, \dots, \Gamma_l\}$ of the form

$$\Gamma = \Big(a, \{ty[m_p^l, m_p^u] : ty[d_p^l, d_p^u]\}_{p \in a}, \{I_p\}_{p \in a}, \Phi(\mathcal{I}), \{m_p\}_{p \in a}, \{d_p\}_{p \in a}, \mathcal{M}\Big),$$

  where

  – $\emptyset \neq a \subset \bigcup_{i=1}^k T_i.P$ is a generic connector,

  – $ty[m_p^l, m_p^u], ty[d_p^l, d_p^u]$, with $[m_p^l, m_p^u], [d_p^l, d_p^u] \subset \mathbb{N}$ non-empty intervals and $ty \in \{mc, sc\}$ (mc means "multiple choice", whereas sc means "single choice"), are, respectively, *multiplicity* and *degree* constraints associated to $p \in a$,

  – I_p (with $|I_p| = m_p$) is the set of index variables associated to p,

  – $\Phi(\mathcal{I})$ is a predicate based on arithmetic operations over the set of index variables $\mathcal{I} = \bigcup_{p \in a} I_p$.

  – $\{m_p\}_{p \in a}$ and $\{d_p\}_{p \in a}$ are the sets of, respectively, multiplicity and degree variables associated to p;

  – $\mathcal{M}$ is a predicate over $\{m_p\}_{p \in a}$, which constrains the valuation of multiplicities;
- a set of constraints $\mathcal{D} = \{D_1, \dots, D_k\}$, where, for each $i \in [1, k]$, D_i is a predicate over the set of degree variables $\{\Gamma.d_p \,|\, \Gamma \in \mathcal{C}, p \in \Gamma.a \cap T_i.P\}$ associated to the port types in $T_i.P$.

Semantics. An architecture $\langle B, \gamma \rangle$ *conforms* to a general architecture diagram $\langle \mathcal{T}, n, \mathcal{C}, \mathcal{D} \rangle$ if, for each $i \in [1, k]$, the number of components of type T_i in $\mathcal{B}$ lies in $[n_i^l, n_i^u]$, the degrees of ports of each component of type T_i satisfy the predicate D_i and γ can be partitioned into disjoint sets $\gamma_1, \dots, \gamma_l$, such that for each connector motif $\Gamma_i = \big(a, \{ty[m_p^l, m_p^u] : ty[d_p^l, d_p^u]\}_{p \in a}, \{m_p\}_{p \in a}, \{d_p\}_{p \in a}, \mathcal{M}\big) \in \mathcal{C}$ and each $p \in a$:

1. there are $m_p \in [m_p^l, m_p^u]$ instances of p in each connector in γ_i; in case of a single choice interval the number of instances of p is equal in all connectors in γ_i;
2. each instance of p is involved in $d_p \in [d_p^l, d_p^u]$ connectors in γ_i; in case of a single choice interval, the number of connectors involving an instance of p is the same for all instances of p;
3. each connector imposes a valuation of index variables $\mathcal{I}$ that satisfies the associated predicate $\Phi(\mathcal{I})$;
4. in each connector, the involved port multiplicities satisfy the corresponding $\mathcal{M}$ predicate;
5. in each component of type T_i, the ports degrees satisfy the predicate D_i.

Lemma 5.4.1. *The expressiveness of index and interval architecture diagrams is incomparable.*

Proof. To show that the expressiveness of index and interval architecture diagrams is incomparable, let us consider the diagrams of Figures 5.21 and 5.13. Suppose there exists an interval diagram, whereof the only conforming architecture is that in the right-hand side of Figure 5.21. By Property 2, it has to be that the set of conforming architectures of this diagram also contains another architecture that differs from the one in Figure 5.21 by having connectors between $p_1 - q_2$ and $p_2 - q_1$ instead of connectors between $p_1 - q_1$ and $p_2 - q_2$. Thus, the cardinality of the set of conforming architectures of this diagram is strictly greater than 1, which proves that no interval architecture diagram can define exactly the architecture shown in Figure 5.21. Now, suppose that it is possible to express the set of architectures shown in Figure 5.13 with an index architecture diagram. The first two architectures imply that the degree of port type p would be equal to 1, whereas the port instance p_1 of the third architecture implies that the degree of port type p would be equal to 2. Since the degree of a port type in index diagrams has a single value this results in a contradiction. $\square$

Proposition 5.4.2. *General architecture diagrams are strictly more expressive than simple, interval and index architecture diagrams.*

Proof. Any interval architecture diagram can be represented by a general diagram with the associated predicate $\Phi(\mathcal{I})$ of each connector equal to true and predicates $\mathcal{D}$, $\mathcal{M}$ equal to true. Thus, general architecture diagrams are at least as expressive as interval architecture diagrams. Any cardinality n of an index architecture diagram can be represented as $[n, n]$. Any multiplicity m (resp. degree d) can be represented as $sc[m, m]$ (resp. $sc[d, d]$). Thus, any index architecture diagram can be represented as a general architecture diagram with predicates $\mathcal{D}$, $\mathcal{M}$ equal to true. Thus, general architecture diagrams are at least as expressive as index architecture diagrams. By Lemma 5.4.1, the expressiveness of index and interval diagrams is incomparable, which further implies that general diagrams are strictly more expressive than index and interval diagrams. $\square$

5.4.2 Architecture style specification examples

Example 5.4.3. The Peer-to-Peer architecture style [44] involves one component type P with port types r (request) and p (provide). Figure 5.24 graphically describes the style. Peers request and provide services through binary connectors between their r and p ports. Since the multiplicities of both ports are 1, the corresponding sets of index variables are $I = \{i\}$ and $J = \{j\}$. The predicate $(i \neq j)$ is satisfied if there is no connector between two ports that belong to the same peer.

Example 5.4.4. The Square Grid architecture style involves one component type M with four port types p, q, r and s. Figure 5.25 graphically describes the style. There are n^2

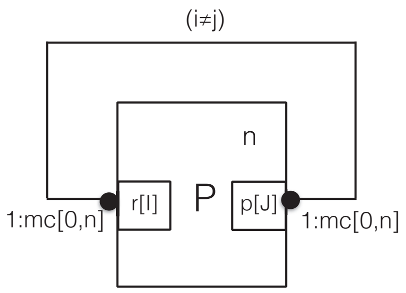


Figure 5.24 – Peer-to-Peer style with architecture diagrams

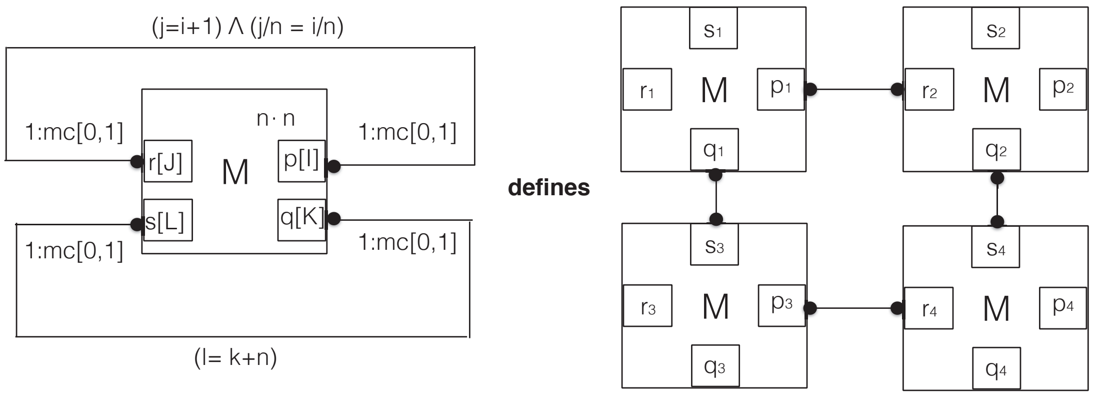


Figure 5.25 – Square Grid style with architecture diagrams

components that form a square. Multiplicities of all port types are equal to 1, allowing only binary connectors and, as in the previous example, uniquely defining the index variables. The multiple choice degrees require that all ports be connected to at most one connector. Predicates $(j = i+1) \wedge (j/n = i/n)$ and $l = k+n$ ensure that components are connected only to their neighbours in the grid. A conforming architecture for a 2×2 grid of components is also shown in Figure 5.25. Equivalent specifications with configuration logics were presented in Example 4.2.17 and Example 4.2.13.

Example 5.4.5. The Triple Modular Redundancy (TMR) [26, 97, 27] architecture style is graphically described in Figure 5.27. A TMR architecture consists of three modules M, three input modules IM, three output modules OM and some voters V. The number of voters spans from 1 to 3. Notice the predicate $d_{in_1} + d_{in_2} = 1$ on the degrees of connector motifs connected to the port type in of component type OM. This predicate ensures that the number of connectors connected to each OM component instance is exactly one. In other words, the output must be taken either directly from a module or from a voter, but not from both sources simultaneously. A representative set of architectures described by the diagram in Figure 5.27 is shown in Figure 5.26.

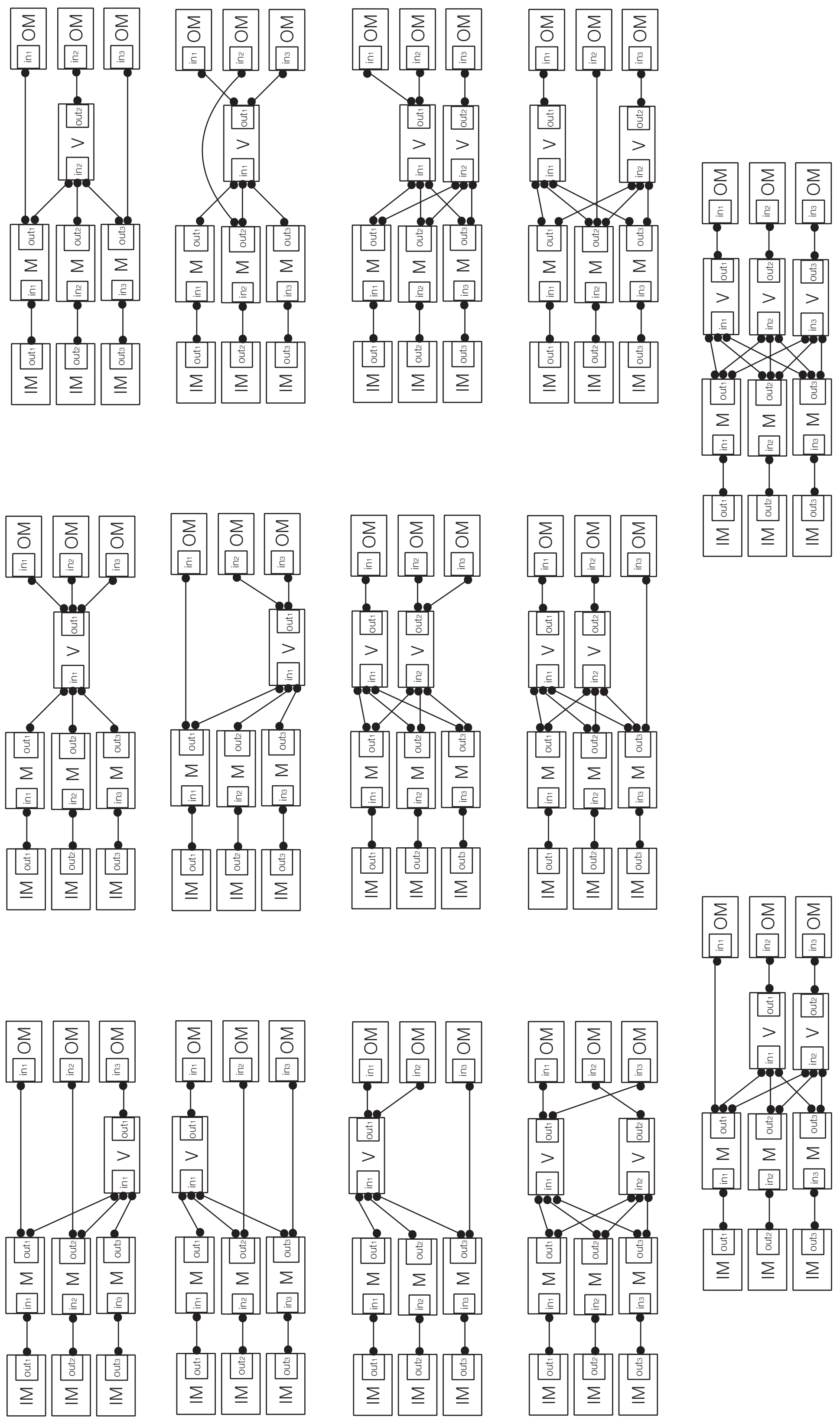


Figure 5.26 – Triple Modular Redundancy architectures

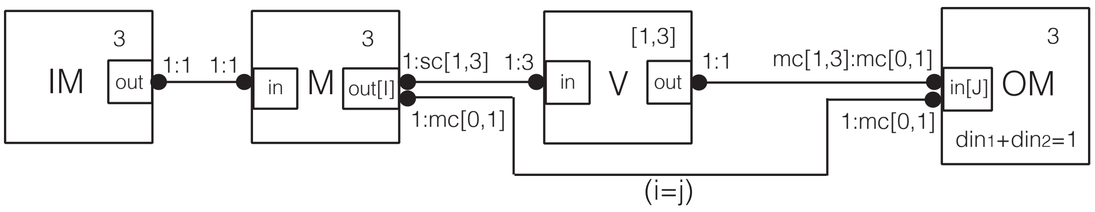


Figure 5.27 – Triple Modular Redundancy style with architecture diagrams

5.5 Checking conformance of diagrams

Algorithm 2 with polynomial-time complexity checks whether an architecture $\langle \mathcal{B}, \gamma \rangle$ conforms to a general architecture diagram $\langle \mathcal{T}, n, \mathcal{C}, \mathcal{D} \rangle$. In particular, Algorithm 2 checks the validity of the following five statements: 1) the number of components of each type T lies in the corresponding cardinality interval; 2) there exists a partition of γ into $\gamma_1, \ldots, \gamma_l$ such that each γ_i corresponds to a different connector-motif $\Gamma_i \in \mathcal{C}$ of the diagram; 3) for each connector motif Γ_i and its corresponding γ_i, the associated index predicates Φ are satisfied; 4) for each connector motif Γ_i and its corresponding γ_i, the number of times each port instance participates in γ_i satisfies the degree constraints and 5) for each component type the associated degree predicate $\mathcal{D}$ are satisfied. The five statements correspond to functions VerifyCardinality, VerifyMultiplicity, VerifyIndex, VerifyDegree and VerifyDegreePredicates, respectively. If all statements are valid the algorithm returns *true*, i.e. the architecture conforms to the diagram.

Function VerifyCardinality takes as input the architecture diagram $\langle \mathcal{T}, n, \mathcal{C} \rangle$ and the set of components $\mathcal{B}$ of the architecture $\langle \mathcal{B}, \gamma \rangle$. It counts the number of components for each component type in $\mathcal{B}$ and it returns *true* if for each component type $\mathcal{T}$ of the diagram its cardinality matches the corresponding number of components in $\mathcal{B}$. Otherwise it returns *false* and algorithm 2 terminates.

Function VerifyMultiplicity takes as input the configuration γ of the architecture $\langle \mathcal{B}, \gamma \rangle$ and the set of connector motifs $\mathcal{C}$ of the architecture diagram $\langle \mathcal{B}, \gamma \rangle$. The function checks whether there exists a partition of γ such that each sub-configuration γ_i of γ corresponds to a distinct connector motif $\mathcal{C}_i$ of $\mathcal{C}$, i.e. each connector k in γ_i conforms to the multiplicity constraints of $\mathcal{C}_i$, for which the associated multiplicity predicates $\mathcal{M}$ evaluate to true. If such a partition exists the function returns it. Otherwise, it returns $\emptyset$ and algorithm 2 terminates.

Function VerifyIndex takes as input a connector motif Γ_i of $\mathcal{C}$ and its corresponding sub-configuration γ_i assigned by VerifyMultiplicity. For each connector in γ_i, it iterates through its port instances and checks whether the associated arithmetic predicates on port indices Φ evaluate to true. If the check fails, algorithm 2 terminates.

Function VerifyDegree takes as input a connector motif Γ_i of $\mathcal{C}$ and its corresponding sub-configuration γ_i assigned by VerifyMultiplicity. For each port instance in the sub-configuration, it checks whether the number of times the port participates in different connectors lies in the corresponding degree interval of the connector motif. If the check fails, algorithm 2 terminates.

Function VerifyDegreePredicates takes as input the architecture diagram $\langle \mathcal{T}, n, \mathcal{C} \rangle$, the set of components $\mathcal{B}$ of the architecture and the partition S_γ of γ returned by VerifyMultiplicity. For each component instance T it checks whether the associated degree predicates $\mathcal{D}$ evaluate to true. To do that, for each connector motif and its associated degree predicate, it iterates through all corresponding connectors in γ and computes the degree of each port instance of T. If the check fails, algorithm 2 terminates.

Algorithm 2 uses a number of auxiliary functions. Function `generic(p)` takes a port instance and returns the corresponding port type. Function `typeof(B)` returns the component type of component `B`. Operation `map[key]++` increases the `value` associated with the `key` by one if the `key` is in the `map`, otherwise it adds a new `key` with `value` 1.

Algorithm 2: VerifyArchitecture

```
Data: Architecture $\langle \mathcal{B}, \gamma \rangle$, diagram $\langle \mathcal{T}, n, \mathcal{C}, \mathcal{D} \rangle$
Result: Returns true if the architecture satisfies the diagram $\langle \mathcal{T}, n, \mathcal{C}, \mathcal{D} \rangle$. Otherwise, it returns
        false.
begin
    if not VerifyCardinality($\mathcal{B}$, $\langle \mathcal{T}, n, \mathcal{C}, \mathcal{D} \rangle$) then
        return false;
    /* Splits connectors between connector motifs according to multiplicities constraints.*/
    $\mathcal{S}_\gamma \longleftarrow$ VerifyMultiplicity($\gamma$, $\mathcal{C}$);
    if $\mathcal{S}_\gamma = \emptyset$ then
        return false;
    /* Verifies degree and index constraints for all port types of all connector motifs. */
    for $\Gamma \in \mathcal{C}$ do
        if VerifyIndex($\mathcal{S}_\gamma[\Gamma], \Gamma$) $\neq$ true then
            return false;
        if VerifyDegree($\mathcal{S}_\gamma[\Gamma], \Gamma$) $\neq$ true then
            return false;
    /* Verifies degree predicates $\mathcal{D}$. */
    if VerifyDegreePredicates($\mathcal{B}$, $S_\gamma$, $\langle \mathcal{T}, n, \mathcal{C}, \mathcal{D} \rangle$)$\neq$ true then
        return false;
    return true;
```

5.6 Encoding of architecture diagrams into configuration logics

To encode architecture diagrams we use the first-order configuration logic with ordered components, which was presented in Section 4.2.2. We start with the encoding of simple diagrams. Then we discuss the modifications necessary for obtaining the encodings of the other three classes of architecture diagrams. Since an empty configuration cannot be part of the model of a configuration logic formula, in the encoding we consider only non-empty

Function VerifyCardinality($\mathcal{B}$, $\langle \mathcal{T}, n, \mathcal{C}, \mathcal{D} \rangle$)

```
Data: Set of components $\mathcal{B}$, diagram $\langle \mathcal{T}, n, \mathcal{C}, \mathcal{D} \rangle$
Result: Returns $true$ if the number of components of each type in $\mathcal{B}$ lies in the corresponding
        cardinality interval specified in the diagram. Otherwise, it returns $false$.
begin
    /* Creates a map with key: component type, value: number of instances */
    $countTypes \longleftarrow \{\}$;
    for $B_i \in \mathcal{B}$ do
        $countTypes[typeof(B_i)] + +$;
    for $T_i \in \mathcal{T}$ do
        if $countTypes[T_i] \notin n(T_i)$ then
            return $false$;
    return $true$;
```

configurations defined by connector motifs.

For notational convenience, we introduce the following additional notations for a set of port variables $X[I] = \{x_l[i_l] \mid l \in [1, c]\}$. $\wp$ represents any of the following quantifiers $\exists, \Sigma, ⅁, \forall$.

- For a parameter $c \in \mathbb{N}$, we define:

$$\wp X[I] : T.p\,(|X[I]| = c)\,.\,F \overset{def}{=}$$

$$\wp x_1[i_1] : T.p\,.\,\wp x_2[i_2] : T.p\,(i_1 \neq i_2)\,.\,\ \ldots\ \wp x_c[i_c] : T.p\left(\bigwedge_{k=1}^{c-1} i_c \neq i_k\right).$$

$$F\Big[\{x_1[i_1], \ldots, x_c[i_c]\}/X[I], \{i_1, \ldots, i_c\}/I\Big]\,.$$

- For n sets of port variables $X_1[I_1], \ldots, X_n[I_n]$, we define

$$\sharp(X_1[I_1], \ldots, X_n[I_n]) \overset{def}{=} \sharp\left(\bigcup_{k=1}^{n} X_k[I_k]\right).$$

For the sake of simplicity, in the following subsections we omit the set of indices I if there is no associated predicate.

Encoding of simple architecture diagrams

Consider a simple architecture diagram $\mathcal{AD} = \langle \mathcal{T}, n, \mathcal{C} \rangle$. In the encoding we consider $\mathcal{T}$ as a given set of component types. Constraints are imposed by the cardinality $n(T)$ of each component type and the connector motifs in $\mathcal{C}$. Cardinalities are independent and thus, we can define a constraint per cardinality. Their combined constraint is obtained by conjuncting all cardinality constraints. The constraints imposed by connector motifs are also independent and their semantics is defined as a disjoint partition of a configuration such that each part corresponds to a different connector motif. Thus, their combined constraint is obtained by

Function VerifyMultiplicity(γ, $\mathcal{C}$)

Data: Configuration γ, Set of connector motifs $\mathcal{C}$

Result: Returns a partition $\mathcal{P}_\gamma$ of γ such that each part corresponds to one $\Gamma \in \mathcal{C}$. Connectors in each part satisfy multiplicity constraint of the corresponding connector motif. If no partition exists, returns $\emptyset$.

```
begin
    /* Creates a map for the partition with key: connector motif and value: sub-configuration.*/
    $partition \longleftarrow \{\}$;
    for $\Gamma \in \mathcal{C}$ do
        $partition[\Gamma] \longleftarrow \emptyset$;
    /* Creates a map for the single choice intervals with key: port type of a connector motif and
    value: chosen value.*/
    $scValues \longleftarrow \{\}$;

    for $k \in \gamma$ do
        /* Creates a map with key: port type and value: number of instances of the port type in the
        connector. */
        $portsCount \longleftarrow \{\}$;
        for $p_i \in k$ do
            $portsCount[generic(p_i)] + +$;
        $x \longleftarrow false$;
        /* Tries to find a connector motif such that connector satisfies its constraints.*/
        for $\Gamma \in \mathcal{C}$ do
            if $a = keys(portsCount)$ then
                $y \longleftarrow true$;
                /* Checks the multiplicity intervals.*/
                for $p \in a$ do
                    if $portsCount[p] \notin [m_p^l, m_p^u]$ then
                        $y \longleftarrow false$;
                        break;
                    /* Additional check in case of the single choice interval.*/
                    if $ty[m_p^l, m_p^u] = sc[m_p^l, m_p^u]$ then
                        if $hasKey(scValues, \langle\Gamma, p\rangle)$ && $scValues[\langle\Gamma, p\rangle] \neq portsCount[p]$ then
                            $y \longleftarrow false$;
                            break;
                        else
                            $scValues[\langle\Gamma, p\rangle] \longleftarrow portsCount[p]$;
                /* Checks if the multiplicity predicate $\mathcal{M}$ is satisfied.*/
                if $MultiplicityPredicate(portsCount, \mathcal{M}) = false$ then
                    $y \longleftarrow false$;
                if $y$ then
                    $partition[\Gamma] \longleftarrow partition[\Gamma] \cup k$;
                    $x \longleftarrow true$;
                    break;
        /* A connector does not satisfy constraints of any connector motif. */
        if $x = false$ then
            return $\emptyset$;
    return partition;
```

Function VerifyIndex(γ_i, Γ)

```
Data: Configuration $\gamma_i$, Connector motif $\Gamma$
Result: Returns true if the index requirements are satisfied. Otherwise returns false.
begin
    for $k \in \gamma_i$ do
        /* Creates a two dimensional array: [port type] [port indices participating in the connector
        k] */
        $ParticipatingPorts \longleftarrow \{\}$;

        /* Iterates through port types. */
        for $p \in a$ do
            $ParticipatingPorts[p] \longleftarrow \{\}$;

            /* Iterates through possible indices and denotes which index participates in the
            connector k. */
            for $i \in I_p$ do
                if $p_i \in k$ then
                    $ParticipatingPorts[p][i] = 1$;
                else
                    $ParticipatingPorts[p][i] = 0$;

        /* Checks arithmetic predicates $\Phi$. */
        if $IndexPredicate(ParticipatingPorts, \Phi) = false$ then
            return $false$;

    return $true$;
```

Function VerifyDegree(γ_i, Γ)

```
Data: Configuration $\gamma_i$, Connector motif $\Gamma$
Result: Returns true if the degree requirements are satisfied. Otherwise returns false.
begin
    /* Creates a map with key: port and value: number of connectors it appears in.*/
    $degrees \longleftarrow \{\}$;

    for $k \in \gamma_i$ do
        for $p_i \in k$ do
            $degrees[p_i] + +$;

    /* Creates a map for the single choice intervals with key: port type and value: chosen value.*/
    $scValues \longleftarrow \{\}$;

    for $p_i \in keys(degrees)$ do
        $p \longleftarrow generic(p_i)$;
        if $degrees[p_i] \notin [d^l_p, d^u_p]$ then
            return $false$;
        /* Additional check in case of the single choice interval.*/
        if $ty[d^l_p, d^u_p] = sc[d^l_p, d^u_p]$ then
            if $hasKey(scValues, p)$ && $scValues[p] \neq degrees[p_i]$ then
                return $false$;
            else
                $scValues[p] \longleftarrow degrees[p_i]$;

    return $true$;
```

Function VerifyDegreePredicates($\mathcal{B}$, $partition$, $\langle\mathcal{T}, n, \mathcal{C}, \mathcal{D}\rangle$)

Data: Set of components $\mathcal{B}$, Partition $partition$, Diagram $\langle\mathcal{T}, n, \mathcal{C}, \mathcal{D}\rangle$)
Result: Returns $true$ if the degree requirements $\mathcal{D}$ are satisfied. Otherwise returns false.

```
begin
    /* Iterates through all component instances. */
    for $B \in \mathcal{B}$ do
        $i = 0$;
        $Degrees \longleftarrow \{\}$;

        /* Iterates through all $\gamma_i$ in the partition. */
        /* Finds degree of each port instance for each $\gamma_i$. */
        for $\gamma_i \in partition$ do
            $Degrees[i] = \longleftarrow \{\}$;
            for $p \in typeof(B).P$ do
                $Degrees[i][p] = 0$;
                for $k \in \gamma_i$ do
                    if $p_t \in k$ then
                        $Degrees[i][p] = Degrees[i][p] + 1$;
            $i = i + 1$;
        /* Checks degree predicates $\mathcal{D}$. */
        if $DegreePredicate(Degrees, \mathcal{D}) = false$ then
            return $false$;

return $true$;
```

coalescing all connector motif constraints. An encoding of simple architecture diagrams can be defined as follows:

$$Encoding(\mathcal{AD}) \overset{def}{=} \bigwedge_{T \in \mathcal{T}} CardinalityConstraint(T) \wedge \sum_{\Gamma \in \mathcal{C}} ConnectorMotifConstraint(\Gamma)\,. \quad (5.5)$$

Cardinality constrains the number of component instances per component type. It can be encoded as follows:

$$CardinalityConstraint(T) \overset{def}{=} \exists c[i] : T\,\big(i = n(T)\big)\,.\,true \wedge\ \forall c[i] : T\,\big(i > n(T)\big)\,.\,false\,. \quad (5.6)$$

Each connector motif $\Gamma = \big(\{T_1.p_1, \ldots, T_n.p_n\}, \{m_p, d_p\}\big)$ imposes two types of constraints: 1) a multiplicity constraint; 2) a set of degree constraints, one constraint per port type. Thus, we have

$$ConnectorMotifConstraint(\Gamma) \overset{def}{=} MultiplicityConstraint(\Gamma)\ \wedge\ \bigwedge_{i=1}^{n} DegreeConstraint(\Gamma, T_i.p_i)\,. \quad (5.7)$$

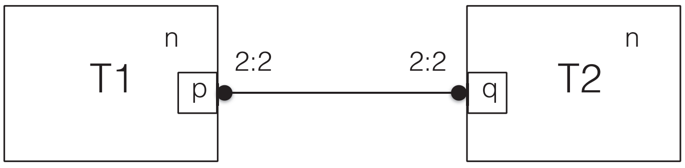


Figure 5.28 – The simple architecture diagram encoded in Example 5.6.1

Multiplicity constrains the number of involved port instances in each connector. It can be encoded as follows:

$$\begin{aligned} \mathit{MultiplicityConstraint}(\Gamma) = {} & ⅁ X_1 : T_1.p_1\,(|X_1| = m_{p_1}) \,.\, \ldots \\ & ⅁ X_n : T_n.p_n\,(|X_n| = m_{p_n}) \,.\, \sharp(X_1, \ldots, X_n)\,, \end{aligned} \tag{5.8}$$

$\sharp(X_1, \ldots, X_n)$ allows only connectors that consist of $m_{p_1}, \ldots, m_{p_n}$ instances of generic ports $p_1, \ldots, p_n$, respectively. The ⅁ quantifier allows to take any number of such connectors.

Degree constrains the number of connectors attached to any instance of a port type. For a generic port p, it can be encoded as follows:

$$\begin{aligned} & \forall x[k] : T.p \,.\, \exists X_1 : U.P \,.\, \ldots \,\exists X_{d_p} : U.P \left(\bigwedge_{i=1}^{d_p} x[k] \in X_i \wedge \bigwedge_{i \neq j} X_i \neq X_j \right) . \\ & \qquad \sum_{i=1}^{d_p} \sharp(X_i) \sqcup \sum_{i=1}^{d_p} \sharp(X_i) + \overline{x[k]}\,, \end{aligned} \tag{5.9}$$

For each port instance of the port type p, the formula (5.9) defines d_p sets of ports, one set per connector that involves the port instance. The union operator allows us to specify all possible configurations. $\sum_{i=1}^{d_p} \sharp(X_i)$ specifies the case that all connectors are attached to the port instance of p. $\sum_{i=1}^{d_p} \sharp(X_i) + \overline{x[k]}$ specifies the case that there is at least one connector that is not attached to the port instance of p.

Example 5.6.1. Consider the diagram shown in Figure 5.28 with two component types with cardinalities $n(T_1) = n(T_2) = n$ and a connector motif between port types p and q with $m_p = d_p = m_q = d_q = 2$. The corresponding encoding is the conjunction of the following three constraints:

- Cardinality constraints (5.6):

$$\begin{aligned} & \exists c[i] : T_1\,(i = n) \,.\, \mathit{true} \wedge \forall c[i] : T_1\,(i > n) \,.\, \mathit{false} \\ & \qquad \wedge\ \exists c[j] : T_2\,(j = n) \,.\, \mathit{true} \wedge \forall c[j] : T_2\,(j > n) \,.\, \mathit{false}\,. \end{aligned}$$

- Multiplicity constraint (5.8):

$$\text{Ɑ} X : T_1.p\,(|X| = 2)\,.\,\text{Ɑ} Y : T_2.q\,(|Y| = 2)\,.\,\sharp(X, Y).$$

- Degree constraints (5.9):

$$\begin{aligned}
&\forall x[i] : T_1.p\,.\,\exists X_1 : U.P\,\big(x[i] \in X_1\big)\,.\\
&\quad \exists X_2 : U.P\,\big(X_1 \neq X_2 \ \wedge\ x[i] \in X_2\big)\,.\\
&\qquad \sharp(X_1, \{x[i]\}) + \sharp(X_2, \{x[i]\}) \sqcup \sharp(X_1, \{x[i]\}) + \sharp(X_2, \{x[i]\}) + \overline{x[i]}\\
&\wedge\\
&\forall x[i] : T_2.q\,.\,\exists X_1 : U.P\,\big(x[i] \in X_1\big)\,.\\
&\quad \exists X_2 : U.P\,\big(X_1 \neq X_2 \ \wedge\ x[i] \in X_2\big)\,.\\
&\qquad \sharp(X_1, \{x[i]\}) + \sharp(X_2, \{x[i]\}) \sqcup \sharp(X_1, \{x[i]\}) + \sharp(X_2, \{x[i]\}) + \overline{x[i]}\ .
\end{aligned}$$

  Notice that the two conjuncts of the formula differ only in the type of the variable in the first quantification—thus, the formula can be simplified by quantifying instead over $x[i] : U.P$. We keep this presentation for the sake of consistency with the encoding in (5.7).

Example 5.6.2. Consider the architecture diagram of Example 5.1.7 that describes the Multi-star architecture style. The encoding of this style is the conjunction of the following three constraints:

- Cardinality constraints:

$$\begin{aligned}
&\exists c[i] : T_1\,(i = n)\,.\,\mathit{true} \wedge \forall c[i] : T_1\,(i > n)\,.\,\mathit{false}\\
&\qquad\qquad \wedge\ \exists c[j] : T_2\,(j = n \cdot d)\,.\,\mathit{true} \wedge \forall c[j] : T_2\,(j > n \cdot d)\,.\,\mathit{false}\,.
\end{aligned}$$

- Multiplicity constraint:

$$\text{Ɑ} x[i] : T_1.p\,.\,\text{Ɑ} y[j] : T_2.q\,.\,\sharp\{x[i], y[j]\}\,.$$

- Degree constraints:

$$\begin{aligned}
&\forall y[j] : T_2.q\,.\,\exists x[i] : T_1.p\,.\,\sharp\{x[i], y[j]\} \sqcup \sharp\{x[i], y[j]\} + \overline{y[j]}\\
&\wedge\\
&\forall x[i] : T_1.p\,.\,\exists y[j_1] : T_2.q\,.\,\exists y[j_2] : T_2.q(j_1 \neq j_2)\,.\,\ldots\\
&\qquad \exists y[j_d] : T_2.q\,.\,\Big(\bigwedge_{k=1}^{d} j_d \neq j_k\Big)\,.\\
&\qquad\quad \sum_{k=1}^{d} \sharp\{x[i], y[j_k]\} \sqcup \sum_{k=1}^{d} \sharp\{x[i], y[j_k]\} + \overline{x[i]}\ .
\end{aligned}$$

Encoding of interval architecture diagrams

Consider an interval architecture diagram $\mathcal{AD} = \langle \mathcal{T}, n, \mathcal{C} \rangle$. Similarly to simple architecture diagrams, there are constraints imposed by cardinalities and connector motifs.

$$\begin{aligned} &\mathit{Encoding}(\mathcal{AD}) = \\ &\qquad \bigwedge_{T \in \mathcal{T}} \mathit{CardinalityConstraints}(T) \ \wedge \sum_{\Gamma \in \mathcal{C}} \mathit{ConnectorMotifConstraint}(\Gamma). \end{aligned} \tag{5.10}$$

The cardinality function is now interval-valued. The constraint that the number of components of type T must be within the interval $[n^l, n^u]$ can be represented as follows:

$$\exists c[i]\!:\!T\,(i = n^l)\,.\,\mathit{true} \ \wedge \ \forall c[i]\!:\!T\,(i > n^u)\,.\,\mathit{false}. \tag{5.11}$$

As in simple architecture diagrams, the constraint imposed by a connector motif $\Gamma = \Big(\{T_1.p_1, \dots, T_n.p_n\}, \{ty[m_p^l, m_p^u] : ty[d_p^l, d_p^u]\}\Big)$ can be encoded as follows:

$$\begin{aligned} &\mathit{ConnectorMotifConstraint}(\Gamma) \overset{def}{=} \\ &\qquad \mathit{MultiplicityConstraint}(\Gamma) \ \wedge \ \bigwedge_{i=1}^{n} \mathit{DegreeConstraint}(\Gamma, T_i.p_i)\,. \end{aligned} \tag{5.12}$$

Multiplicity requires that the number of involved port instances in each connector be within the given interval. By choosing a specific value from each multiplicity interval, we specify a set of connectors that involve the chosen number of port instances of each port type. This set of values can be encoded with (5.8). All connectors that are defined by the connector motif can be obtained by taking all values from all multiplicity intervals. The types of the intervals define what subsets of these connectors can form configurations. The *mc* type does not forbid connectors with different multiplicities, thus we take a disjunction over all values in the *mc*-intervals. On the other hand, the *sc* type allows only connectors with the same multiplicity, so we take a union between values of *sc*-intervals.

Assume that multiplicities of ports $T_1.p_1, \dots, T_k.p_k$ have type *sc* and multiplicities of ports $T_{k+1}.p_{k+1}, \dots, T_n.p_n$ have type *mc*. Since ports in the connector motif are not ordered, this assumption does not restrict the encoding and is only used to simplify the formula. The multiplicity constraint can be encoded as follows:

$$\mathit{MultiplicityConstraint}(\Gamma) = \bigsqcup_{m_{p_1}=m^l_{p_1}}^{m^u_{p_1}} \dots \bigsqcup_{m_{p_k}=m^l_{p_k}}^{m^u_{p_k}} \bigvee_{m_{p_{k+1}}=m^l_{p_{k+1}}}^{m^u_{p_{k+1}}} \dots \bigvee_{m_{p_n}=m^l_{p_n}}^{m^u_{p_n}} \mathit{MConstraint}(m_{p_1}, \dots, m_{p_n}), \quad (5.13)$$

where *MConstraint* is defined similarly to (5.8):

$$\mathit{MConstraint}(m_{p_1}, \dots, m_{p_n}) = \text{Ɑ} X_1 : T_1.p_1\,(|X_1| = m_{p_1})\,.\,\dots\; \text{Ɑ} X_n : T_n.p_n\,(|X_n| = m_{p_n})\,.\,\sharp(X_1, \dots, X_n)\,. \quad (5.14)$$

The degree associated to a port type p within a connector motif requires all port instances to be involved in at least d^l_p and at most d^u_p connectors. This is encoded as a union of formulas similar to (5.9) for each value in $[d^l_p, d^u_p]$. The type of the interval controls the order of operators in the encoding. For *sc*-intervals, degrees have to be equal for all port instances, while for all ports with *mc*-interval degrees, we can take any value within the interval. The degree constraint can be encoded as follows:

$$\mathit{DegreeConstraint}(\Gamma, T.p) = \begin{cases} \bigsqcup_{d\in[d^l_p,d^u_p]} \forall x[k] : T.p\,.\,\mathit{DConstraint}(\Gamma, x[k], d)\,, & \text{if } ty = sc\,, \\ \forall x[k] : T.p\,.\,\bigsqcup_{d\in[d^l_p,d^u_p]} \mathit{DConstraint}(\Gamma, x[k], d)\,, & \text{if } ty = mc\,, \end{cases} \quad (5.15)$$

where $\mathit{DConstraint}(\Gamma, x[k], 0) = \overline{x[k]}$ and

$$\mathit{DConstraint}(\Gamma, x[k], d \neq 0) = \exists X_1 : U.P\,.\,\dots\,\exists X_d : U.P \Big(\bigwedge_{i=1}^{d_p} x[k] \in X_i \wedge \bigwedge_{i\neq j} X_i \neq X_j \Big)\,.\; \sum_{i=1}^{d} \sharp(X_i, \{x[k]\}) \sqcup \sum_{i=1}^{d} \sharp(X_i, \{x[k]\}) + \overline{x[k]}\,. \quad (5.16)$$

Since the chosen value can be 0, we distinguish this special case.

Example 5.6.3. Consider the architecture diagram of Example 5.2.8 that describes the Repository architecture style. The diagram is encoded as the conjunction of the following constraints:

- Cardinality constraints of R and A (5.12):

$$\exists c[i] : R\,(i = 1)\,.\, \mathit{true} \wedge \forall c[i] : R\,(i > 1)\,.\, \mathit{false} \\ \wedge\ \exists c[i] : A\,(i = n_2)\,.\, \mathit{true} \wedge \forall c[i] : A\,(i > n_2)\,.\, \mathit{false}\,.$$

- Multiplicity constraint (5.13):

$$\bigvee_{m_q=1}^{n_2} \text{Ɑ} p[i] : R.p\,.\, \text{Ɑ} Q : A.q\,(|Q| = m_q)\,.\, \sharp(Q, \{p[i]\})\,.$$

- Degree constraint of $R.p$ (5.15):

$$\forall x[k] : R.p\,.\, \bigsqcup_{d \in [1,n_2]} \exists X_1 : U.P\,.\, \ldots \\ \exists X_d : U.P\, \Big(\bigwedge_{i=1}^{d_p} x[k] \in X_i \wedge \bigwedge_{i \neq j} X_i \neq X_j \Big)\,. \\ \sum_{i=1}^{d} \sharp(X_i, \{x[k]\}) \sqcup \sum_{i=1}^{d} \sharp(X_i, \{x[k]\}) + \overline{x[k]}\ .$$

- Degree constraint of $A.q$ (5.15):

$$\forall x[k] : A.q\,.\, \exists X : U.P\,.\, \sharp(X, \{x[k]\}) \sqcup \sharp(X, \{x[k]\}) + \overline{x[k]}\ .$$

Notice that the degree constraint of $R.p$ is implied by the other constraints and it can be excluded from the formula.

Encoding of index architecture diagrams

Index architecture diagrams introduce arithmetic predicates over the set of index variables on ports. Arithmetic predicates are associated with connector motifs. They are encoded as part of the *MultiplicityConstraint* of the corresponding connector motif. For a connector motif $\Gamma = (\{T_1.p_1, \ldots, T_n.p_n\}, \{m_p : d_p\}, \Phi(\mathcal{I}))$ with an associated predicate $\Phi(\mathcal{I})$, multiplicity can be encoded as follows:

$$\mathit{MultiplicityConstraint}(cm) = \text{Ɑ} X_1[I_1] : T_1.p_1\,(|X_1[I_1]| = m_{p_1})\,.\, \ldots \\ \text{Ɑ} X_n[I_n] : T_n.p_n\,(|X_n[I_n]| = m_{p_n} \wedge \Phi(I_1 \cup \cdots \cup I_n))\,. \\ \sharp(X_1[I_1], \ldots, X_n[I_n])\,. \quad (5.17)$$

(5.17) differs from (5.8) in that in the last quantification the extra condition $\Phi(I_1 \cup \cdots \cup I_n))$ is introduced that ensures the satisfaction of the predicate.

Example 5.6.4. Consider the architecture diagram of Example 5.3.3 that describes the Ring architecture style. The encoding of this style is a conjunction of the cardinality, multiplicity and degree constraints:

$$
\begin{aligned}
&\exists c[i]\!:\!T\ (i = n)\ .\ \mathit{true} \wedge \forall c[i]\!:\!T\ (i > n)\ .\ \mathit{false} \\
&\wedge \\
&\text{Ɑ}\mathit{out}[i]\!:\!T.\mathit{out}\ .\ \text{Ɑ}\mathit{in}[j]\!:\!T.\mathit{in}\,(j = i + 1\ \mathit{mod}\ n)\ .\ \sharp\{\mathit{out}[i], \mathit{in}[j]\} \\
&\wedge \\
&\forall r[i]\!:\!T.P\ .\ \exists r[j]\!:\!T.P\,(r[i] \neq r[j])\ .\ \sharp\{r[i], r[j]\} \sqcup \sharp\{r[i], r[j]\} + \overline{r[i]}\ .
\end{aligned}
$$

Notice the quantification over $r[i]\!:\!T.P$ in the degree constraint instead of the two conjuncts for each port type.

Encoding of general architecture diagrams

General architecture diagrams combine both extensions of simple architecture diagrams and additionally have a set of predicates $\mathcal{D}$ and $\mathcal{M}$ over the set of degree and multiplicity variables. Consider a general architecture diagram $\langle \mathcal{T}, n, \mathcal{C}, \mathcal{D} \rangle$. An encoding of general diagrams can be defined as follows:

$$
\begin{aligned}
&\mathit{Encoding}(\mathcal{AD}) = \\
&\qquad \bigwedge_{T \in \mathcal{T}} \mathit{CardinalityConstraints}(T) \ \wedge \sum_{\Gamma \in \mathcal{C}} \mathit{ConnectorMotifConstraint}(\Gamma) \\
&\qquad\qquad \wedge \bigwedge_{D_T \in \mathcal{D}} \mathit{PredicateOnDegrees}(D_T), \qquad (5.18)
\end{aligned}
$$

where the cardinality constraint is defined as in interval diagrams 5.11.

The constraint imposed by a connector motif
$\Gamma = \Big(\{T_1.p_1, \ldots, T_n.p_n\}, \{ty[m_p^l, m_p^u] : ty[d_p^l, d_p^u]\}, \{m_p\}, \{d_p\}, \mathcal{M}\Big)$ can be encoded as follows:

$$
\begin{aligned}
&\mathit{ConnectorMotifConstraint}(\Gamma) \overset{def}{=} \\
&\qquad \mathit{MultiplicityConstraint}(\Gamma) \ \wedge \bigwedge_{i=1}^{n} \mathit{DegreeConstraint}(\Gamma, T_i.p_i)\,, \qquad (5.19)
\end{aligned}
$$

where the degree constraint is defined as in interval diagrams 5.15.

Assume that multiplicities of ports $T_1.p_1, \ldots, T_k.p_k$ have type *sc* and multiplicities of ports $T_{k+1}.p_{k+1}, \ldots, T_n.p_n$ have type *mc*. Since ports in the connector motif are not ordered, this assumption does not restrict the encoding and is only used to simplify the formula. The multiplicity constraint with the predicate $\mathcal{M}$ can be defined as follows:

$$
\begin{aligned}
\mathit{MultiplicityConstraint}(\Gamma) = & \bigsqcup_{m_{p_1}=m^l_{p_1}}^{m^u_{p_1}} \cdots \bigsqcup_{m_{p_k}=m^l_{p_k}}^{m^u_{p_k}} \\
& \bigvee_{m_{p_{k+1}}=m^l_{p_{k+1}}}^{m^u_{p_{k+1}}} \cdots \bigvee_{\substack{m_{p_n}=m^l_{p_n} \\ \mathcal{M}(m_{p_1},\ldots,m_{p_n})}}^{m^u_{p_n}} \mathit{MConstraint}(m_{p_1},\ldots,m_{p_n})\,, \qquad (5.20)
\end{aligned}
$$

where *MConstraint* is defined as in interval diagrams (5.14):

$$
\begin{aligned}
\mathit{MConstraint}(m_{p_1},\ldots,m_{p_n}) = {} & \text{Ɑ}X_1[I_1]\!:\!T_1.p_1\,(|X_1[I_1]| = m_{p_1})\,.\ \ldots \\
& \text{Ɑ}X_n[I_n]\!:\!T_n.p_n\,(|X_n[I_n]| = m_{p_n}\ \wedge\ \Phi(I_1\cup\cdots\cup I_n))\,. \\
& \sharp(X_1[I_1],\ldots,X_n[I_n])\,, \qquad (5.21)
\end{aligned}
$$

where Φ is a predicate over the set of index variables.

Let us now encode the degree predicates of general diagrams. Each degree predicate $D_T \in \mathcal{D}$ defines a constraint over a set of degree variables $\{\Gamma_p.d_p \mid \Gamma \in \mathcal{C}, p \in \Gamma.a \cap T.P\}$ associated with the port types in $T.P$. Based on the degree intervals associated with each port type, the predicate D_T has a set of possible solutions $\{\langle d^i_p, d^i_q, \ldots, d^i_r\rangle\}_{i\in I}$. The degrees of ports of each component must satisfy one of the solutions. Thus, the predicate can be encoded as follows:

$$
\mathit{PredicateOnDegrees}(D_T) \ = \ \forall c[j] \ : \ T\,.\bigsqcup_{i\in I}\sum_{p\in T.P}\mathit{DConstraint}(\Gamma_p, c[j].p, d^i_p)\,, \qquad (5.22)
$$

where *DConstraint* is defined by the formula (5.16).

Example 5.6.5. Consider the architecture diagram of Example 5.4.3 that describes the Peer-to-Peer architecture style. The constraints defined by this diagram are encoded as follows:

$$
\begin{aligned}
& \exists c[i]\!:\!P\,(i = n)\,.\,\mathit{true}\ \wedge\ \forall c[i]\!:\!P\,(i > n)\,.\,\mathit{false} \\
\wedge\ & \text{Ɑ}x[j]\!:\!P.p\,.\,\text{Ɑ}y[i]\!:\!P.r\,(i \neq j)\,.\,\sharp\{x[j], y[i]\} \\
\wedge\ & \forall x[k]\!:\!U.P\,. \\
& \bigsqcup_{d\in[1,n]}\Big(\exists x[i_1]\!:\!U.P\,.\ \ldots\ \exists x[i_d]\!:\!U.P\,(\bigwedge_{j=1}^{d} x[i_j] \neq x[k]\ \wedge\ \bigwedge_{j\neq j'} x[i_j] \neq x[i_{j'}])\,. \\
& \qquad \sum_{j=1}^{d}\sharp\{x[k], x[i_j]\} \sqcup \sum_{j=1}^{d}\sharp\{x[k], x[i_j]\} + \overline{x[k]}\Big) \sqcup \overline{x[k]}\,.
\end{aligned}
$$

Example 5.6.6. Consider the architecture diagram of Example 5.4.5 describing the TMR architecture style. The encoding of the diagram is a conjunction of the following constraints:

- Cardinality constraints for all component types:

$$\begin{aligned}
&\exists c[i] : IM\, (i = 3) \,.\, true \;\wedge\; \forall c[i] : IM\, (i > 3) \,.\, false\\
\wedge\;& \exists c[i] : M\, (i = 3) \,.\, true \;\wedge\; \forall c[i] : M\, (i > 3) \,.\, false\\
\wedge\;& \exists c[i] : V\, (i = 1) \,.\, true \;\wedge\; \forall c[i] : V\, (i > 3) \,.\, false\\
\wedge\;& \exists c[i] : OM\, (i = 3) \,.\, true \;\wedge\; \forall c[i] : OM\, (i > 3) \,.\, false.
\end{aligned}$$

- Coalescing of constraints for all connector motifs (all represented as the conjunction of multiplicity constraint and two degree constraints):
  - Connector motif between IM and M: both multiplicities and both degrees are equal to 1.

$$\begin{aligned}
&\text{Ɑ} x[i] : IM.out \,.\, \text{Ɑ} y[j] : M.in \,.\, \sharp\{x[i], y[j]\}\\
\wedge\;& \forall x[i] : IM.out \,.\, \exists y[j] : M.in \,.\, \sharp\{x[i], y[j]\} \sqcup \sharp\{x[i], y[j]\} + \overline{x[i]}\\
\wedge\;& \forall y[j] : M.in \,.\, \exists x[i] : IM.out \,.\, \sharp\{x[i], y[j]\} \sqcup \sharp\{x[i], y[j]\} + \overline{y[j]} \,.
\end{aligned}$$

  - Connector motif between M and V: both multiplicities are equal to 1; degree of $M.out$ is $sc[1,3]$, thus we take a union of three cases; degree of $V.in$ is 3.

$$\begin{aligned}
&\text{Ɑ} x[i] : M.out \,.\, \text{Ɑ} y[j] : V.in \,.\, \sharp\{x[i], y[j]\}\\
\wedge\;& \Bigg(\Big(\forall x[i] : M.out \,.\, \exists y[j] : V.in \,.\, \sharp\{x[i], y[j]\} \sqcup \sharp\{x[i], y[j]\} + \overline{x[i]}\Big)\\
&\sqcup \Big(\forall x[i] : M.out \,.\, \exists y[j_1] : V.in \,.\, \exists y[j_2] : V.in\, (j_1 \neq j_2) \,.\\
&\qquad \sum_{k=1}^{2} \sharp\{x[i], y[j_k]\} \sqcup \sum_{k=1}^{2} \sharp\{x[i], y[j_k]\} + \overline{x[i]}\Big)\\
&\sqcup \Big(\forall x[i] : M.out \,.\, \exists y[j_1] : V.in \,.\, \exists y[j_2] : V.in\, (j_1 \neq j_2) \,.\\
&\qquad \exists y[j_3] : V.in (j_1 \neq j_3 \neq j_2) \,.\\
&\qquad\quad \sum_{k=1}^{3} \sharp\{x[i], y[j_k]\} \sqcup \sum_{k=1}^{3} \sharp\{x[i], y[j_k]\} + \overline{x[i]}\Big)\Bigg)\\
\wedge\;& \forall y[i] : V.in \,.\, \exists x[j_1] : M.out \,.\, \exists x[j_2] : M.out\, (j_1 \neq j_2) \,.\\
&\qquad \exists x[j_3] : M.out (j_1 \neq j_3 \neq j_2) \,.\\
&\qquad\quad \sum_{k=1}^{3} \sharp\{y[i], x[j_k]\} \sqcup \sum_{k=1}^{3} \sharp\{y[i], x[j_k]\} + \overline{y[i]}\Big) \,.
\end{aligned}$$

    Since there is a binary connector between every pair of M and V, the encoding can be simplified:

$$\Sigma x[i] : M.out \,.\, \Sigma y[j] : V.out \,.\, \sharp\{x[i], y[j]\} \,.$$

  - Connector motif between V and OM: multiplicity of $V.out$ is 1 and of $OM.in$ is

$mc[1,3]$, thus it is encoded as a disjunction of three cases; degree of $V.out$ is 1; degree of $OM.in$ is $mc[0,1]$ represented as a union of two cases for values 0 and 1.

$$
\begin{aligned}
&\text{Ɑ} x[i]\!:\!V.out\,.\,\Big(\text{Ɑ} y[j]\!:\!OM.in\,.\,\sharp\{x[i],y[j]\}\\
&\qquad\qquad \vee\ \text{Ɑ} P\!:\!OM.in\,(|P|=2)\,.\,\sharp(P,\{x[i]\})\\
&\qquad\qquad \vee\ \text{Ɑ} P\!:\!OM.in\,(|P|=3)\,.\,\sharp(P,\{x[i]\})\Big)\\
\wedge\ &\forall x[i]\!:\!V.out\,.\,\exists P\!:\!OM.in\,.\\
&\qquad\qquad \sharp(P,\{x[i]\})\sqcup\sharp(P,\{x[i]\})+\overline{x[i]}\\
\wedge\ &\forall y[i]\!:\!OM.in\,.\,\Big(\overline{y[i]}\ \sqcup\ \exists P\!:\!U.P\,.\\
&\qquad\qquad \sharp(\{P,y[i]\})\sqcup\sharp(\{P,y[i]\})+\overline{y[i]}\ \Big)\,.
\end{aligned}
$$

– Connector motif between M and OM: both multiplicities are equal to 1; both degrees are $mc[0,1]$ represented as unions of two cases.

$$
\begin{aligned}
&\text{Ɑ} x[i]\!:\!M.out\,.\,\text{Ɑ} y[j]\!:\!OM.in\,.\,\sharp\{x[i],y[j]\}\\
\wedge\ &\forall x[i]\!:\!M.out\,.\,\Big(\overline{x[i]}\ \sqcup\ \exists y[j]\!:\!OM.in\,.\\
&\qquad\qquad \sharp\{x[i],y[j]\}\sqcup\sharp\{x[i],y[j]\}+\overline{x[i]}\ \Big)\\
\wedge\ &\forall y[i]\!:\!OM.in\,.\,\Big(\overline{y[i]}\ \sqcup\ \exists x[j]\!:\!M.out\,.\\
&\qquad\qquad \sharp\{y[i],x[j]\}\sqcup\sharp\{y[i],x[j]\}+\overline{y[i]}\ \Big)\,.
\end{aligned}
$$

- Predicate over the set of degree variables of port $OM.in$ that requires the sum of two degrees be equal to 1. There exist two solutions: $\langle 0,1\rangle$ and $\langle 1,0\rangle$.

$$
\begin{aligned}
&\forall x[j]\!:\!OM.in\,.\\
&\quad\Big(\overline{x[j]}\ +\ \Big(\exists y[i]\!:\!V.out\,.\,\sharp\{x[j],y[i]\}\sqcup\sharp\{x[j],y[i]\}+\overline{x[j]}\ \Big)\Big)\\
&\quad\sqcup\Big(\overline{x[j]}\ +\ \Big(\exists y[i]\!:\!M.out\,.\,\sharp\{x[j],y[i]\}\sqcup\sharp\{x[j],y[i]\}+\overline{in[j]}\ \Big)\Big).
\end{aligned}
$$

5.7 Related work

A plethora of approaches exist for architecture specification. Patterns [35, 51] are commonly used for specifying architectures in practical applications. The specification of architectures is usually done in a graphical way using general purpose graphical tools. Such specifications are easy to produce but the meaning of the design may not be clear since the graphical conventions lack formal semantics and thus, are not amenable to formal analysis.

A number of Architecture Description Languages (ADLs) have been developed for architecture specification [75, 104, 80]. Nevertheless, according to [69], architectural languages used in practice mostly originate from industrial development instead of academic research. Practitioners extensively use UML although many scientific questions remain about its formal properties [47]. On the contrary, some ADLs have formal semantics [4, 68, 34], however, they require the use of formal languages which are considered as difficult for practitioners to master [69]. To address this issue, we propose architecture diagrams that combine the benefits of graphical languages and rigorous formal semantics. By relying on a small set of notions, we emphasize the conceptual clarity of our approach.

A number of paradigms for unifying interactions in heterogeneous systems have been studied [40, 8, 56]. In these works, unification is achieved by reduction to a common low-level semantic model. Coordination mechanisms and their properties are not studied independently of behaviour. Architecture diagrams were developed to accommodate architecture specification in BIP [7], wherein connectors are relations among ports and do not carry any additional behaviour. This strict separation of computation from coordination allows reasoning about the coordination constraints structurally and independently from the behaviour of coordinating components. However, our approach can be extended to describe architecture styles in other languages by explicitly associating the required behaviour to connector motifs. For example, this can be applied to specify connector patterns in Reo [5], by associating multiplicity and degree to source and sink nodes of connectors. The main difficulty is to correctly instantiate the behaviour depending on the number of ends in the connector.

Architecture diagrams consist of component types and connector motifs, respectively comparable to UML components and associations [54, 78]. One important difference between connector motifs and UML associations is that the latter cannot specify interactions that involve two or more instances of the same component type [78]. In UML, the term “multiplicity” is used to define both 1) the number of instances of a UML component and 2) the number of UML links connected to a UML component. In architecture diagrams, we call these, respectively, “cardinality” and “degree”. We use the term “multiplicity” to denote the number of components of the same class that can be connected by the same connector. The distinction between multiplicity and degree is key for allowing n-ary connectors involving several instances of the same component type.

The use of context-free grammars [49, 64, 28, 62, 63] allows inductive definitions and reasoning about architectures and dynamic reconfigurations. Graph grammars can be used to define architecture styles: a style admits all the configurations that can be derived by its defining grammar. The downside is that such definitions require additional non-terminal symbols to represent variable size structures, e.g. list of all slaves in a Master/Slave architecture. We take a different approach, whereby all constraints appear directly in the architecture diagram for which we provide denotational semantics. The rationale is the following: we assume that the reasoning is carried out by an “expert”, who defines the architectural style, whereas the “user” only needs the minimal information in order to select and instantiate it. Thus, structural information, e.g. necessary information for an inductive proof that the style imposes a certain property, does not appear in the diagram, but only the entities that form the target system.

5.8 Summary

We presented architecture diagrams, a graphical language rooted in well-defined semantics for the description of architecture styles. Using architecture diagrams instead of purely logic-based specifications confers the advantages of graphical formalisms. We proposed four classes of architecture diagrams. Simple architecture diagrams express uniform degree and multiplicity constraints. Index architecture diagrams have in addition index variables on ports and arithmetic predicates which introduce constraints over the set of index variables. Interval architecture diagrams use intervals for defining cardinality, multiplicity and degree constraints. General architecture diagrams combine the constraints of index and interval diagrams and have in addition arithmetic predicates on multiplicities and degrees. We showed that the four classes of diagrams form a hierarchy. In particular, index and interval diagrams are strictly more expressive than simple architecture diagrams and general diagrams are strictly more expressive than all other classes of diagrams. We studied a full encoding of architecture diagrams into configuration logics and a polynomial-time algorithm for checking whether an architecture conforms to the architecture style specified by a general architecture diagram.

Additionally, we proposed a set of necessary and sufficient consistency conditions and a synthesis procedure for generating compositionally the set of conforming architectures for simple and interval diagrams. We are currently working on extending the synthesis procedure for index and general diagrams. Nevertheless, the set of consistency conditions cannot be generalised to index and general architecture diagrams, since these contain predicates involving port instances and thus, must be checked at the level of architecture and not at the level of architecture style. To check the consistency of index and general architecture diagrams, we can encode the diagrams into configuration logic formulas and check for non-satisfiability.

6 Case studies

In this chapter, we present two satellite on-board software case studies, CubETH and Sentinel 3, in which we applied the architecture-based approach to achieve correctness by construction. The CubETH case study is presented in detail, whereas for Sentinel 3 we give only a very high level presentation of the case-study due to confidentiality restrictions.

The architecture-based design approach consists of the following three stages (Figure 6.1):

1. **Pre-design**: architecture styles, which represent recurring patterns that are relevant for the application domain, are identified and formally defined. The taxonomy of architecture styles identified in the CubETH case study was reused in the Sentinel 3 case study, and it can be further reused in the design of other satellite on-board software. Ideally, this stage is realised only once for each application domain.
2. **Design**: requirements to be satisfied by the system are analysed and formalised. Next, atomic components realising the basic functionality of the system are designed and used as operands for the application of architectures instantiated from the styles defined in the first stage. The choice of which architectures to apply is driven by the requirements.
3. **Verification**: the resulting system is checked for deadlock-freedom. Properties which are not enforced by construction through architecture application must be verified a posteriori. In the CubETH case study, we illustrate all steps of this process, except the requirement formalisation.

6.1 The CubETH case study

CubETH is a nanosatellite based on the CubeSat standard [29]. It contains the following subsystems: `EPS` (electrical power subsystem), `CDMS` (command and data management subsystem), `COM` (telecommunication subsystem), `ADCS` (attitude determination and control subsystem), `PL` (payload) and the mechanical structure including the antenna deployment subsystem.

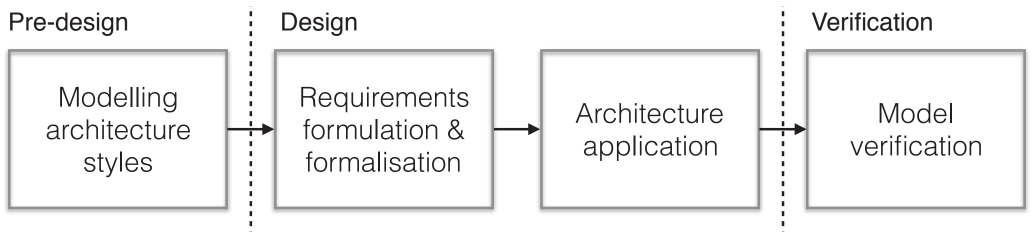


Figure 6.1 – Architecture-based design flow

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

A high-level BIP model of the case-study is shown in Figure 6.2. For the sake of simplicity, we omit some of the connectors. In particular, we show the connectors involving the `HK_to_MEM`, `HK_to_I2C` and `HK_to_I2C_NOFAIL` interfaces of the `HK_COM` subsystem, but we omit the respective connectors involving the other three Housekeeping subsystems. The `MESSAGE_LIBRARY`, `MEMORY_LIBRARY`, `I2C_sat_LIBRARY`, `s3_5`, `s3_6`, `s15_1` and `s15_2` components are atomic. The rest are composite components, i.e. *compounds*.

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

6.1.1 A taxonomy of architecture styles for on-board software

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

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

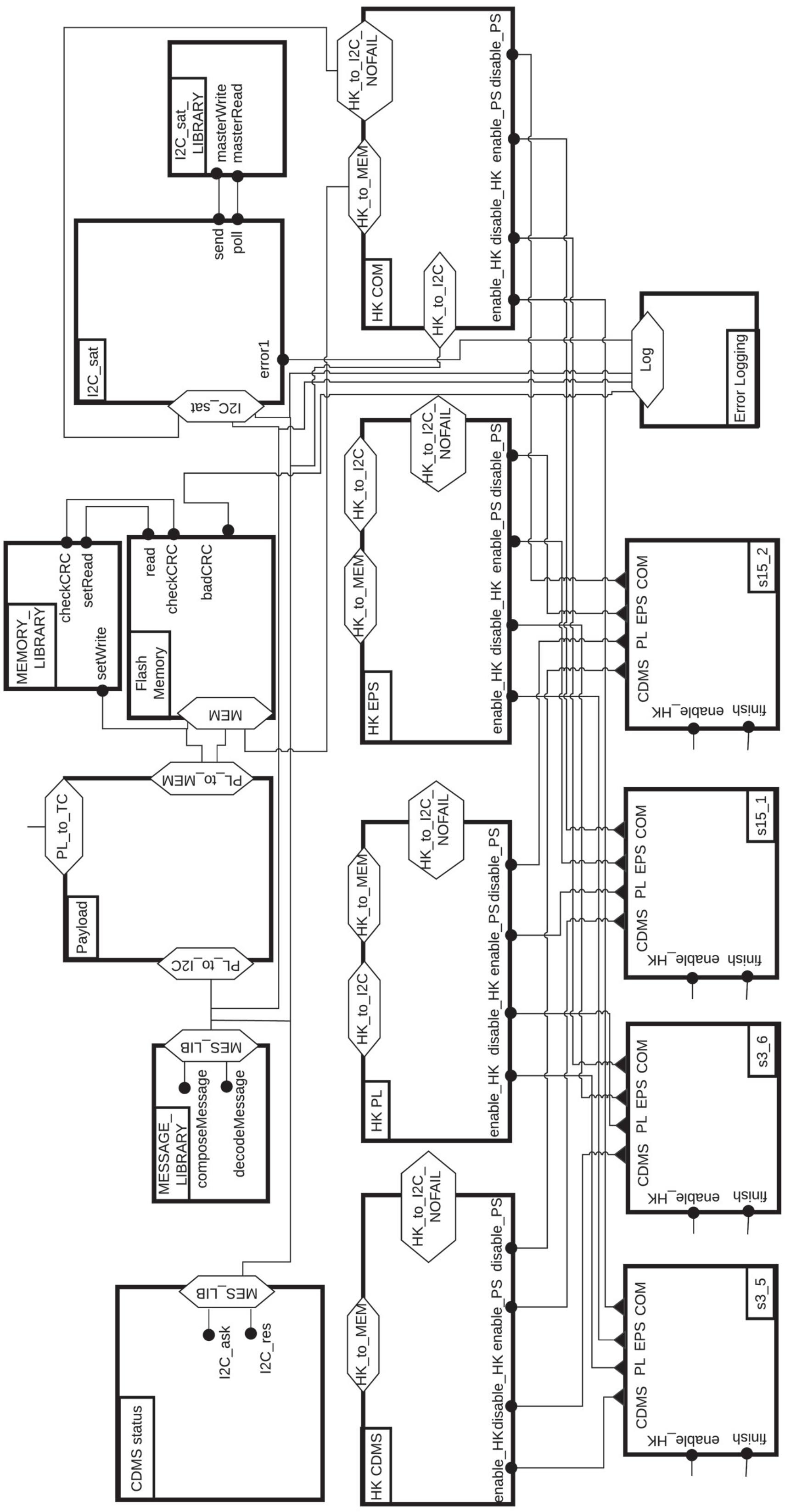


Figure 6.2 – The high-level interaction model

We have identified 9 recurring patterns, which we defined formally as architecture styles by using simple and interval architecture diagrams. For the sake of simplicity of the presentation, in the next subsections, we omit the cardinality of a port type if it is equal to 1. The cardinality of a component type is indicated right next to its name.

Mutual Exclusion style

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

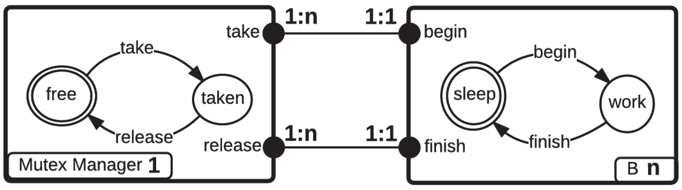


Figure 6.3 – Mutual Exclusion style with architecture diagrams

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

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

  where $cs[i]$ is an atomic predicate that evaluates to true when the $B[i]$ component is in the critical section (e.g. in state `work` for the behaviour shown in Figure 6.3).

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

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

Client-Server style

The Client-Server architecture style ensures that only one client can use a service offered by the server at any time. It consists of two parameter component types `Server` and `Client` with l and n instances, respectively. In the diagram of Figure 6.4, the Server provides two services through port types `offer`, `offer2`. Client has two corresponding port types `use`, `use2`. Since the cardinalities of `offer` and `offer2` are k and k', respectively, each component instance of type `Server` has k port instances of type `offer` and k' port instances of type `offer2`. Similarly, each component instance of type `Client` has m port instances of type `use` and m' port instances of type `use2`.

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

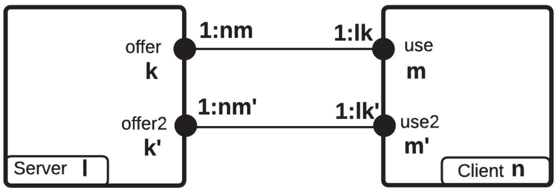


Figure 6.4 – Client-Server style with architecture diagrams

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

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

Action flow style

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

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

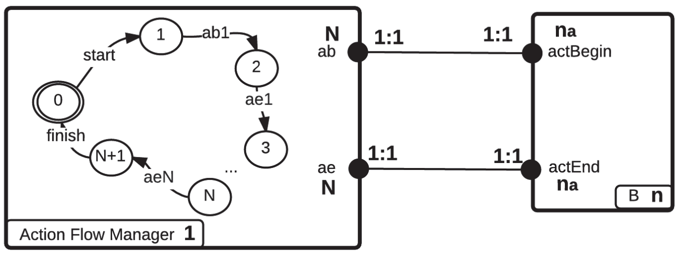


Figure 6.5 – Action flow style with architecture diagrams

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

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

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

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

  We consider the following mappings:

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

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

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

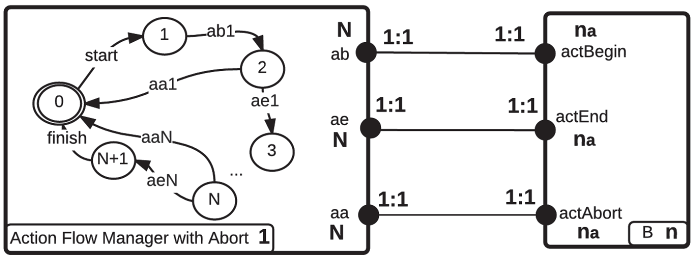


Figure 6.6 – Action flow with abort style with architecture diagrams

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

Action flow with abort style

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

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

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

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

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

We consider the following mappings:

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

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

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

Failure monitoring style

The Failure monitoring (Figure 6.7) provides monitor components that observe the state of other components. It consists of n coordinator components of type `Failure Monitor` and n parameter components of type `B1`. The cardinality of all port types is 1. Multiplicities and degrees require that each `B1` component instance be connected to its dedicated `Failure monitor` instance.

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

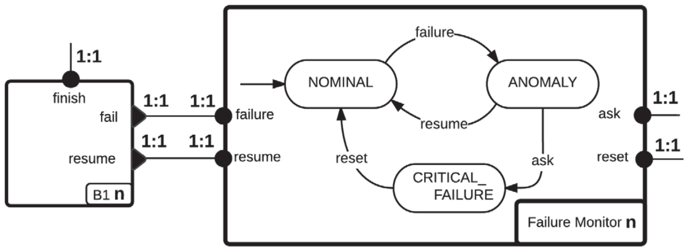


Figure 6.7 – Failure monitoring style with architecture diagrams

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

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

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

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

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

Mode management style

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

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

`B1` has k port types `modeBegin` with cardinality `mc`$[0, 1]$. In other words, a component instance of `B1` might have any number of port instances of types `modeBegin` from 0 until k. `B1` also has a `modeEnd` port type with cardinality k. `mib` stands for "mode i begin" and indicates that an action that is enabled in mode i has begun its execution. `mie` stands for "mode i end" and indicates that an action that is enabled in mode i has finished its execution. Each `inMode` port instance of the `Mode Manager` must be connected with the corresponding `modeBegin` port instances of all `B2` components through an n-ary connector.

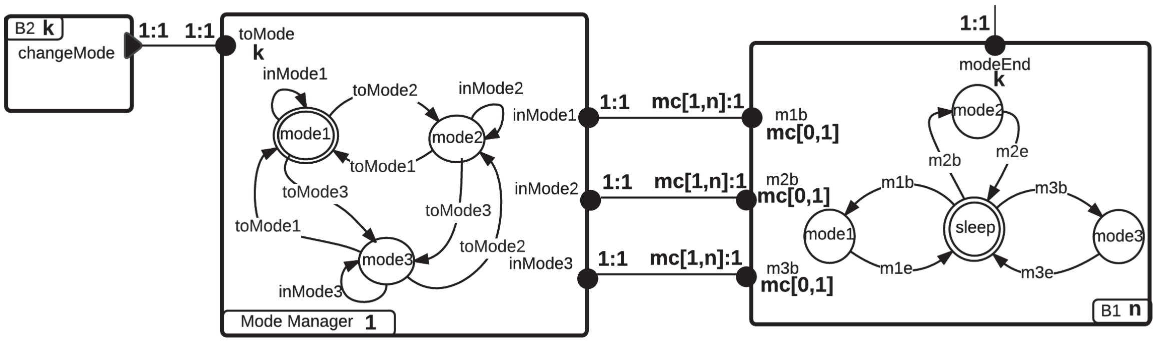


Figure 6.8 – Mode management style with architecture diagrams (component behaviour is shown for k=3)

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

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

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

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

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

Buffer management style

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

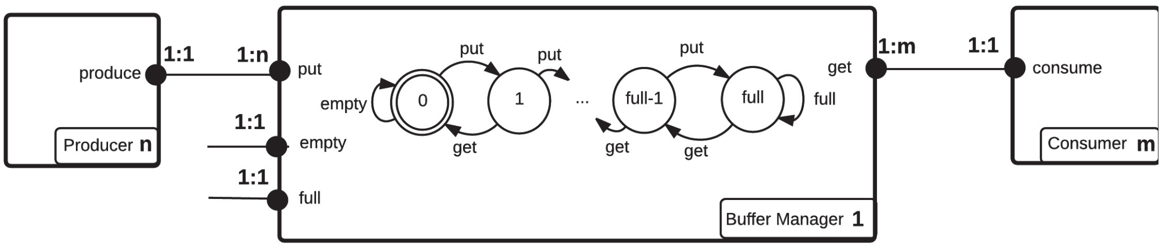


Figure 6.9 – Buffer management style with architecture diagrams

- **Assumed property of operands:** no assumptions
- **Characteristic property of architecture** is the conjunction of the following properties: 1)‘*data can be stored to the buffer only if the buffer is not full*’; 2)‘*data can be retrieved from the buffer only if the buffer is not full*’

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

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

Event monitoring style

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

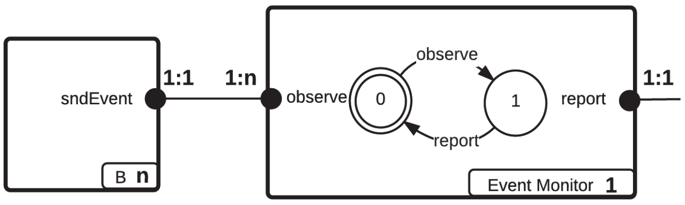


Figure 6.10 – Event monitoring style with architecture diagrams

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

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

Priority management style

The Priority management style enforces a priority protocol on the set of actions of n components. It consists of a single coordinator component of type `Priority Manager` and n parameter components of type `B`. The `Priority Manager` checks first whether the action with the highest priority can be executed. If this action is enabled, the `Priority Manager` executes this action and returns to its initial state. If this action is not enabled, the `Priority Manager` checks whether the action with the second highest priority can be executed. If this action is enabled, the `Priority Manager` executes this action and returns to its initial state. If this action is not enabled, it checks whether the action with the third highest priority can be executed, etc. The cardinalities of the `action` and `noAction` port types are n. The cardinalities of all other port types are 1. The multiplicity and degree constraints require that each `B` component instance be connected to the `Priority manager` component instance through binary connectors.

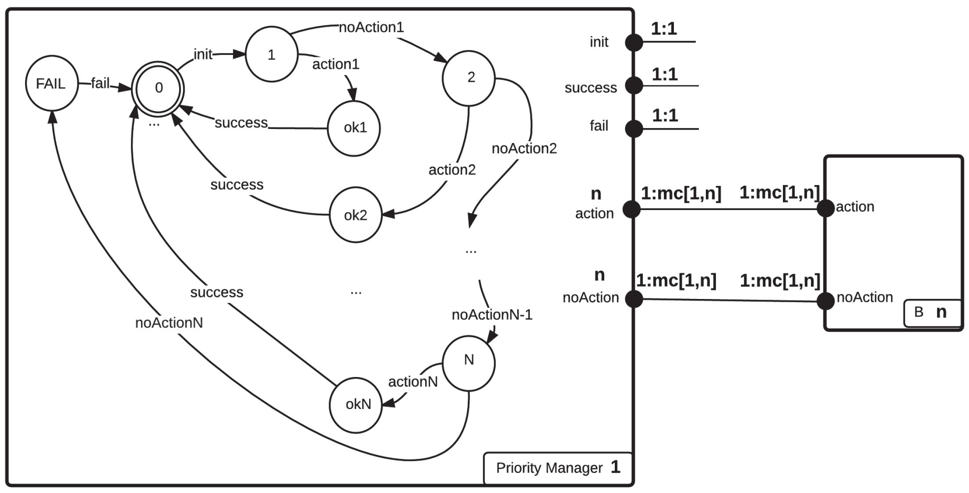


Figure 6.11 – Priority management style with architecture diagrams

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

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

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

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

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

Table 6.1 – CubETH: Representative requirements for CDMS status and HK_PL

ID	Description
CDMS-007	The CDMS shall periodically reset both the internal and external watchdogs and contact the EPS subsystem with a “heartbeat".
HK-001	The CDMS shall have a Housekeeping activity dedicated to each subsystem.
HK-003	When line-of-sight communication is possible, housekeeping information shall be transmitted through the COM subsystem.
HK-004	When line-of-sight communication is not possible, housekeeping information shall be written to the non-volatile flash memory.
HK-005	A Housekeeping subsystem shall have the following states: NOMINAL, ANOMALY and CRITICAL_FAILURE.

6.1.2 Architecture application and composition examples

We illustrate the architecture-based approach on the `CDMS status`, `MESSAGE_LIBRARY` and `HK PL` components. In particular, we present the application of Action flow, Mode management, Client-Server and Failure monitoring architectures to discharge a subset of the CubETH functional requirements presented in Table 6.1. The set of requirements of the CubETH case-study is presented in Table A.1 We additionally present the result of the composition of Client-Server and Mode management architectures.

Application of Action flow architecture

Requirement CDMS-007, presented in Table 6.1, describes the functionality of `CDMS status`. The corresponding BIP model is shown in Figure 6.12. `Watchdog reset` is an operand component, which is responsible for resetting the internal and external watchdogs. `CDMS status ACTION FLOW` is the coordinator of the architecture applied on `Watchdog reset` that imposes the following order of actions: 1) reset internal watchdog; 2) reset external watchdog; 3) send heartbeat and 4) receive result.

Application of Client-Server architecture

Requirements HK-001 and HK-003, presented in Table 6.1, suggest the application of the Client-Server architecture on the `HK PL`, `HK CDMS`, `HK EPS` and `HK COM` housekeeping compounds (Figure 6.2). The four housekeeping compounds are the clients of the architecture. In Figure 6.13a, we show how Client-Server is applied on the `HK PL process` component, which is a subcomponent of `HK PL`. The `HK PL process` component uses the `composeMessage` and `decodeMessage` C/C++ functions of the `MESSAGE_LIBRARY` component to encode and decode information transmitted to and from the `COM` subsystem. Thus, the `MESSAGE_LIBRARY`

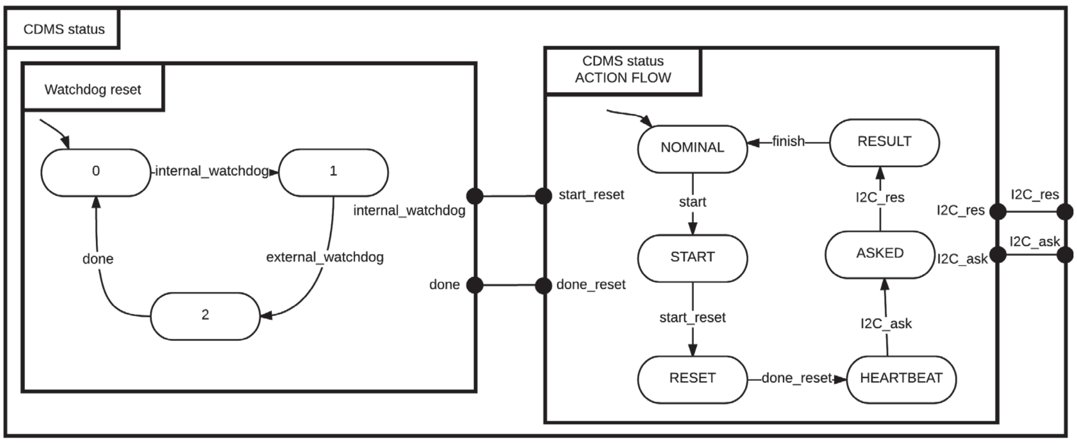
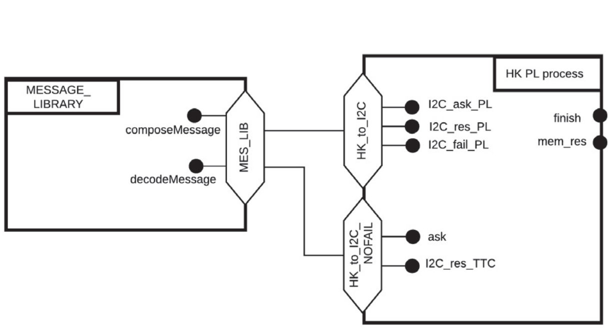
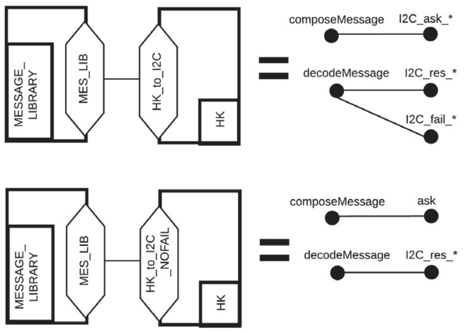


Figure 6.12 – Application of Action flow architecture



(a) Architecture application



(b) Hexagons of Figure 6.13a

Figure 6.13 – Application of Client-Server architecture

is a server used by the `HK PL process` client. To enhance readability of figures in Figure 6.13a, we use hexagons to group interaction patterns of components. The meaning of these hexagons is explained in Figure 6.13b.

Application of Failure monitoring architecture

Requirement HK-005, presented in Table 6.1, suggests the application of the Failure monitoring architecture as shown in Figure 6.14. The BIP model comprises the `HK PL process` operand and the `HK PL FAILURE MONITORING` coordinator. The `success` port of `HK PL FAILURE MONITORING` is connected with the `mem_res` and `I2C_res_TTC` ports of `HK PL process`. The `failure` port of `HK PL FAILURE MONITORING` is connected with the `I2C_fail_PL` port of `HK PL process`. The `HK PL process` component executes 6 actions in the following order: 1) start procedure; 2) ask `Payload` for engineering data; 3) receive result from `Payload` or (in case of fail) abort; 4) if line of sight communication is possible send data to `COM`, otherwise make a write request to the memory; 5) depending on action 4 either receive `COM` result or memory result and 6) finish procedure.

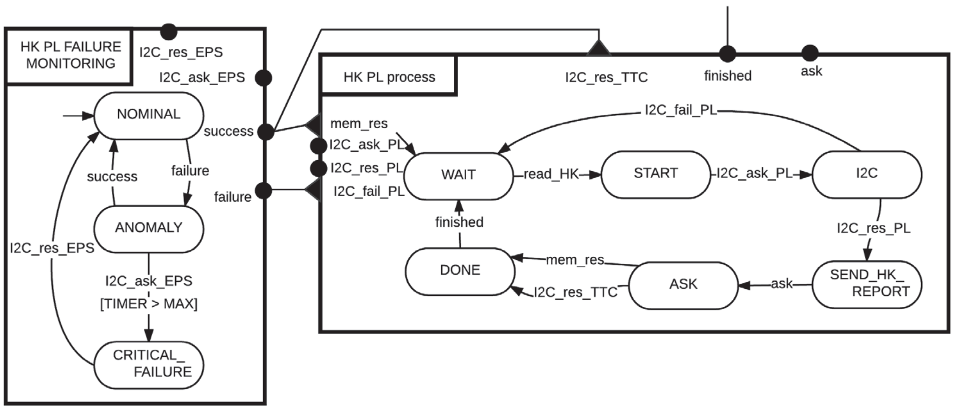


Figure 6.14 – Application of Failure monitoring architecture

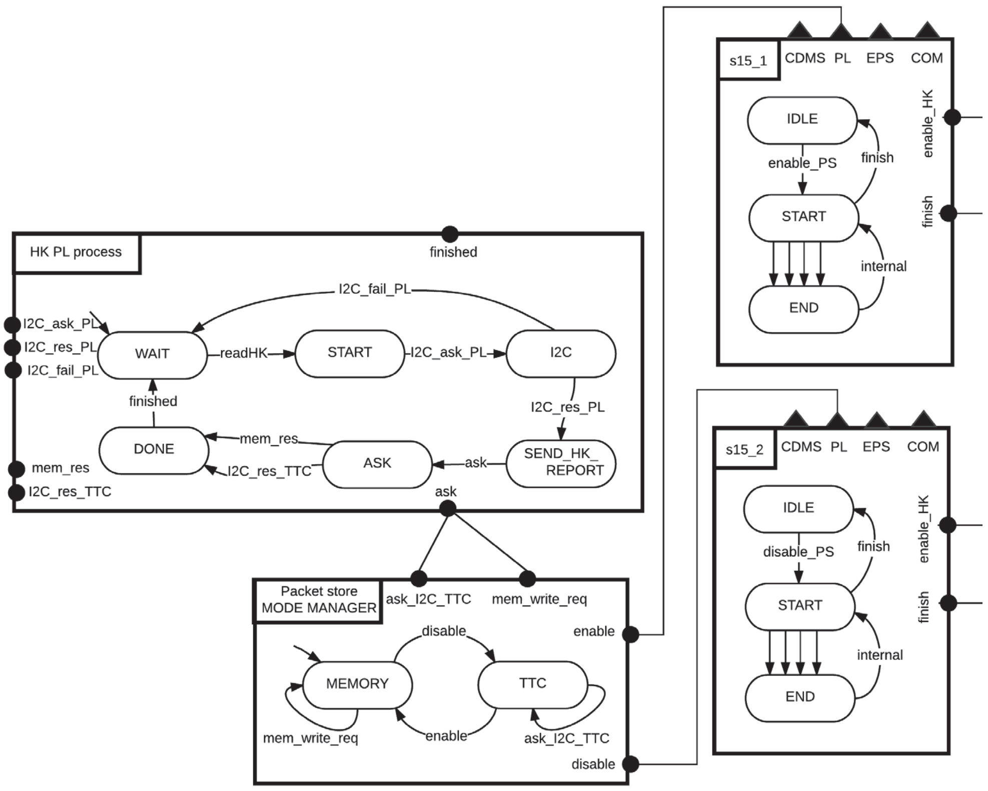


Figure 6.15 – Application of Mode management architecture

Application of Mode management architecture

Requirements HK-003 and HK-004, presented in Table 6.1, suggest the application of a Mode management architecture with two modes: 1) `TTC` mode, in which line of sight communication is possible and 2) `MEMORY` mode, in which line of sight communication is not possible. The corresponding BIP model, shown in Figure 6.15, comprises the `HK PL process`, `s15_1` and `s15_2` operands and the `Packet store MODE MANAGER` coordinator. During `NOMINAL` operation, the `Payload` subsystem is contacted to retrieve engineering data. Depending on the mode of `Packet store MODE MANAGER`, that data is then sent to the non-volatile memory, i.e. `mem_write_req` transition, or directly to the `COM` subsystem, i.e. `ask_I2C_TTC` transition. The mode of `Packet store MODE MANAGER` is triggered by the `s15_1`, `s15_2` services.

Composition of architectures

The architecture composition was formally defined in Definition 2.4.4. Here, we provide only an illustrative example. Combined application of architectures to a common set of operand components results in merging the connectors that involve ports used by several architectures. For instance, Figure 6.16 shows the composition of Client-Server and Mode management architectures. The `HK PL process` component is a sub-component of `HK PL`. The application of the Client-Server architecture (Figure 6.13) connects its port `ask` with the port `composeMessage` of `MESSAGE_LIBRARY` through the `MES_LIB-HK_to_I2C` interface with a binary connector. Similarly, the application of the Mode management architecture (Figure 6.15) connects the same port with the port `ask_I2C_TTC` of `Packet store MODE MANAGER` with another binary connector. The composition of the two architectures results in the two connectors being merged into the ternary connector `ask-ask_I2C_TTC-composeMessage` (Figure 6.16).

6.1.3 Model verification

Recall (Section 2.4) that safety properties imposed by architectures are preserved by architecture composition [6]. Thus, all properties that we have associated to the CubETH requirements are satisfied *by construction* by the model of the case study, which is included in the appendix (Section A.3).

Architectures enforce properties by restricting the joint behaviour of the operand components. Therefore, combined application of architectures can generate deadlocks. We have used the `D-Finder` tool [11] to verify deadlock-freedom of the case study model. `D-Finder` applies compositional verification on BIP models by over-approximating the set of reachable states, which allows it to analyse very large models. The tool is sound, but incomplete: due to the above mentioned over-approximation it can produce false positives, i.e. potential deadlock states that are unreachable in the concrete system. However, our case study model was shown to be deadlock-free without any potential deadlocks. Thus, no additional reachability

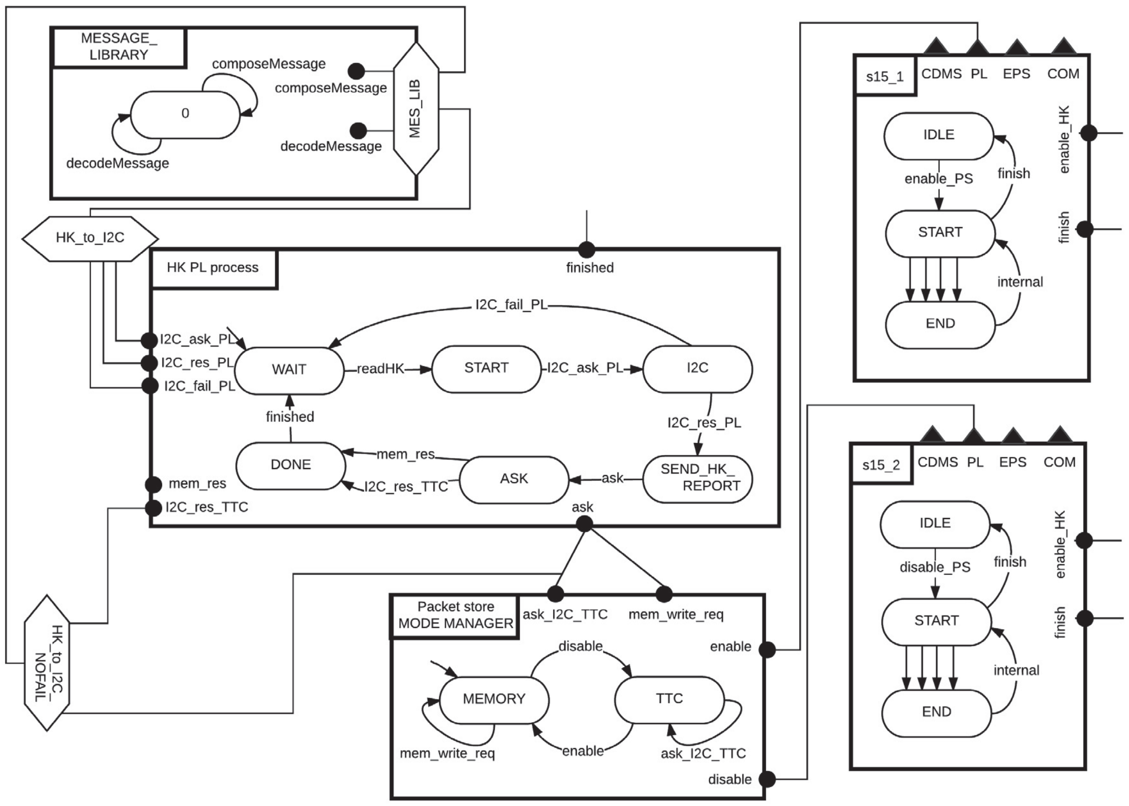


Figure 6.16 – Composition of Client-Server and Mode management architectures

analysis was needed.

6.1.4 Validation of the approach

The key advantage of our architecture-based approach is that the burden of verification is shifted from the final design to architectures, which are considerably smaller in size and can be reused. In particular, all the architecture styles that we have identified for the case study are simple. Their correctness—enforcing the characteristic properties—can be easily proved by inspection of the coordinator behaviour. However, in order to increase the confidence in our approach, we have conducted additional verification, using `nuXmv` to verify that the characteristic properties of the architectures are indeed satisfied. We used the `BIP-to-NuSMV` tool[1] to translate our BIP models into NuSMV—the `nuXmv` input language [16].

Verification of the complete model with `nuXmv` did not succeed, running out of memory after four days of execution. Thus, we repeated the procedure (BIP-to-NuSMV translation and verification using `nuXmv`) on individual sub-systems. All connectors that crossed sub-system boundaries were replaced by their corresponding sub-connectors. This introduces additional interactions, hence, also additional execution branches. Since no priorities are used in the

[1]http://risd.epfl.ch/bip2nusmv

Table 6.2 – CubETH: Statistics of models and verification

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

RSS = Reachable State Space

model, this modification does not suppress any existing behaviour. Notice that the CTL properties enforced by the presented architecture styles use only universal quantification (`A`) over execution branches. Hence, the above approach is a sound abstraction, i.e. the fact that the properties were shown to hold in the sub-systems immediately entails that they also hold in the complete model. Table 6.2 presents the complexity measures of the verification, which was carried out on an Intel Core i7 at 3.50GHz with 16GB of RAM. Notice that component count in sub-systems adds up to more than 49, because some components contribute to several sub-systems.

6.2 The Sentinel 3 case study

Sentinel 3 is a LEO Earth Observation satellite [94]. The Sentinel 3 mission is part of the Earth watch line of missions within the ESA's Living Planet Program. The goal of its mission is mainly sea-surface topography, measurement of the sea-surface temperature and the ocean colour characteristics.

The Sentinel 3 case study was based on an extraction of the complete set of requirements of the Sentinel 3 on-board software. Since the information of the case study is confidential, we do not provide the requirements and the resulting BIP model of the satellite. Nevertheless it is interesting to mention that in order to build the resulting model we reused the architecture styles presented in Section 6.1.1. The complete BIP model of Sentinel 3 has a small number of operands; it primarily consists of architectures generated from the architecture styles. The essential safety properties were obtained by construction through the architecture-based approach.

Next, we give a very high-level description of the Sentinel 3 on-board software. The software architecture of the on-board software is presented in Figure 6.17. It is composed of the

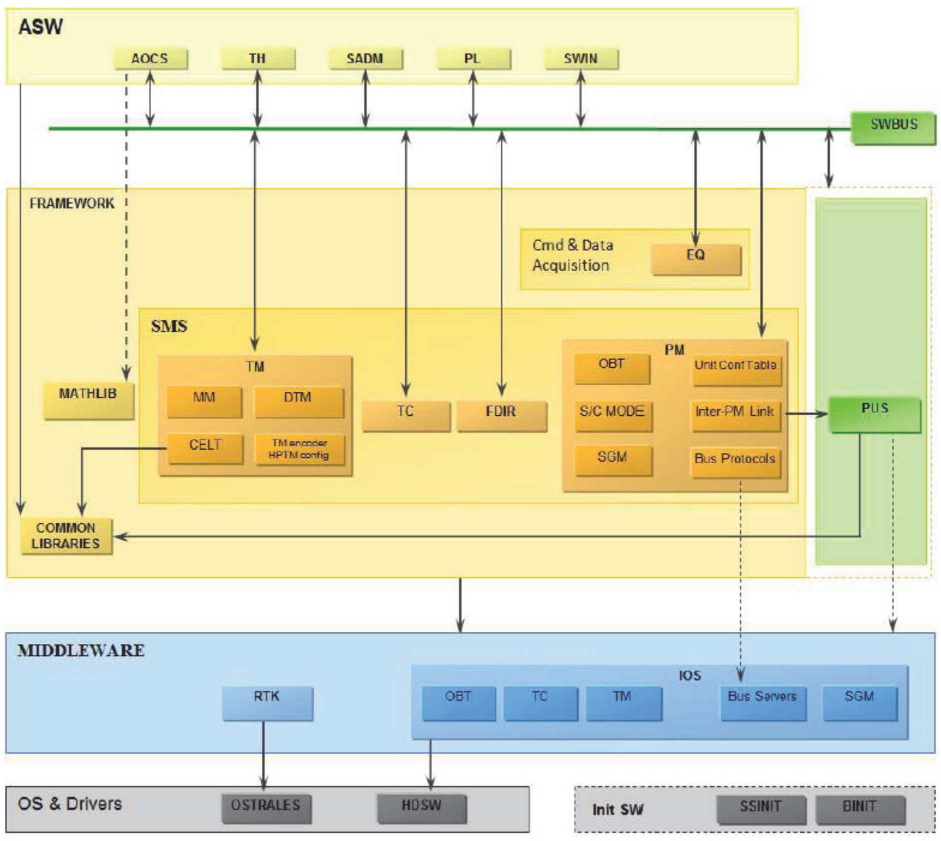


Figure 6.17 – Software architecture of Sentinel 3 [94]

following 6 main parts:

- the OS and Drivers layer, which comprises the Hardware Dependent Software (HDSW) and the Real-Time Operating System (RTOS);
- the Middleware, which is mainly composed by Input-Output Services (IOS);
- the Framework which comprises the System Management Software (SMS) and generic code libraries;
- the Packet Utilisation Service (PUS) library;
- the Software Bus (SWBUS), which allows the different components of the architecture to communicate;
- the Application Software (ASW), which consists of a set of components for subsystem management.

6.3 Summary

The architecture-based approach consists in the application of a number of architectures starting with a minimal set of atomic components. Each architecture enforces *by construction* a characteristic safety property on the joint behaviour of the operand components. The combined application of architectures is defined by an associative and commutative operator [6], which guarantees the preservation of the enforced properties. Since architectures enforce properties by restricting the joint behaviour of the operand components, combined application of architectures can lead to deadlocks. Thus, the final step of the design process consists in verifying the deadlock-freedom of the obtained model. The key advantage of this approach is that the burden of verification is shifted from the final design to architectures, which are considerably smaller in size and can be reused. This advantage is illustrated by our verification results: while model-checking of the complete model was inconclusive, verification of deadlock-freedom took only a very short time using the `D-Finder` tool.

The case studies serve as a feasibility proof for the use of architecture-based approach in satellite on-board software design. An additional contribution of the presented work is the identification and formal modelling—using architecture diagrams—of a taxonomy of architecture styles. Architecture styles represent recurring coordination patterns: those identified in the CubETH case study were also reused in the Sentinel 3 case study and can be further reused in other satellite on-board software.

7 Conclusion and future work

The main goal of this thesis was the development of an expressive and usable framework for the specification of architectures and architecture styles. We focused on providing languages, methodologies and tools that could be introduced in the system development process. Our work is part of a broader research program, which has been pursued for more than 15 years, investigating correct-by-construction approaches. The program aims at developing the BIP component framework for rigorous system design [89].

The following subsections summarize how our work is related to the three fundamental questions posed earlier in the introduction. In particular, Section 7.1 addresses the first question of "how does one model and implement architectures?", while Section 7.2 addresses the second and third questions of "how to guide software developers in their choice of architectures?" and "how does one specify architecture styles?".

7.1 Modelling and implementation of architectures

We presented interaction logics for modelling architectures. We studied a modelling methodology based on the first-order interaction logic for writing architecture specifications. The architecture specifications are sufficiently versatile to be applicable to a variety of components and do not rely on the characteristics of any specific execution platform. We have developed JavaBIP, relying on first-order interaction logic for the specification of the coordination architecture.

JavaBIP integrates architectures into mainstream software development and thus, raises the abstraction level and supports separation of concerns, e.g. decoupling between functionality and coordination. JavaBIP follows an exogenous approach to component coordination relying on existing APIs for the interaction with the controlled components. All JavaBIP models are defined separately from the functional code of the components. Furthermore, all input specifications are defined and modified without altering the functional code, which is not possible with traditional approaches. We presented experimental results that validate the effectiveness of JavaBIP for the coordination of software components. JavaBIP specifications are formally defined and thus, can be verifiable. Verification tools, e.g. DFinder and nuXmv, can be used to validate JavaBIP specifications, ensuring correctness of the designed systems. Additionally, JavaBIP coordination has been tested and validated in Connectivity Factory TM by Crossing-Tech S.A.

Future work

Future work comprises several directions. We are currently working on adding dynamicity in the JavaBIP framework in order to allow insertion and deletion of components at run-time. Furthermore, we consider creating a library of JavaBIP architecture specifications so that a software developer can pick a preferred coordination mechanism directly from the library. It would also be interesting to add a graphical interface for the specification of coordination mechanisms based on architecture diagrams.

Another possible extension concerns the implementation of JavaBIP. Currently, JavaBIP is centralised due to the implementation of the JavaBIP engine. Scalability of JavaBIP can be increased by combining existing techniques from the BIP framework for parallelising and distributing the JavaBIP engine. Another possible direction for increasing the efficiency of the JavaBIP engine is to add *incrementality* so that an interaction can be fired without waiting for all the other components to inform.

7.2 Modelling architecture styles

We studied two rigorous approaches for the specification of architecture styles, which are both applicable to a variety of architecture styles. We studied configuration logics, which are a powerful and expressive tool to characterize configurations between instances of typed components. We studied the properties as well as a sound and complete axiomatisation for the propositional configuration logic. Configuration logics are integrated in a unified semantic framework, which is equipped with a decision procedure for checking that a given architecture model meets given style requirements. We proposed three high-level extensions of the propositional configuration logic and showed their applicability in the specification of several architecture styles.

Furthermore, we studied architecture diagrams, which is a graphical language for architecture style specification rooted in rigorous semantics. Using architecture diagrams instead of purely logic-based specifications confers the usability advantages of graphical formalisms. We proposed methods to assist software developers to correctly specify architecture styles, generate architectures from a style and check whether a given architecture model meets given style requirements.

To guide developers in their choice of architectures, we grouped architectures that enforce the same characteristic properties and involve the same component types into families of architectures, i.e. architecture styles. In the case studies, presented in Section 6, we defined a taxonomy of architecture styles that is specific to on-board satellite software systems [74]. These case studies serve as a feasibility proof for the use of the architecture-based approach in on-board satellite software design and, more generally, in building systems that are correct-by-construction. Creating taxonomies and subsequently libraries of architecture styles promotes component re-use and thus, the development of systems in a cost-effective manner. Additionally, organising knowledge in the form of architecture styles facilitates the understanding of design decisions and makes communication more efficient.

Future work

From the specification perspective, we plan to incorporate hierarchically structured interactions, data transfer among the participating ports and mechanisms to express dynamic evolution of architectures. From the practical perspective, we plan on extending the synthesis procedure to incorporate index and general architecture diagrams, as well as implementing it using an SMT solver. We have already implemented the checking procedure for the consistency conditions of simple architecture diagrams using the Z3 SMT solver[1] and have integrated it in the Architecture Manipulation tool that automatically generates and composes architectures from given styles. We are going to extend the current implementation for interval architecture diagrams.

In the CubETH case study (Section 6.1) we defined a taxonomy of architecture styles from which architectures were generated and composed to develop the BIP model of the satellite. We plan on expanding our taxonomy of architecture styles and applying the architecture-based approach to other application domains.

[1]https://z3.codeplex.com/

A Appendix

A.1 JavaBIP use case: Publish-Subscribe server

In this section, we present a JavaBIP implementation of the Publish-Subscribe server. The server manages a number of different topics to which the clients can subscribe, unsubscribe and publish messages. To that end, clients send commands, which are handled by the server in a concurrent fashion.

Consider the model in Figure A.1. The solid connectors between ports denote strong synchronizations, i.e., the execution of the corresponding transitions must be synchronized. In all synchronizations, data are exchanged between components. The dotted arrows from transitions to ports represent asynchronous communication realized by generating events corresponding to the spontaneous transitions of the receiving components.

The server consists of components of the six types shown in Figure A.1. For each client, there is a dedicated TCPReader, responsible for receiving commands from the client. Additionally, for each client there is dedicated a ClientProxy, responsible for receiving acknowledgements that the client has been added or removed from a topic and messages published from other clients registered in the same topic. Upon reception by a TCPReader, each command is forwarded to the unique CommandBuffer component through the synchronization of the `give` and `put` enforceable transitions. The guard `commandExists` of the TCPReader is used to check whether it has received a new command and the guard `notFull` of the CommandBuffer is used to check whether the buffer is not full before receiving a new command (Figure A.2:

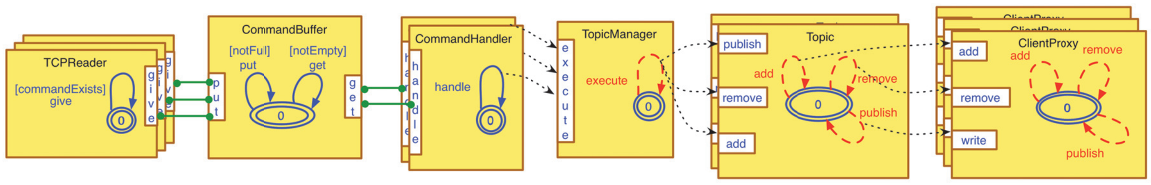


Figure A.1 – JavaBIP models of a Publish-Subscribe server

```
 1 @Ports({
 2   @Port(name="put", type=PortType.enforceable),
 3   @Port(name="get", type=PortType.enforceable)
 4 })
 5
 6 @ComponentType(initial="0", name="CommandBuffer")
 7 public class CommandBuffer {
 8   private LinkedList<Command> commandList;
 9   private int bufferSize;
10
11   public CommandBuffer(int bufferSize){
12     this.bufferSize = bufferSize;
13     this.commandList = new LinkedList<Command>();
14   }
15
16   @Transition(name="get", source="0", target="0", guard="notEmpty")
17   public void get() {
18     commandList.remove();
19   }
20
21   @Guard(name="notEmpty")
22   public boolean notEmpty() { return !commandList.isEmpty(); }
23
24   @Transition(name="put", source="0", target="0", guard="notFull")
25   public void put(@Data(name="input") Command cmd) {
26     commandList.add(cmd);
27   }
28
29   @Guard(name="notFull")
30   public boolean notFull() { return commandList.size() < bufferSize; }
31
32   @Data(name="command")
33   public Command getNextCommand() { return commandList.get(0); }
34 }
```

Figure A.2 – Annotations for the CommandBuffer component type

lines 29–30). If both guards evaluate to *true*, the command is transferred as data to the CommandBuffer (Figure A.2: line 25).

The CommandBuffer is a passive component: the responsibility for retrieving commands from the CommandBuffer belongs to CommandHandlers. This happens through the synchronization of the `handle` and `get` enforceable transitions, when the `notEmpty` guard evaluates to *true* (Figure A.2: lines 21–22). There can be arbitrarily many CommandHandlers that are concurrently handling commands. The CommandHandlers asynchronously forward commands to the TopicManager, by generating the event associated to the `execute` spontaneous transition of the TopicManager (Figure A.3: lines 2 & 13). Notice that the `@Data` annotation is used to define input data necessary for the processing of spontaneous events (Figure A.3: line 14). This mechanism allows CommandHandlers to send the commands as data to the TopicManager, in a manner equivalent to asynchronous message passing.

Depending on the type of the command (Figure A.3: lines 16, 20, 24), the TopicManager asynchronously triggers one of the `add`, `remove` or `publish` transitions of the corresponding Topic (Figure A.3: lines 18, 22, 26). The Topic executes the commands and triggers the corresponding transitions of a ClientProxy to either send an acknowledgment to the client

```
@Ports({
 @Port(name="execute", type=PortType.spontaneous)
})

@ComponentType(initial="0", name="TopicManager")
public class TopicManager {
  private HashMap<String, BIPActor> topics;

  public TopicManager(HashMap<String, BIPActor> topics) {
    this.topics = topics;
  }

  @Transition(name="execute", source="0", target="0")
  public void execute(@Data(name="value") Command c) {
    switch (c.getId()) {
    case SUBSCRIBE:
      Topic topic = topics.get(c.getTopic());
      topic.add(c.getClient());      // Generate an "add" event in the topic
      break;
    case UNSUBSCRIBE:
      Topic topic = topics.get(c.getTopic());
      topic.remove(c.getClient()); // Generate a "remove" event in the topic
      break;
    case PUBLISH:
      Topic topic = topics.get(c.getTopic());
      topic.publish(c.getClient(), c.getMessage());
      break;                        // Generate a "publish" event in the topic
    default:
      break;
    }
  }
}
```

Figure A.3 – Annotations for the TopicManager component type

in the case of the subscribe/unsubscribe commands or to distribute the message to all the subscribed clients of the topic in the case of a publish command. All transitions of TopicManager, Topic and ClientProxy components are spontaneous, i.e. they are executed asynchronously upon reception of the corresponding events.

An example of the above execution is the following: A TCPReader wants to forward the command `subscribe epfl`. This data transfer happens through the synchronization of the `give` transition of the TCPReader with the `put` transition of the CommandBuffer. A CommandHandler receives the command from the CommandBuffer through the synchronization of the `get` and `handle` transitions. Then, the CommandHandler forwards the command to the TopicManager. This triggers the `execute` spontaneous transition during which the client id is transferred to the `epfl` topic. This results in the asynchronous execution of the `add` transition of the `epfl` topic. During the execution of `add` the client id is stored and the name of the topic (epfl) is forwarded to the dedicated ClientProxy. The ClientProxy stores the topic name and writes an acknowledgment in the socket while executing the `add` transition in an asynchronous manner.

A.2 List of requirements of CubETH case study

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

HK-005	A Housekeeping subsystem shall have the following states: NOMINAL, ANOMALY, and CRITICAL_FAILURE.
HK-006	In NOMINAL state, the subsystem shall perform correctly.
HK-007	If a process failure occurs or if the engineering data are not correct the subsystem shall enter the ANOMALY state.
HK-008	After MAX[1] seconds in ANOMALY, the subsystem shall enter the CRITICAL_FAILURE state.
HK-009	In CRITICAL_FAILURE state, the subsystem shall contact the EPS and demand a restart of the malfunctioning subsystem.
HK-010	During NOMINAL operation the subsystem shall be contacted to retrieve engineering data.
I2C-001	A single user shall send one message at a time.
I2C-002	I2C_sat shall implement the I^2C protocol.
Log-001	Every time a hardware error is produced, it shall be stored in a memory region in the RAM.
Log-002	The dedicated RAM region shall be read and written atomically.
Mem-001	The Central Software System shall have a dedicated Flash Memory Manager activity for managing flash memory operations.
Mem-002	Flash memory shall be read and written atomically.
Mem-003	Flash Memory Manager shall return the SUCCESS or FAIL status for each requested operation.
Mem-004	Upon a read request, the Flash Memory Manager shall read the data from the flash memory and perform the Circular Redundancy Check (CRC).
Mem-005	If CRC fails, the Flash Memory Manager shall reread the data from the flash memory.
Mem-006	For the same read request, the number of attempts by the Flash Memory Manager to read data from the flash memory shall have a value not larger than the parameter MAX_FM_READS.
Mem-007	If the number of attempts by the Flash Memory Manager to read data from the flash memory exceeds MAX_FM_READS, the read operation shall be abandoned and a failure shall be reported.

Table A.1 – CubETH: Complete list of requirements

A.3 BIP model of CubETH case study

In this section, we present the full componentization of the case study. The complete model consists of 22 operand components and 27 coordinator components. We omit the presentation of the `CDMS status` compound, which was already presented in Section 6.1.2. We have

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

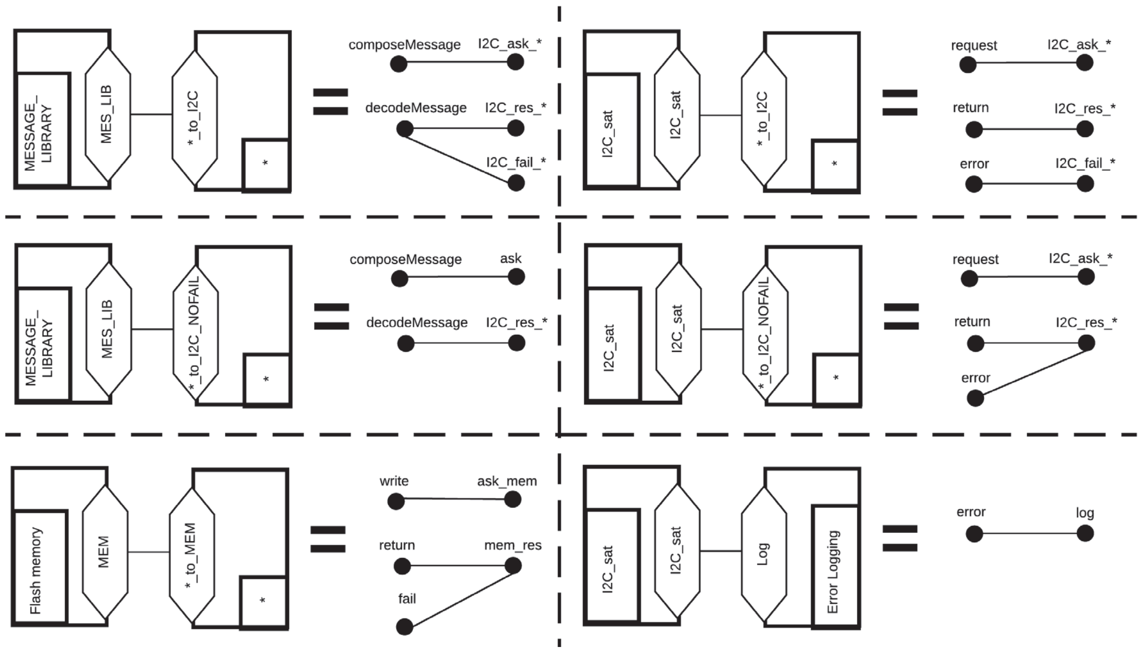
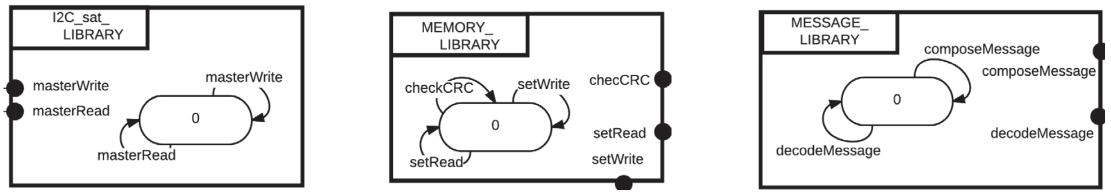


Figure A.4 – Meaning of connections between hexagons in Figure 6.2



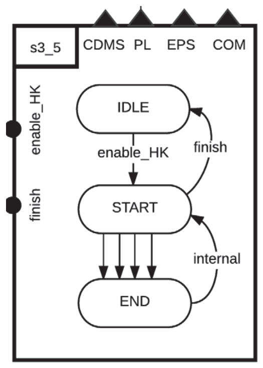
(a) The I2C_sat library component (b) The memory library component (c) The message library component

Figure A.5 – Library components

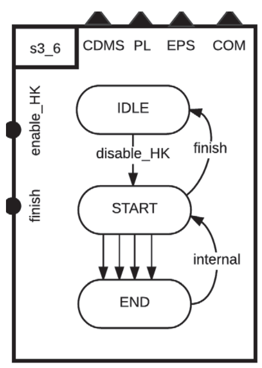
shown the high level interaction model of the case study in Figure 6.2. As discussed earlier, we use hexagons to group interaction patterns of components. The meaning of connections between the hexagons in Figure 6.2 is explained in Figure A.4.

Libraries

The model includes three library components that contain helper C/C++ functions. Figure A.5 shows the `I2C_sat_LIBRARY`, the `MEMORY_LIBRARY` and the `MESSAGE_LIBRARY` components. All of them are operands. `I2C_sat_LIBRARY` contains functions that allow communication on the `I2C_sat` bus. `I2C_sat_LIBRARY` contains functions that allow writing, reading and checking the CRC in the non-volatile flash memory. `I2C_sat_LIBRARY` allows a user to compose a message that is going to be send over the `I2C_sat` bus and also to decode a message received from the `I2C_sat` bus.

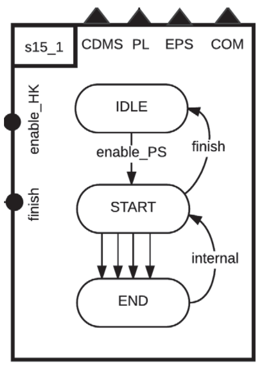


(a) The s3_5 component

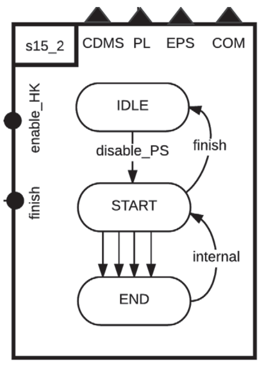


(b) The s3_6 component

Figure A.6 – Housekeeping report enable and disable



(a) The s15_1 component



(b) The s15_2 component

Figure A.7 – Packet storage enable and disable

Housekeeping report enable, housekeeping report disable

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

Packet storage enable and packet storage disable

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

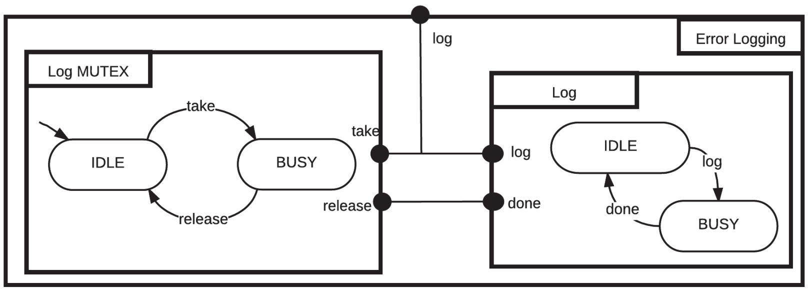


Figure A.8 – The error logging compound

Error logging

The `Error logging` compound, shown in Figure A.8, protects a specific RAM region that is accessible by many users. Every time a hardware error is produced, the error is stored in an array managed by this compound. No two users can log an error at the same time. It is composed of an operand component `Log` and a Mutual exclusion coordinator `Log MUTEX`.

Payload

The `Payload` (PL) compound, shown in Figure A.9, is in charge of payload operations. It is composed of the following subcomponents: 1) `s128_1` which adds a scenario to the payload board; 2) `s128_4` which executes a scenario telecommand; 3) `128_5` which aborts a payload operation; 4) `data_transfer` which transfers data from the payload to the non-volatile memory; 5) `status_verification` which checks the advancement of the payload board internal algorithms and 6) and a `PL mode management` coordinator.

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

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

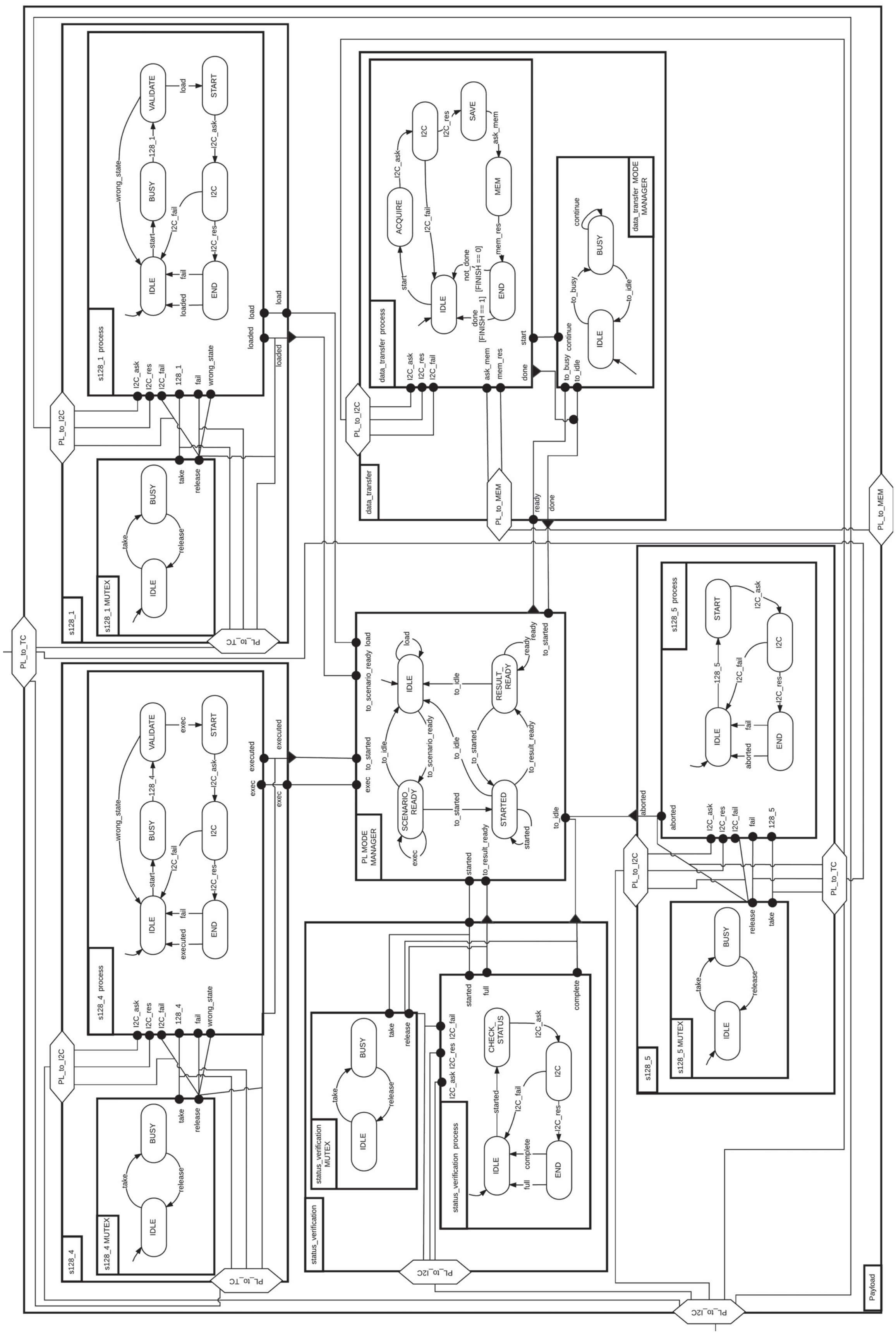


Figure A.9 – The Payload compound

Housekeeping subsystems

There are three Housekeeping subsystems, namely HK PL, shown in Figure A.10, HK EPS, shown in Figure A.11, and HK COM, shown in Figure A.12, that are in charged of recovering engineering from the PL, EPS, COM subsystems of the satellite, respectively. They are composed of the same subcomponents: 1) a HK MUTEX coordinator; 2) an HK process operand; 3) an HK MODE MANAGER coordinator; 4) an Packet store MODE MANAGER coordinator and 5) a HK FAILURE MONITORING coordinator.

The HK MUTEX component ensures mutual exclusion on the access of the subsystem. During NOMINAL operation, a subsystem is contacted to retrieve engineering data. That data is then sent to the non-volatile memory or directly to the COM subsystem, depending on the Packet store MODE MANAGER coordinator. The HK MODE MANAGER inhibits the HK process.

Internal Housekeeping

The HK CDMS compound, shown in Figure A.13, is internal to the CDMS subsystem. It is very similar to the rest of Housekeeping subsystems. The difference is that the Housekeeping data of the HK CDMS is not retrieved through the I2C bus but through internal processes (e.g. GPIO and state registers). The Failure monitoring coordinator is also removed because the EPS subsystem directly monitors the HK CDMS through the heartbeats.

I2C_sat

The I2C_sat compound, shown in Figure A.14, implements the I2C_sat bus protocol. The request transition is enabled as soon as a user wants to send a message through the I2C bus. It consists of the I2C_sat read operand, the I2C_sat MODE MANAGEMENT coordinator and the I2C_sat write operand. Mutual exclusion is ensured by the fact that this compound is the only one implementing the use of the I2C peripheral. Thus, once a request is issued no other user can access the bus until it has retuned to the IDLE state of I2C_sat write.

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

Flash memory management

The reading and writing procedures to the external non-volatile NOR flash memory are represented by the compound shown in Figure A.15. The memory device can be read and written through the write and read transitions. The Flash Memory MUTEX compound ensures that the memory is accessed by the users atomically. It consists of two operands (Flash Memory Write and Flash Memory Read), two Mode Manager coordinators and a Mutual exclusion coordinator.

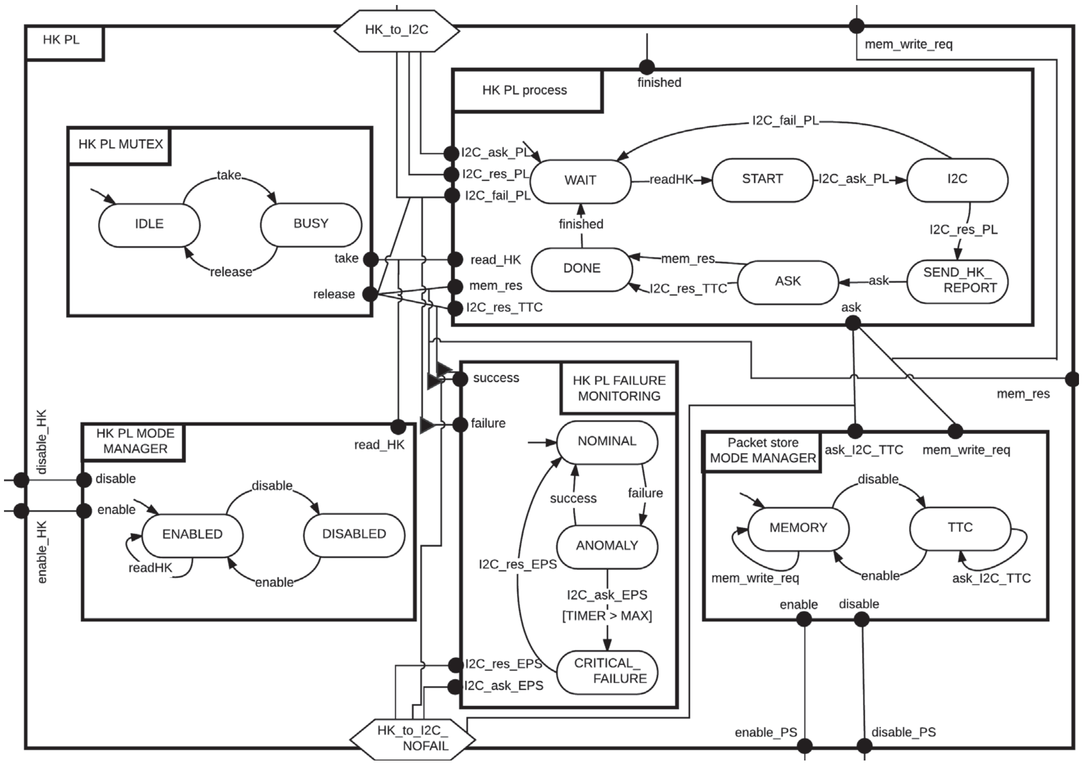


Figure A.10 – The Housekeeping Payload compound

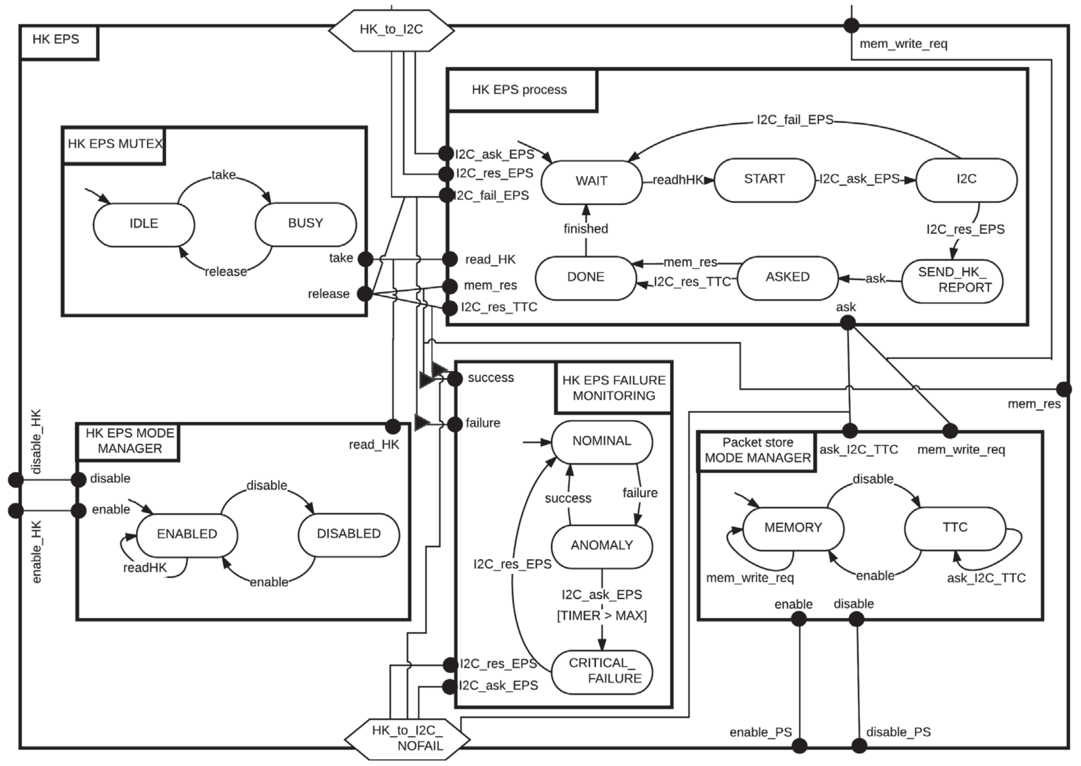


Figure A.11 – The Housekeeping EPS compound

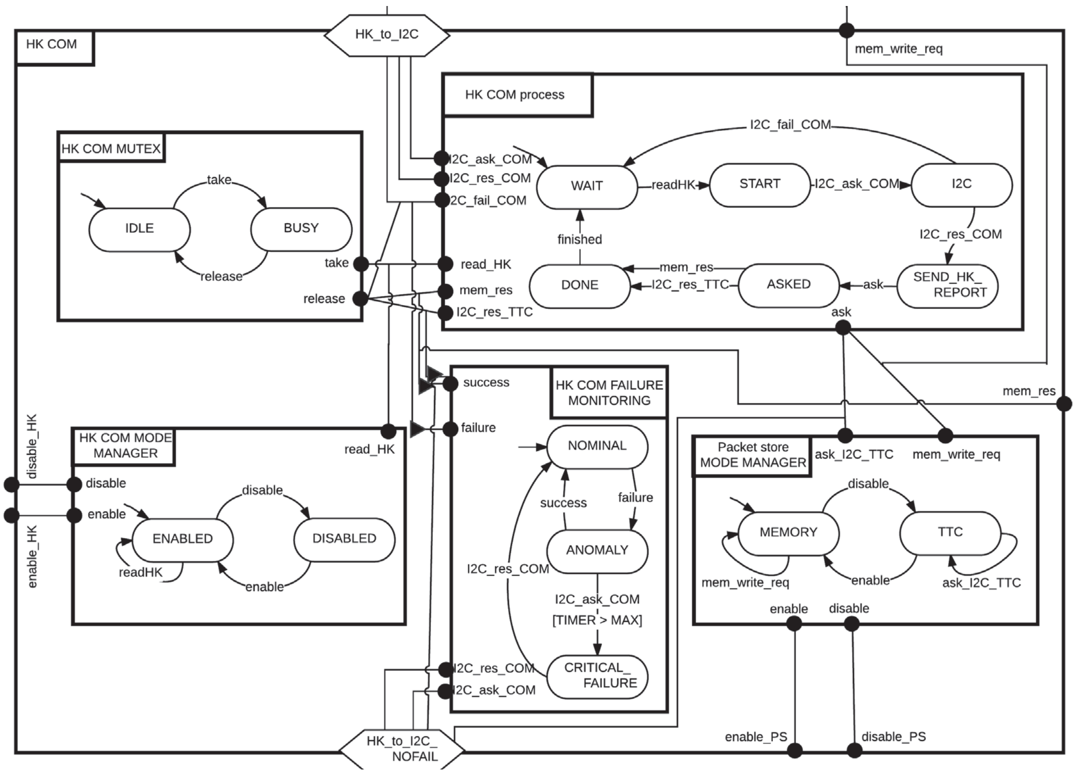


Figure A.12 – The Housekeeping COM compound

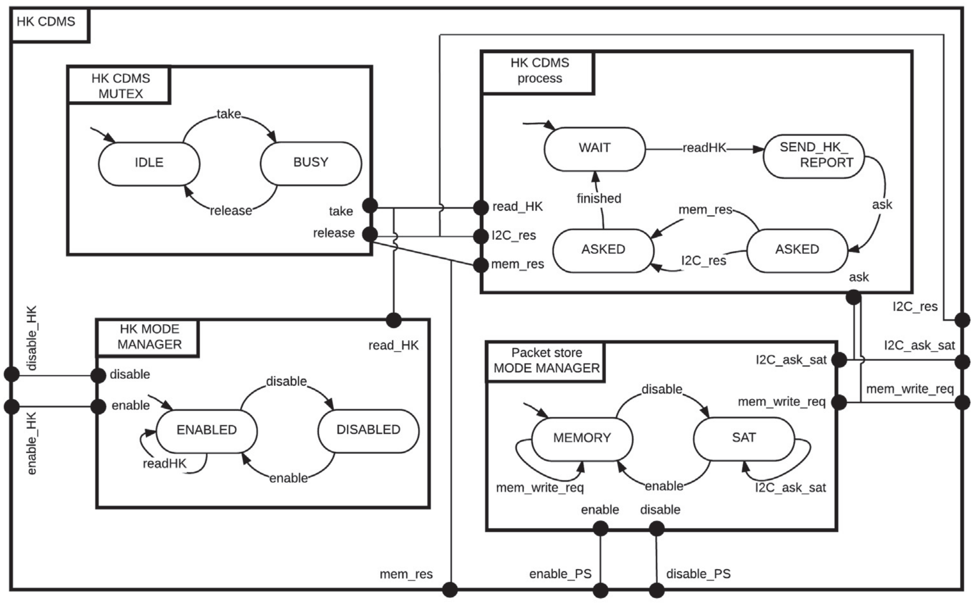


Figure A.13 – The Internal Housekeeping compound

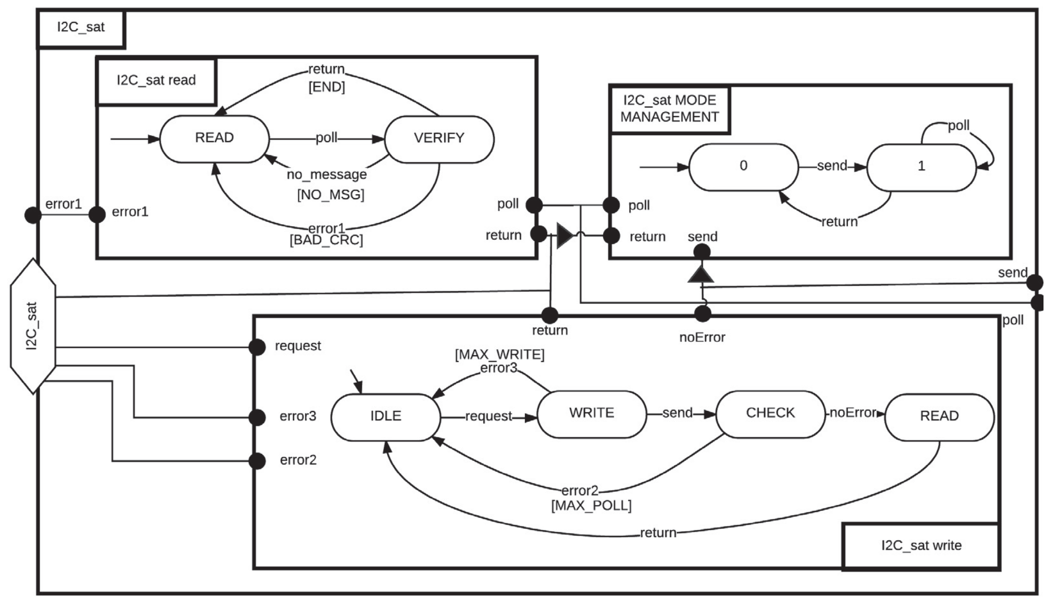


Figure A.14 – The I2C_sat compound

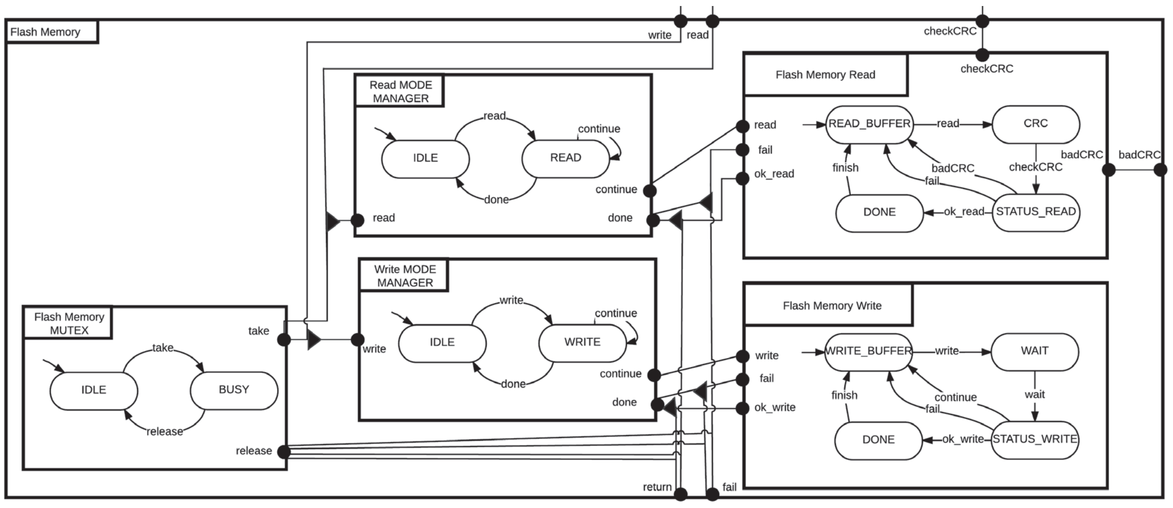


Figure A.15 – The flash memory compound

After a write request, `Flash Memory Write` takes the buffer prepared by the user and starts writing it in the memory. From `WAIT` to `STATUS_WRITE` there is an internal transition which adds a delay for a certain period of time. This is the minimum time to wait for the memory device to effectively write on the buffer. If the buffer is bigger than the write sector, then this procedure continues (synchronization of `write` with the `continue` of the `Write MODE MANAGER`). If there is no more data to write and the procedure was successful, the `ok_write` transition is fired and the coordinator enters the `DONE state`. The internal `finish` transition leads back to the `WRITE_BUFFER` state. If the memory does not perform as expected, e.g. internal timeout or error occurs, the writing procedure is aborted with the `fail` transition.

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

Bibliography

[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[2] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.

[3] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *OOPSLA '09*, pages 1015–1022, New York, NY, USA, 2009. ACM.

[4] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the 16th international conference on Software engineering*, pages 71–80. IEEE Computer Society Press, 1994.

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

[6] Paul Attie, Eduard Baranov, Simon Bliudze, Mohamad Jaber, and Joseph Sifakis. A general framework for architecture composability. In *Proceedings of the 12th International Conference on Software Engineering and Formal Methods (SEFM 2014)*, number 8702 in LNCS, pages 128–143. Springer, 2014.

[7] Paul Attie, Eduard Baranov, Simon Bliudze, Mohamad Jaber, and Joseph Sifakis. A general framework for architecture composability. *Formal Aspects of Computing*, 28(2):207–231, 2016.

[8] Krishnakumar Balasubramanian, Aniruddha Gokhale, Gabor Karsai, Janos Sztipanovits, and Sandeep Neema. Developing applications using model-driven design environments. *Computer*, 39(2):33–40, 2006.

[9] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the BIP framework. *Software, IEEE*, 28(3):41–48, 2011.

[10] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. DFinder: A tool for compositional deadlock detection and verification. In *Computer Aided Verification*, pages 614–619. Springer, 2009.

[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, pages 453–458, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] BIP. `http://www-verimag.imag.fr/~async/ BIP/bip.html`

[13] Jan Olaf Blech. Ensuring OSGi component based properties at runtime with behavioral types. In *MoDeVVa@ MoDELS*, pages 51–60. Citeseer, 2013.

[14] Jan Olaf Blech. Towards a framework for behavioral specifications of OSGi components. formal engineering approaches to software components and architectures. *Electronic Proceedings in Theoretical Computer Science*, 2013.

[15] Jan Olaf Blech and Peter Herrmann. Behavioral types for component-based development of cyber-physical systems. In *Software Engineering and Formal Methods*, pages 43–52. Springer, 2015.

[16] Simon Bliudze, Alessandro Cimatti, Mohamad Jaber, Sergio Mover, Marco Roveri, Wajeb Saab, and Qiang Wang. Formal verification of infinite-state BIP models. In *International Symposium on Automated Technology for Verification and Analysis*, pages 326–343. Springer, 2015.

[17] Simon Bliudze, Anastasia Mavridou, Radoslaw Szymanek, and Alina Zolotukhina. Coordination of software components with BIP: Application to OSGi. In *Proceedings of the 6th International Workshop on Modeling in Software Engineering*, MiSE 2014, pages 25–30, New York, NY, USA, 2014. ACM.

[18] Simon Bliudze, Anastasia Mavridou, Radoslaw Szymanek, and Alina Zolotukhina. Exogenous Coordination of Concurrent Software Components with JavaBIP. *Submitted to Software - Practice and experience. Under review.*, 2016.

[19] Simon Bliudze and Joseph Sifakis. The algebra of connectors — Structuring interaction in BIP. In *Proc. of the EMSOFT'07*, pages 11–20. ACM SigBED, October 2007.

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

[21] Simon Bliudze and Joseph Sifakis. Causal semantics for the algebra of connectors. *Formal Methods in System Design*, 36(2):167–194, June 2010.

[22] Simon Bliudze and Joseph Sifakis. Synthesizing glue operators from glue constraints for the construction of component-based systems. In Sven Apel and Ethan Jackson, editors, 10th *International Conference on Software Composition*, volume 6708 of *LNCS*, pages 51–67. Springer, 2011.

[23] 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*, pages 169–178. ACM, 2014.

[24] Grady Booch, James Rumbaugh, and Ivar Jacobson. The unified modeling language user guide. *Addison-Welsley Longman Inc*, 1999.

[25] Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis. Modeling dynamic architectures using Dy-BIP. In Thomas Gschwind, Flavio Paoli, Volker Gruhn, and Matthias Book, editors, *Software Composition*, volume 7306 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2012.

[26] Marco Bozzano, Alessandro Cimatti, and Cristian Mattarei. Automated analysis of reliability architectures. In *Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on*, pages 198–207. IEEE, 2013.

[27] Marco Bozzano and Adolfo Villafiorita. *Design and safety assessment of critical systems.* CRC press, 2010.

[28] Roberto Bruni, Alberto Lluch-Lafuente, Ugo Montanari, and Emilio Tuosto. Style-based architectural reconfigurations. *Bulletin of the EATCS*, 94:161–180, 2008.

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

[30] Vincent Cavé, Zoran Budimlić, and Vivek Sarkar. Comparing the usability of library vs. language approaches to task parallelism. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 9:1–9:6. ACM, 2010.

[31] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting software architectures: views and beyond.* Pearson Education, 2002.

[32] Daniel D. Corkill. Blackboard systems. *AI expert*, 6(9):40–47, 1991.

[33] Ivica Crnkovic and Magnus Peter Henrik Larsson. *Building reliable component-based software systems.* Artech House, 2002.

[34] Carlos E Cuesta, Pablo de la Fuente, Manuel Barrio-Solórzano, and Encarnación Beato. Coordination in a reflective architecture description language. In *International Conference on Coordination Languages and Models*, pages 141–148. Springer, 2002.

[35] Robert Daigneau. *Service design patterns: Fundamental design solutions for SOAP/WSDL and restful Web Services.* Addison-Wesley, 2011.

[36] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[37] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP 2004*, volume 3086 of *LNCS*, pages 465–490. Springer, 2004.

[38] Christopher Dragert, Juergen Dingel, and Karen Rudie. Generation of concurrency control code using discrete-event systems theory. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 146–157, New York, NY, USA, 2008. ACM.

[39] Hartmut Ehrig and Barbara Konig. Deriving bisimulation congruences in the DPO approach to graph rewriting. In *FoSSaCS*, volume 2987 of *LNCS*, pages 151–166. Springer, 2004.

[40] Johan Eker, Jörn W Janneck, Edward A Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[41] Gian Luigi Ferrari, Dan Hirsch, Ivan Lanese, Ugo Montanari, and Emilio Tuosto. Synchronised hyperedge replacement as a model for service oriented computing. In *Formal Methods for Components and Objects*, pages 22–43. Springer, 2006.

[42] David Garlan. Software architecture: a travelogue. In *Proceedings of the on Future of Software Engineering*, pages 29–39. ACM, 2014.

[43] David Garlan, Robert Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '97, pages 159–173. IBM Press, 1997.

[44] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, pages 1–39. World Scientific Publishing Company, 1993.

[45] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. In *Proceedings of the first workshop on Self-healing systems*, pages 33–38. ACM, 2002.

[46] Gregor Gößler and Joseph Sifakis. Priority systems. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 3188 of *Lecture Notes in Computer Science*, pages 314–329. Springer, 2003.

[47] David Harel and Bernhard Rumpe. Meaningful modeling: what's the semantics of "semantics" ? *Computer*, 37(10):64–72, 2004.

[48] Thomas A. Henzinger and Joseph Sifakis. *FM 2006: Formal Methods: 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006. Proceedings*, chapter The Embedded Systems Design Challenge, pages 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[49] Dan Hirsch, Paola Inverardi, and Ugo Montanari. Modeling software architectures and styles with graph grammars and constraint solving. In Patrick Donohoe, editor, *Software Architecture*, volume 12 of *IFIP*, pages 127–143. Springer, 1999.

[50] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall, April 1985.

[51] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: designing, building, and deploying messaging solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[52] Yi Huang, Eric Cheung, Laura K. Dillon, and R. E. Kurt Stirewalt. A thread synchronization model for SIP servlet containers. In *IPTComm '09*, pages 7:1–7:12. ACM, 2009.

[53] ISO/IEC/IEEE 42010. *Systems and software engineering — Architecture description*, 2011.

[54] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, and Jaime R Silva. Documenting component and connector views with UML 2.0. Technical report, DTIC Document, 2004.

[55] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.

[56] Ethan K Jackson and Janos Sztipanovits. Using separation of concerns for embedded systems design. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 25–34. ACM, 2005.

[57] Pallavi Joshi and Koushik Sen. Predictive typestate checking of multithreaded java programs. In *Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering*, pages 288–296. IEEE Computer Society, 2008.

[58] Tomas Kalibera and Petr Tuma. Distributed component system based on architecture description: The Sofa experience. In *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, pages 981–994. Springer, 2002.

[59] P.K. Kapur, R.B. Garg, and Santosh Kumar. *Contributions to hardware and software reliability*, volume 3. World Scientific, 1999.

[60] Uwe Keller. Some remarks on the definability of transitive closure in first-order logic and Datalog. Internal report, Digital Enterprise Research Institute (DERI), University of Innsbruck, 2004.

[61] Jung Soo Kim and David Garlan. Analyzing architectural styles with Alloy. In *Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis*, ROSATEA '06, pages 70–80, New York, NY, USA, 2006. ACM.

[62] Christian Koehler, Alexander Lazovik, and Farhad Arbab. Connector rewriting with high-level replacement systems. *Electronic Notes in Theoretical Computer Science*, 194(4):77–92, 2008.

[63] Christian Krause, Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. Modeling dynamic reconfigurations in Reo using high-level replacement systems. *Sci. of Comp. Prog.*, 76(1):23–36, 2011.

[64] Daniel Le Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, July 1998.

[65] Douglas Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.

[66] Edward A. Lee and Yuhong Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing*, 16(3):210–237, 2004.

[67] Leonid Libkin. *Elements of finite model theory*. Springer Science & Business Media, 2013.

[68] David C. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events, 1996.

[69] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Anthony Tang. What industry needs from architectural languages: A survey. *Software Engineering, IEEE Transactions on*, 39(6):869–891, 2013.

[70] Anastasia Mavridou, Eduard Baranov, Simon Bliudze, and Joseph Sifakis. Configuration logics: Modelling architecture styles. 2015. Proceedings of the 12th International Conference on Formal Aspects of Component Software, Springer, 2015.

[71] Anastasia Mavridou, Eduard Baranov, Simon Bliudze, and Joseph Sifakis. Architecture Diagrams: A Graphical Language for Architecture Style Specification. In *Proceedings of the 9th Interaction and Concurrency Experience*, Electronic Proceedings in Theoretical Computer Science, 2016.

[72] Anastasia Mavridou, Eduard Baranov, Simon Bliudze, and Joseph Sifakis. Configuration Logics and Architecture Diagrams for Modelling Architecture Styles. *Submitted to Science of Computer Programming. Under review*, 2016.

[73] Anastasia Mavridou, Eduard Baranov, Simon Bliudze, and Joseph Sifakis. Configuration logics: Modeling architecture styles. *Journal of Logical and Algebraic Methods in Programming*, 86(1):2 – 29, 2017.

[74] Anastasia Mavridou, Emmanouela Stachtiari, Simon Bliudze, Anton Ivanov, Panagiotis Katsaros, and Joseph Sifakis. Architecture-based Design: A Satellite On-board Software Case Study. *Accepted to the 13th International Conderence of Formal Aspects in Component Software. To appear*, 2016.

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

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

[77] Dan North. JBehave. A framework for behaviour driven development (BDD), 2011.

[78] OMG Unified Modeling Language (OMG UML) specification, Version 2.5. http://www.omg.org/spec/UML/2.5/. (Accessed on 19/01/2016).

[79] Mourad Oussalah, Adel Smeda, and Tahar Khammaci. An explicit definition of connectors for component-based software architecture. In *Engineering of Computer-Based Systems, 2004. Proceedings. 11th IEEE International Conference and Workshop on the*, pages 44–51. IEEE, 2004.

[80] Mert Ozkaya and Christos Kloukinas. Are we there yet? Analyzing architecture description languages for formal analysis, usability, and realizability. In *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*, pages 177–184. IEEE, 2013.

[81] Mert Ozkaya and Christos Kloukinas. Design-by-contract for reusable components and realizable architectures. In *Proceedings of the 17th international ACM Sigsoft symposium on Component-based software engineering*, pages 129–138. ACM, 2014.

[82] Sebastian Pavel, Jacques Noyé, Pascal Poizat, and Jean-Claude Royer. A Java implementation of a component model with explicit symbolic protocols. In *Software Composition*, volume 3628 of *LNCS*, pages 115–124. Springer, 2005.

[83] Doron A. Peled. *Software reliability methods.* Springer Science & Business Media, 2013.

[84] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[85] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[86] Grzegorz Rozenberg. *Handbook of graph grammars and computing by graph transformation*, volume 1. World Scientific, 1997.

[87] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on software engineering*, 21(4):314–335, 1995.

[88] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*, volume 1. Prentice Hall Englewood Cliffs, 1996.

[89] Joseph Sifakis. A framework for component-based construction. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 293–299. IEEE, 2005.

[90] Joseph Sifakis. Rigorous system design. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 292–292. ACM, 2014.

[91] Joseph Sifakis, Saddek Bensalem, Simon Bliudze, and Marius Bozga. A theory agenda for component-based design. In *Software, Services, and Systems*, pages 409–439. Springer, 2015.

[92] James Smith and Giovanni De Micheli. Automated composition of hardware components. In *Proceedings of the 35th annual Design Automation Conference*, pages 14–19. ACM, 1998.

[93] Carlos Solís and Xiaofeng Wang. A study of the characteristics of behaviour driven development. In *SEAA 2011*, pages 383–387. IEEE, 2011.

[94] Agence spatiale européenne and Karen Fletcher. *Sentinel-3: ESA's Global Land and Ocean Mission for GMES Operational Services.* ESA communications, 2012.

[95] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[96] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. *Software architecture: foundations, theory, and practice.* Wiley Publishing, 2009.

[97] Darshan D. Thaker, Rajeevan Amirtharajah, Francois Impens, Isaac L. Chuang, and Frederic T. Chong. Recursive TMR: Scaling fault tolerance in the nanoscale era. *Design & Test of Computers, IEEE*, 22(4):298–305, 2005.

[98] The Apache Software Foundation. Apache Camel:Routes. http://camel.apache.org/routes.html. (Accessed on 12/02/2016.).

[99] Rob Van Ommering, Frank Van Der Linden, Jeff Kramer, and Jeff Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

[100] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With UML.* Addison-Wesley, 1998.

[101] Peter Wegner. Coordination as constrained interaction (extended abstract). In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 28–33. Springer Berlin Heidelberg, 1996.

[102] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes*, 27(4):218–228, July 2002.

[103] Anthony Williams. *C++ Concurrency in Action: Practical Multithreading.* Manning Publications, 2012.

[104] Eoin Woods and Rich Hilliard. Architecture description languages in practice session report. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 243–246. IEEE Computer Society, 2005.

[105] Charles Zhang. FlexSync: An aspect-oriented approach to Java synchronization. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 375–385, Washington, DC, USA, 2009. IEEE Computer Society.

[106] Da-Qian Zhang, Kang Zhang, and Jiannong Cao. A context-sensitive graph grammar formalism for the specification of visual languages. *The Computer Journal*, 44(3):186–200, 2001.

ANASTASIA MAVRIDOU

EPFL – Rigorous System Design Laboratory
INJ 338 Station 14,1015 Lausanne, Switzerland
Email: anastasia.mavridou@epfl.ch
Web: https://people.epfl.ch/anastasia.mavridou

EDUCATION

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland   September 2012 – present
PhD in Computer Science

- PhD thesis title: Modeling Architecture Styles
- Supervisors: Prof. Joseph Sifakis and Dr. Simon Bliudze
- Studied Configuration Logics (propositional and higher-order) and Architecture Diagrams (graphical language) for modeling architecture styles
  - Identified and modeled architecture styles of on-board satellite software
  - This work was partially funded by the European Space Agency (ESA) as part of the *Catalogue of System and Software Properties* project
- Designed and implemented the JavaBIP engine for run-time coordination of software components
  - Symbolic implementation based on Binary Decision Diagrams
  - Implementation available online at: http://risd.epfl.ch/javabip
  - This work was partially funded by the Swiss Commission for Technology and Innovation as part of the *Coordination of software components in Java* project

University of Tulsa (TU), Oklahoma, USA   August 2010 – May 2012
MSc in Computer Science

- GPA: 4.0/4.0
- MSc thesis title: A Situational Awareness Framework for the Smart Grid
- Supervisor: Prof. Mauricio Papa

Aristotle University of Thessaloniki (AUTH), Greece   October 2004 – February 2010
Dipl.-Ing (5-year program) in Electrical and Computer Engineering

- Diploma thesis title: Information Assurance for Cybersecurity and Critical Infrastructure Protection: Policies, Guidelines and Standards
- Supervisors: Prof. George Pangalos (AUTH, Greece) and Prof. Bernard S. Doherty (Aston University, UK)

RESEARCH INTERESTS

- **Architecture-based design**
  - Specification of architecture styles
  - Formal semantics
  - Synthesis of configurations
  - Composition of architecture styles
- **Coordination of software components**
  - BIP coordination mechanisms
  - Separation of concerns between functionality and coordination
  - Coordination of dynamic systems
- **Modeling and analysis of critical systems**
  - Correctness by construction

PROFESSIONAL EXPERIENCE

Supervision of a Master thesis, EPFL February 2015 – June 2016

- Title: Introducing Dynamicity in JavaBIP
- Student name: Valentin Rutz
- Extended the theory and implementation of the JavaBIP framework with dependency graphs to handle insertion and deletion of components at run-time

Research Visit, Vanderbilt University June 2015 – August 2015

- Institute for Software Integrated Systems, Nashville, Tennessee, USA
- Supervisor: Prof. Janos Sztipanovits
- Modeled architecture styles of the Greybus protocol (Google Ara Project) using Configuration Logics and Architecture Diagrams

Graduate Research Assistant, TU August 2010 – May 2012

- Institute for Information Security, Tulsa, Oklahoma, USA
- Worked on a Critical Infrastructure Protection research project partially funded by DARPA
- Designed and developed a monitoring system that collects and analyzes industrial traffic of SCADA systems to detect behavioral anomalies
- Supervised two undergraduate students

Industrial Internship June 2008 – August 2008

- Panel Industrial Electric company, Istanbul, Turkey
- Industrial internship as part of the IAESTE student exchange program
- Developed Ladder Logic programs for Siemens Programmable Logic Controllers

Reviewer for Journal

- Information Systems, Elsevier

TEACHING EXPERIENCE

Teaching Assistant, EPFL September 2013 – present

- **Programming in C** (CS-111(f) & CS-111(b))
  - taught undergraduate students in exercise sessions
  - updated course contents
  - graded exams
- **Concurrency** (CS 206)
  - taught undergraduate students in exercise sessions
  - graded projects and exams
- **Linear Algebra** (MATH-111(e))
  - taught undergraduate students in exercise sessions

PUBLICATIONS

Proceedings

1. **Mavridou, A.**, Stachtiari, E., Bliudze, S., Ivanov, A., Katsaros, P., & Sifakis, J. (2016). *Architecture-based Design: A Satellite On-board Software Case Study*. Accepted to the 13th International Conference on Formal Aspects of Component Software.

2. **Mavridou, A.**, Baranov, E., Bliudze, S., & Sifakis, J. (2016). *Architecture Diagrams: A Graphical Language for Architecture Style Specification*. 9th Interaction and Concurrency Experience. Electronic Proceedings in Theoretical Computer Science.
3. **Mavridou, A.**, Baranov, E., Bliudze, S., & Sifakis, J. (2015). *Configuration Logics: Modelling Architectures Styles*. 12th International Conference on Formal Aspects of Component Software. Lecture Notes in Computer Science 9539.
4. Bliudze, S., **Mavridou, A.**, Szymanek, R., & Zolotukhina, A. (2014). *Coordination of software components with BIP: application to OSGi*. 6th International Workshop on Modeling in Software Engineering (pp. 25-30). ACM.
5. **Mavridou, A.**, & Papa, M. (2012). *A Situational Awareness Architecture for the Smart Grid*. 7th International Conference on Global Security, Safety and Sustainability & e-Democracy (pp. 229-236). Springer.
6. Kerkiri, T., Manitsaris, A., **Mavridou, A.** (2007). *Reputation Metadata for Recommending Personalized e-Learning Resources.* 2nd International Workshop on Semantic Media Adaptation and Personalization (pp.110-115). IEEE.

Journal

1. **Mavridou, A.**, Baranov, E., Bliudze, S., & Sifakis, J. (2016). *Configuration Logics and Architecture Diagrams for Modelling Architecture Styles*. Submitted to: Science of Computer Programming. Under review.
2. Bliudze, S., **Mavridou, A.**, Szymanek, R., & Zolotukhina, A. (2016). *Exogenous Coordination of Concurrent Software Components with JavaBIP*. Submitted to: Software: Practice and Experience. Under review.
3. **Mavridou, A.**, Baranov, E., Bliudze, S., & Sifakis, J. (2016). *Configuration Logics: Modelling Architectures Styles*. Journal of Logical and Algebraic methods in Programming.
4. **Mavridou, A.**, Zhou, V., Dawkins, J., & Papa, M. (2012). *A situational awareness framework for securing the smart grid using monitoring sensors and threat models.* International Journal of Electronic Security and Digital Forensics, 4(2/3), 138-153.

Book chapter

1. Brundage, M., **Mavridou, A.**, Johnson, J., Hawrylak, P. J., & Papa, M. (2012). *Distributed Monitoring: A Framework for Securing Data Acquisition*. Securing Critical Infrastructures and Critical Control Systems: Approaches for Threat Protection, 144.

Technical reports

1. **Mavridou, A.**, Baranov, E., Bliudze, S., & Sifakis, J. (2016). *Architecture Diagrams- A Graphical Language for Architecture Style Specification*. EPFL Technical report.
2. **Mavridou, A.**, Baranov, E., Bliudze, S., & Sifakis, J. (2015). *Configuration Logics - Modelling Architecture Styles*. EPFL Technical report.
3. Bliudze, S., **Mavridou, A.**, Szymanek, R., & Zolotukhina, A. (2013). *Integration of BIP into Connectivity Factory: Implementation*. EPFL Technical report.

Presentations

1. Bliudze, S., **Mavridou, A.**, Szymanek R., & Zolotukhina, A. (2013). *For Coordination, State Component Transitions*. EclipseCon.
2. Brundage, M., Johnson, J., **Mavridou, A.**, Hawrylak, P., Papa, M. (2011). *The Internet of Things - Critical Infrastructure: Complete Security Solution*. AIM Expo.

FUNDS AND SCHOLARSHIPS

- Participation fund, Marktoberdorf Summer School on Dependable Software Systems Engineering, Germany (2014)
- Graduate Tuition Award, University of Tulsa, USA (8/2010 – 5/2012)
- Erasmus scholarship, Aston University, UK (3/2008-6/2008)
- Distinction and award, Ministry of Education, Greece (2004)

SKILLS

Programming languages and tools

- Java, C/C++
- Matlab, Ladder Logic
- JavaBDD and CUDD libraries for manipulating Binary Decision Diagrams

Modelling languages and tools

- BIP
- Alloy, Alloy Analyzer

Other tools

- Z3 theorem prover
- nuXmv model checker
- JaCoP constraint solver

Languages

- Greek: native
- English: full professional
- French: basic