Universiteit
Leiden
The Netherlands

Scheduled protocol programming

Dokter, K.P.C.

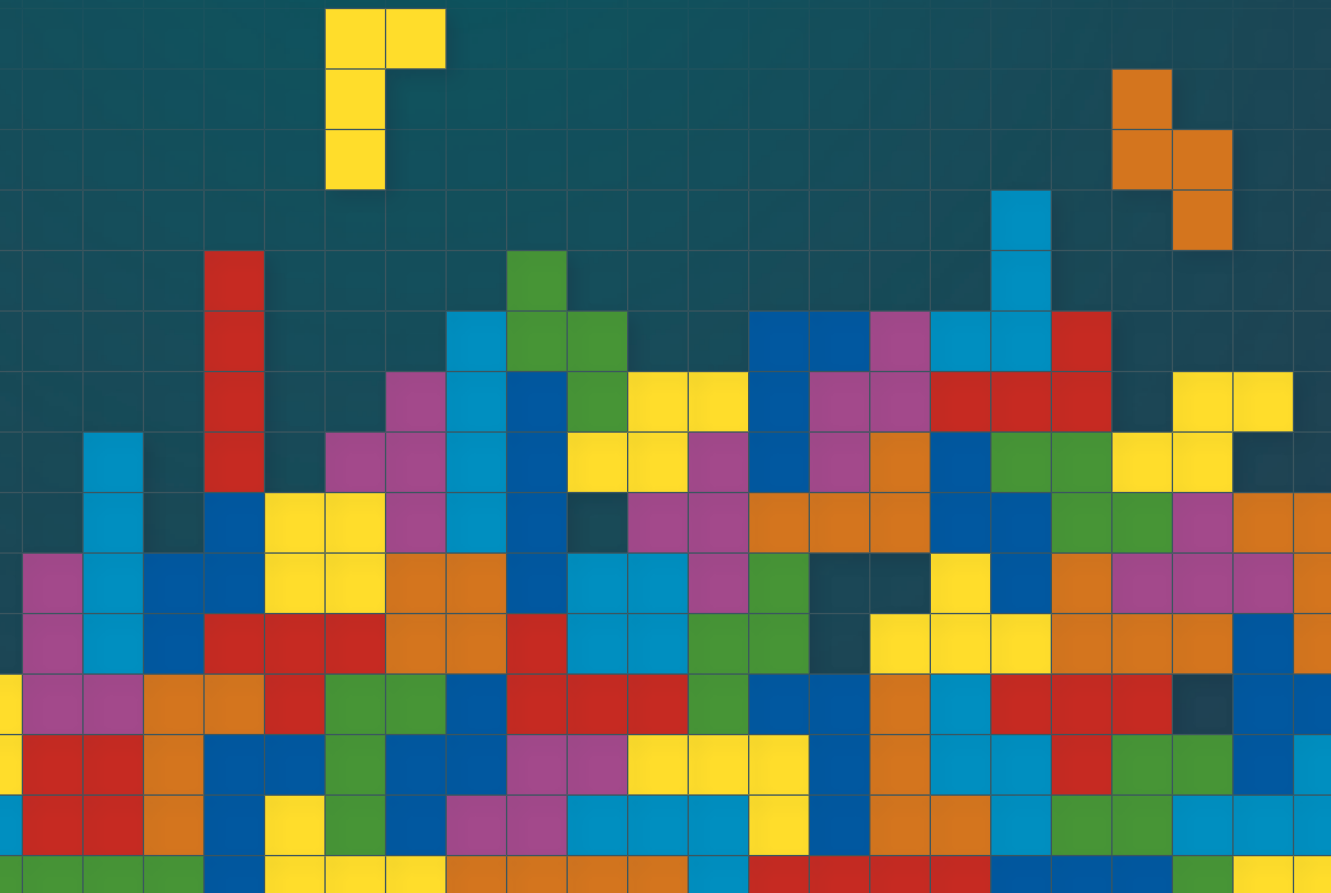
Citation
Dokter, K. P. C. (2023, May 24). *Scheduled protocol programming*. Retrieved from https://hdl.handle.net/1887/3618490

Version:	Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/3618490

Note: To cite this publication please use the final published version (if applicable).

Scheduled Protocol Programming

Kasper Dokter



Scheduled Protocol Programming

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Leiden,
op gezag van rector magnificus prof.dr.ir. H. Bijl,
volgens besluit van het college voor promoties
te verdedigen op woensdag 24 mei 2023
klokke 11:15 uur

door

Klaas Pieter Cornelis Dokter
geboren te Putten
in 1990

Promotores

prof. dr. F. Arbab
prof. dr. M.M. Bonsangue

Promotiecommissie

prof. dr. C. Baier	Technische Universität Dresden
prof. dr. H.C.M. Kleijn	
prof. dr. J.C. van de Pol	Aarhus University
prof. dr. A. Plaat	
prof. dr. F.J. Verbeek	

The work in this thesis has been carried out at CWI (Centrum Wiskunde & Informatica) and Universiteit Leiden, under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Contents

1 Introduction **7**
1.1 Context . . . 7
1.2 Problem . . . 8
1.3 Related work . . . 10
1.4 Contributions . . . 11

I Coordination **17**

2 Coordination Languages **19**
2.1 Overview of BIP and Reo . . . 20
2.2 Port automata and BIP architectures . . . 29
2.3 Stateless CA's and interaction models . . . 38
2.4 Data-sensitive BIP architectures . . . 43
2.5 Related work . . . 53
2.6 Discussion . . . 54

3 Protocols with Preferences **57**
3.1 Preliminaries on soft constraints . . . 58
3.2 Soft constraint automata with memory . . . 65
3.3 Case study . . . 72
3.4 Application to context-sensitivity . . . 75
3.5 Related work on constraint automata . . . 77
3.6 Discussion . . . 79

II Compilation **81**

4 Protocol Syntax **83**
4.1 Treo syntax . . . 85
4.2 Semantic sorts . . . 88
4.3 Denotational semantics . . . 90
4.4 Input/output nodes . . . 94
4.5 Related work . . . 96
4.6 Discussion . . . 96

5 Protocols as Constraints **99**
5.1 Related work . . . 100
5.2 Syntax and semantics . . . 101
5.3 Regular constraints . . . 103
5.4 Reflexive constraints . . . 105
5.5 Rule-based form . . . 107
5.6 Composition . . . 109
5.7 Complexity . . . 112
5.8 Abstraction . . . 114
5.9 Application . . . 115
5.10 Discussion . . . 116

6 Protocols as Petri nets **117**
6.1 Preliminaries . . . 120
6.2 Multilabeled Petri nets . . . 122
6.3 Composition . . . 125
6.4 Abstraction . . . 131
6.5 Discussion . . . 135

III Scheduling **137**

7 Protocols with Workloads **139**
7.1 Work automata . . . 140
7.2 State Space Minimization . . . 145
7.3 Related work . . . 151
7.4 Discussion . . . 151

8 Protocol Scheduling **153**
8.1 Graph games . . . 154
8.2 Scheduling Game . . . 158
8.3 Protocol restriction . . . 161
8.4 Related work . . . 163
8.5 Discussion . . . 164

9 Conclusion **165**
9.1 Summary . . . 165
9.2 Future work . . . 167

Bibliography **169**

Appendices **185**

A Original CSDF program **185**

B Scheduled CSDF program **187**

Curriculum Vitae **191**

List of Publications **193**

Samenvatting **195**

Summary **196**

Acknowledgments **198**

Chapter 1

Introduction

1.1 Context

A *schedule* is a ternary relation amongst time, resources, and tasks. That is, a schedule is an allocation of time and resources to tasks. We represent time as all non-negative, real-valued timestamps. The collection of all related tasks forms an *application*. An application can run indefinitely, which may happen due to repetitive tasks. We implicitly allow tasks to admit subtasks, although this hierarchical structure is irrelevant for the work presented in this thesis. The number of resources is generally finite, which introduces scarcity.

A *protocol* is a constraint on schedules that determines a set of valid schedules. For example, a protocol may demand that each resource is allocated to at most one task at any given time. A protocol may also constrain the execution times of tasks to follow a particular partial order. An application is sequential if its timing constraint forms a total order, and the application is concurrent, otherwise. Typically, timing constraints of an application specify only the relative order of the start and completion times of its tasks. Real-time applications involve stricter timing constraints that may specify exact (i.e., absolute) start times, completion times, and/or durations for its tasks. Typically, the timing constraints of tasks are not specified explicitly; instead, they are implied by the ordering of actions of a task and their durations.

Scheduling is the activity of selecting a valid schedule (i.e., that respects the constraints of the protocol of the application) that optimizes some given criterion such as total execution time, throughput, latency, or resource utilization. The selected schedule can have a significant impact on the quality of the application. For example, in mission critical applications like a pacemaker or a flight controller it is imperative to avoid catastrophic failure by producing reliable, predictable output.

Applications can exhibit two types of non-deterministic behavior. Internal non-determinism is under our control and is resolved by scheduling. External non-determinism (such as user input) is uncontrolled and requires flexible schedules. However, the ternary relations defined above are deterministic and leave no room for uncertainty. Therefore, a scheduler generally determines a *scheduling strategy*,

which is a collection of schedules. At run time, a scheduler can follow any such schedule that is consistent with the given external non-determinism.

The simple and intuitive definitions of schedules and protocols above have significant consequences that we exploit in the work presented in this thesis. The above definitions show that scheduling strategies and protocols are essentially the same (both being sets of schedules), and that as such, protocols and scheduling strategies are convertible into each other. We exploit this convertibility property in Chapter 8 to improve the performance of the operating system scheduler for a cyclo-static dataflow program.

Although both protocols and scheduling strategies are essentially identical, they serve different purposes. Roughly speaking, a protocol primarily expresses the *correctness constraints* of a concurrent application, whereas a scheduling strategy primarily expresses the *efficiency constraints* under which an application or a set of applications execute efficiently, as determined by some objective function that optimizes various concerns such as execution time, throughput, resource utilization, etc. Just as the boundary between functional versus quality of service properties of an application is not always sharply defined, the distinction between correctness constraints (i.e., protocols) from efficiency constraints (i.e., scheduling strategies) is not always crisp, either. Importantly, protocols often contain a wealth of useful information, essential for optimal scheduling. We consider protocols in Part I and scheduling strategies in Part III.

Although we define protocols as constraints on schedules, in practice, protocols are never literally specified as constraints on schedules (i.e., constraints on ternary relations). Instead, traditional models of concurrency and concurrent programming constructs specify a protocol only as a derived by-product of the timing constraints of the tasks that comprise a software. In contrast to traditional models of concurrency, exogenous coordination[1] languages, such as Reo [Arb04], offer first-class constructs for expressing concurrency/coordination protocols as concrete, identifiable, reusable software modules, independent of the computation carried out by an application [Arb11, Arb16].

1.2 Problem

1.2.1 Resource virtualization

Scheduling is generally easy in case of *non-interference*, that is, no two tasks require a shared resource at the same time. Indeed, non-interference assures that we can allocate resources on demand. A trivial example of non-interference is a set of tasks that do not share any resources. However, such task sets are uncommon, because cooperative tasks require shared resources for communication. A more common example of non-interference is a set of tasks that consists of coroutines. Only one of the tasks is active at any given time, which implies non-interference of processors. Other examples of non-interference are a read-only shared variable, or an antenna that can broadcast at different frequencies simultaneously.

[1] Coordination models and languages offer higher-level constructs for specification of protocols that coordinate the tasks that comprise a concurrent application [Arb98].

Non-interference allows for *resource virtualization*, which duplicates a physical resource into multiple virtual resources that can be used by multiple independent applications. Virtualization minimizes the number of otherwise-necessary physical resources, which reduces cost. For example, coroutines can run on dedicated processors. However, since only one of these processors is active at any given time, these dedicated processors can be virtual processors that map to a single physical processor. Similarly, tasks that require antennas to broadcast on different frequency bandwidths can of course use a dedicated physical antenna, each. Non-interference of their bandwidths, however, allows a virtualization that maps all dedicated antennas of these tasks as virtual antennas onto a single physical antenna. Another example of virtualization is virtual memory, which divides memory into pages. Since tasks often work on a single page at time, these memory pages can be virtual memory pages that map to a single physical memory page. Of course, we need additional cheap memory to store unused pages.

In general, resource virtualization requires *interference resolution*. For example, we can virtualize a printer by means of a queue that ensures non-interference. The queue plays a dual role. On the one hand, the queue implements a mutual exclusion protocol that regulates access to the printer. On the other hand, the queue implements the first-come-first-served scheduling strategy. Hence, we observe a first glimpse of a duality between protocols and schedules. Later on, we exploit this duality by transforming a schedule into a protocol.

Software developers generally outsource part of the scheduling by virtualization of frequently used resources. The prime example is the virtualization of processors by the operating system, which creates virtual duplicates of the physical processors in the form of processes and threads. The operating system allows for context-switching by reserving some memory to save the state of each virtual processor. By assigning to each task a dedicated virtual processor, the scheduling problem becomes trivial from the perspective of the software developer.

1.2.2 Conflicting objectives

Resource virtualization usually resolves interference via a generic scheduling strategy that aims at a fair distribution of the physical resource over the virtual ones. Hence, the objective of generic schedulers generally differs from the objectives of applications that use these resources. For example, an operating system scheduler implements a variant of round-robin scheduling that allocates processing time to non-blocked virtual processors. Virtual processors are blocked whenever their associated task accesses another resource (like an I/O device) that is already in use.

However, fair distribution of the physical resources over the virtual ones does not necessarily optimize the desired scheduling objective. For example, consider two tasks that produce ice creams and a single consuming task. Assume that production requires the same amount of work as consumption. Fair distribution of resources yields a production rate that is twice the consumption rate. As a result, most ice creams melt before being consumed. Ideally, the production and consumption rates should be equal to yield maximal throughput.

To optimize the schedule of an application, we must inform the default scheduler in virtual resources about relevant application-specific scheduling information. This

brings us to the main question of this thesis:

How to optimize scheduling in the presence of virtual resources?

There are two ways to inform the generic scheduler in a virtual resource. The first and most straightforward approach is to hard-code this information by replacing the scheduler in the virtual resource with an application-specific scheduler. However, an application-specific scheduler is often unacceptable, as one of the goals of virtualization is to provide seamless resource sharing by multiple independent applications. Hence, the application-specific scheduler is possible only for systems dedicated to a single application, such as embedded devices with a real-time operating system.

The second and less intrusive approach is to change the application (and its constituent tasks), so that the generic default scheduler behaves optimally (or as close to it as possible). This approach is obviously more complex than changing the mapping of virtual resources. In Part III, we develop a technique that alters the application to optimize the scheduling. Our approach is based on the duality between scheduling strategies and protocols mentioned in Section 1.1.

1.3 Related work

The literature on scheduling offers partial solutions to our main question, and we broadly classify them in three categories, namely online scheduling, low-level offline scheduling and high-level offline scheduling.

Online scheduling The OS scheduler uses a variant of generic round-robin scheduling strategy, which schedules fairly by assigning a fixed amount of time (a time quantum) per task, and cycles through the tasks. If a tasks performs a blocking operation, such as read from/write to memory or acquisition of a lock, then the task enters a waiting state. The OS detects when a process is waiting, and does not assign time quanta to that process.

Round-robin scheduling does not require any a priori knowledge of the running task. The OS scheduler cannot predict which tasks it will encounter in the future. Consequently, its performance is suboptimal. One can still provide guarantees on the quality of a given scheduling algorithm, A, by considering its *competitive ratio*, which is defined as the smallest[2] factor C such that

$$\text{cost}_A(x) \leq C \cdot \text{cost}_{\text{optimal}}(x), \qquad \text{for every input } x$$

where $\text{cost}_A(x)$ is the cost of the given online scheduler A that cannot predict the future, and $\text{cost}_{\text{optimal}}(x)$ is the cost of a clairvoyant scheduler that knows all future tasks. For example, if we consider the cost of the schedule length (or makespan), then list scheduling has a competitive ratio of $2 - 1/n$, with $n \geq 1$ the number of machines [Gra66]. Of course, any competitive ratio is at least 1, which corresponds to the ideal situation wherein the scheduler has perfect information on the behavior of the application.

[2]Or the greatest lower bound if no smallest C exists.

Low-Level offline scheduling Round-robin scheduling leaves plenty of room for optimization. To squeeze even more performance out of a given software, we turn our attention to application-specific scheduling.

One generic approach in improving the performance of software is to perform *instruction scheduling*. The GNU Compiler Collection is a C compiler that performs instruction scheduling when using the `-march` or `-mtune` flags. Instruction scheduling is a compiler optimization that reorders independent statements in a basic block to optimize the total execution time. A basic block is a sequence of instructions without branching, which constitutes a single node in the control flow graph of the application. Two instructions are independent and can be reordered, when neither instruction writes to a memory location that is accessed by the other. For example, two instructions that do not access the same memory location are independent, and two instructions that both read from the same memory location are independent.

Since all instructions and their dependencies are known, it is possible to produce an optimal instruction order. However, finding the optimal schedule is computationally intensive. Therefore, one often pretends that instruction scheduling is an online scheduling problem, and uses the list scheduling that we discussed above. The resulting execution time at most doubles, because the competitive ratio of list scheduling is $2 - 1/n$, for n machines.

High-Level offline scheduling Instruction scheduling improves the software execution by considering very low-level information on the software. As a result, instruction scheduling alone is generally insufficient, as it does not consider high-level information, like the way different parts of the software interact.

For specific application domains, such as digital signal processing (DSP), this extra performance gain is necessary. To this end, one develops an application in a domain specific language (DSL) that is amenable to the extraction of relevant scheduling information. Examples of such DSLs include CSDF graphs and Kahn process networks.

However, most software is written in a general-purpose language, such as Python, C, or Java[3]. Scheduling techniques developed for software written in DSLs for specific application domains do not readily apply to software written in these general-purpose languages, because the extraction of relevant scheduling information is much more difficult for software written in general-purpose languages than for software written in these DSLs. The primary reason is that the protocol of software written in languages that use traditional models of concurrency is implicit, nebulous, and diffused among the large number of lines of code that comprise the software [Arb11]. Without an explicit protocol, we cannot know in advance which schedules will be valid at run time.

1.4 Contributions

This thesis is structured into three parts, namely coordination, compilation, and scheduling.

[3] `https://www.tiobe.com/tiobe-index/`

1.4.1 Coordination

Scheduling requires detailed information on the protocol of an application. For software written in general-purpose programming languages, such information is generally not available. Therefore, we turn in Part I our attention to coordination languages, which simplify design and development of concurrent systems. Particularly, exogenous coordination languages, like BIP [BBS06, BS07] and Reo [Arb04, Arb11], enable system designers to express the interactions among components in a system explicitly. A formal relation between exogenous coordination languages comprises the basis for a solid comparison and consolidation of their fundamental concepts.

In Chapter 2, we establish a formal relation between BI(P) (i.e., BIP without the priority layer) and Reo, by defining transformations between their semantic models. We show that these transformations preserve all properties expressible in a common semantics. We use these transformations to define data-sensitive BIP architectures and their composition.

In Chapter 3, we address the discrepancy between Reo and BIP regarding priority by revising *soft constraint automata*, wherein transitions are weighted and each action has an associated preference value [AS12, KAT16, KAT17]. Soft constraint automata can be used as a semantics for Reo connectors. These preferences can be used to guide the selection of an interaction that is enabled by an (unpredictable) environment.

We revise soft constraint automata in three ways. First, we relax the underlying algebraic structure to allow bipolar preferences, which allow one to express positive preferences (I like X) as well as negative preferences (I do not like X). Next, we equip automata with memory locations, that is, with an internal state to remember and update information from transition to transition. Finally, we revise automata operators, such as composition and hiding, providing examples on how such memory locations interact with preferences.

We show the utility of our revised soft constraint automata by encoding context-sensitive behavior in terms of these soft constraint automata.

1.4.2 Compilation

The exogenous coordination languages from Part I offer an explicit protocol specification that governs the interactions amongst a set of tasks. However, such a protocol specification does not contain all information necessary for our scheduling purposes. The main shortcoming is that the protocol specifications from Part I do not specify the *workload* of each task, which is the amount of processing time required by that task. Furthermore, the implementation of a protocol specification may require one or more auxiliary (protocol) tasks.

In Part II, we provide all necessary ingredients for a complete protocol specification by developing a compiler that translates these protocol specifications into executable code. The original tasks, together with the auxiliary tasks, completely define the scheduling problem. Although it is certainly possible to derive or approximate the workload of each task by analyzing the code generated by our compiler (including the target architecture), we leave this as future work.

Recent benchmarks show that compiling high-level Reo specifications produces executable code that can compete with or even beat the performance of hand-

crafted programs written in languages such as C or Java using conventional concurrency constructs [JHA14, JA15, JA16b, JA16a, JA18].

The original declarative graphical syntax of Reo does not support intuitive constructs for parameter passing, iteration, recursion, or conditional specification [ECT]. This shortcoming hinders Reo's uptake in large-scale practical applications. Therefore, a number of Reo-inspired syntax alternatives have appeared in the past, such as RSL [Klü12] and FOCAML [Jon16]. However, none of them follow the primary design principles of Reo: a) declarative specification; b) user-defined channels and channel types; and c) channels compose via shared nodes.

In Chapter 4, we offer a textual syntax for Reo (called Treo) that respects these principles and supports flexible parameter passing, iteration, recursion, and conditional specification. The Treo language is extensible and allows the creation of new semantics for components, such as the soft constraint automaton semantics from Chapter 3. In on-going work, we use this textual syntax to compile Reo into target languages such as Java, Promela, and Maude.

The state-of-the-art Reo compiler (Lykos) by Jongmans represents protocols as constraint automata [Jon16], which specify protocols as labeled transition systems that preserve synchronization under composition. Besides compilers, constraint automata have been used as a basis for other tools like model checkers [Klü12]. Unfortunately, composition of transition systems suffers from state space and transition space explosions, which limits scalability of the tools based on constraint automata.

Clarke et al. avoid these problems by representing protocols as constraints [CPLA11]. These constraints determine, at every point in time, which combination of actions constitute an interaction. Constraints are composed via conjunction, which means that the size of the representation is potentially linear in the number of components. However, the approach by Clarke et al. relies on a constraint solver, which incurs a significant overhead in the run time.

In Chapter 5, we avoid the overhead of the constraint solver by transforming constraints directly into executable code, without relying on a constraint solver. More precisely, we apply a commandification technique developed by Jongmans that generates low-level executable code for a constraint that does not contain any disjunction [JA15]. We extend this commandification to arbitrary constraints by rewriting them into a *rule-based form*. We provide sufficient conditions under which our approach avoids transition space explosions. As a result, we can now compile protocols that could not be compiled by Lykos.

Constraint-based representations of a Reo protocol are inherently sequential. Therefore, straightforward compilation of constraints produces sequential code that does not utilize all available computational resources.

In Chapter 6, we make concurrency explicit by reformulating our ideas for constraints in terms of (multilabeled) Petri nets. Similar to the constraint-based approach, our novel composition of Petri nets avoids a state space explosion, which makes them an adequate intermediate representation for code generation. Moreover, these Petri nets can be viewed as an inherently parallel generalization of constraint automata, which makes it possible to generate concurrent code. This approach is a refinement of the so called synchronous region decomposition of Reo connectors [JCP16].

Moreover, we also present an abstraction operator for multilabeled nets that

eliminates internal transitions, which optimizes the execution of multilabeled nets.

1.4.3 Scheduling

The Reo compiler developed in Part II produces code that relies on the operating system (OS) scheduler to assign computational resources to its tasks. The OS scheduler aims at fair distribution of processing time amongst its given tasks. As mentioned earlier (Section 1.2.2), this objective usually conflicts with the scheduling objective of any given application. In Part III, we address the main question in Section 1.2.2 by computing an optimal scheduling strategy for a given Reo protocol, and guiding the OS scheduler to follow this optimal strategy.

In Chapter 7, we introduce work automata to express all relevant scheduling information of a given Reo protocol. The work automaton semantics of Reo addresses an important shortcoming of the protocol specifications in Part I by explicitly specifying the workload of each task. The modular structure of the Treo language allows us to add syntax to express primitive work automata. Our generic (semantics-agnostic) Reo compiler composes these primitive work automata to produce the work automaton for the complete application. We potentially[4] benefit from all semantics-independent compiler optimizations, such as the queue-optimization [JHA14] and protocol splitting [JCP16].

Work automata specify behavior and workloads. The specification of behavior is fully controlled by the developer of the application. The workloads, however, are not directly under the control of the application developer, and these workloads can be determined only after compilation. In the current work, we assume that the work automata (including workloads) are given, and we leave the derivation or approximation of workloads as future work.

We provide a formal semantics for work automata, based on which we introduce equivalences such as weak simulation and weak language inclusion. Subsequently, we define operations on work automata that simplify them while preserving these equivalences. Where applicable, these operations simplify a work automaton by merging its different states into a state with a 'more inclusive' state-invariant. The resulting state-invariant defines a region in a multidimensional real vector space that potentially contains holes, which in turn expose mutual exclusion among processes. Such exposed dependencies provide additional insight into the behavior of an application, which can enhance scheduling. Our operations, therefore, potentially expose implicit dependencies among processes that otherwise may not be evident to exploit.

In Chapter 8, we use the scheduling information of a Reo protocol (presented as a work automaton) to compute an optimal scheduling strategy for that application. To be precise, we use a generic, game-theoretic scheduling framework to find optimal non-preemptive schedules for an application.

A straightforward approach to implement the resulting schedule is to replace the OS scheduler with a custom scheduling strategy. However, as mentioned earlier, such replacement is highly non-trivial and defeats the purpose of the OS as it turns it into a platform optimized to run only one application.

[4] Our current compiler does not yet implement the queue-optimization or protocol splitting.

Therefore, our answer to the main question in Section 1.2.2 is to alter the application so that its scheduling problem becomes trivial. Our main idea is to use the duality from Section 1.1 to convert the ideal scheduling strategy into a scheduling protocol. Our compositional framework allows us to compose this scheduling protocol with the original application protocol. This composition can be viewed as a source-to-source Reo transpiler that addresses scheduling concerns. The resulting new protocol of the application has only one valid schedule, which is the optimal schedule of the original application. As a result, we can safely outsource the scheduling to the operating system, as the operating system scheduler is forced to(closely) follow this optimal schedule.

We evaluate our work by comparing the throughput of two versions of a cyclo-static dataflow network: one version with the usual protocol, and the other version with a restricted protocol.

In Chapter 9, we present our conclusions and future work.

Part I

Coordination

Chapter 2

Coordination Languages

Software scheduling requires detailed information on the interaction and workload of the tasks in the given application[1]. For software written in general-purpose programming languages, such information is generally not available, because the interaction protocol is implemented with basic primitives such as locks, semaphores, and (a)synchronous message passing. With such primitives, the code of the interaction protocol gets easily mixed with the application code, which renders the analysis, optimization and reusability of the implemented protocol impossible.

Exogenous coordination languages, like BIP [BBS06, BS07] and Reo [Arb04, Arb11], make the interaction protocol explicit by separating coordination of interactions from computation in processes [PA01]. This enables designers to control interaction using language constructs, making coordination visible to tools like model checkers, compilers and schedulers.

In BIP, a concurrent system consists of a superposition of three layers: behavior, interaction and priorities. The behavior layer contains the processes that need to be coordinated. The interaction layer explicitly specifies which interactions are possible, which gives full control over the interactions in the system. Mutually exclusive execution of these interactions ensures that overlapping interactions do not cause a conflict. If multiple interactions are possible, then the priority layer selects a preferred one.

In Reo, processes interact by means of a coordination protocol. A protocol consists of a graph-like structure, called a connector, that models the synchronization and dataflow among the processes. Reo connectors may compose together to form more complex connectors, allowing reusability and compositional construction of coordination protocols.

Although BIP and Reo address the same coordination problem, their underlying design principles and toolchains (containing tools for editing, code generation and model checking [bip16, reo16, Arb11]) differ significantly. By combining their principles and tools, we would conquer new terrain in the field of concurrent languages. However, some principles(visible in the formal definitions of each language) may be conflicting, and prevent such a complete unification. A formal relation between BIP and Reo is necessary to identify these conflicts.

[1]The work in this chapter is based on [DJAB17, DJAB15]

In this chapter, we provide such a formal relation between BIP and Reo by relating their semantic models. We consider two kinds of semantic models for BIP and Reo: data-agnostic and data-sensitive. In the data-agnostic domain, we relate port automata as semantics of Reo and BIP architectures [JA12, ABB+14]. We show that connectors in BIP and Reo coincide modulo internal transitions and independent progress of transitions. In the data-sensitive domain, we relate stateless constraint automata as semantics of Reo with BIP interaction models [JA12, BSBJ14]. The restriction to stateless constraint automata arises from the fact that BIP interaction models are stateless. We show that stateless constraint automata and BIP interaction models have the same observable behavior.

Stateful data-sensitive Reo connectors require stateful constraint automata for their semantics, which informally correspond to data-sensitive BIP architectures. A data-sensitive BIP architecture consists of a (data-sensitive) BIP interaction model together with a set of coordinating components. However, current literature on BIP does not provide definitions that allow composition of data-sensitive BIP architectures. Indeed, only hierarchical composition of interaction models is defined in [BSBJ14], which is insufficient to define a full composition of data-sensitive BIP architectures.

We address this problem by using our formal translations to propose a composition operator for data-sensitive BIP architectures. In addition, we show that it is possible to relate (stateful) constraint automata and data-sensitive BIP architectures.

Although BIP's notion of priority is equally applicable to the constraint automata semantics of Reo, Reo provides no syntax to specify such global priority preferences.[2] Therefore, in this chapter, "BIP" generally refers to "BI(P)", a name that others have already used to designate BIP without its priority layer.

The rest of this chapter is organized as follows: In Section 2.1, we recall the semantic models of BI(P) and Reo. In Section 2.2, we relate port automata in Reo and BIP architectures. In Section 2.3, we relate BIP interaction models with stateless constraint automata in Reo. In Section 2.4, we propose an extension of data-agnostic BIP architectures to the data-sensitive domain, and show how this enables incremental translation from stateful constraint automata to data-sensitive BIP architectures. In Section 2.5, we discuss related work. In Section 2.6, we conclude and point out future work.

2.1 Overview of BIP and Reo

2.1.1 BIP

A BIP system consists of a superposition of three layers: Behavior, Interaction, and Priority. The behavior layer encapsulates all computation, consisting of *atomic components* processing sequential code. *Ports* form the interface of a component through which it interacts with other components. BIP represents these atomic

[2] Reo does have a weaker priority mechanism to specify local preferences, called *context-sensitivity*. A premier example in the Reo literature is the context-sensitive channel LossySync, which prefers locally maximal dataflow. Clarke et al. first studied context-sensitivity through a special context-sensitive semantic model for Reo [CCA07]; later, Jongmans et al. showed how to encode context-sensitivity in non-context-sensitive models [JKA11].

components as *Labelled Transition Systems* (LTS) having transitions labelled with ports and extended with data stored in local variables. The second layer defines component coordination by means of *BIP interaction models* [BSBJ14]. For each *interaction* among components in a BIP system, the interaction model of that system specifies the set of ports synchronized by that interaction and the way data is retrieved, filtered and updated in each of the participating components. In the third layer, priorities impose scheduling constraints to resolve conflicts in case alternative interactions are possible.

In the rest of this chapter, we disregard priorities and focus mainly on interaction models (cf. footnote 2).

Data-agnostic semantics We first introduce a data-agnostic semantics for BIP.

Definition 2.1.1 (BIP component [ABB+14])**.** A *BIP component* C over a set of ports P_C is a labelled transition system $(Q, q^0, P_C, \rightarrow)$ over the alphabet 2^{P_C}. If $\mathcal{C}$ is a set of components, we say that $\mathcal{C}$ is *disconnected* iff $P_C \cap P_{C'} = \emptyset$ for all distinct $C, C' \in \mathcal{C}$. Furthermore, we define $P_{\mathcal{C}} = \bigcup_{C \in \mathcal{C}} P_C$.

Then, BIP defines an *interaction model* over a set of ports P to be a set of subsets of P. Interaction models are used to define synchronizations among components, which can be intuitively described as follows. Given a disconnected set of BIP components $\mathcal{C}$ and an interaction model γ over $P_{\mathcal{C}}$, the state space of the corresponding *composite component* $\gamma(\mathcal{C})$ is the cross product of the state spaces of the components in $\mathcal{C}$; $\gamma(\mathcal{C})$ can make a transition labelled by an interaction $N \in \gamma$ iff all the involved components (those that have ports in N) can make the corresponding transitions. A straightforward formal presentation can be found in [BS07] (cf. Definition 2.1.3 below). Thus, BIP interaction models are *stateless*: every interaction in γ is always allowed; it is enabled if all ports in the interaction are ready. However, [ABB+14] shows the need for stateful interaction, which motivates *BIP architectures*.

Definition 2.1.2 (BIP architecture [ABB+14])**.** A *BIP architecture* is a tuple $A = (\mathcal{C}, P_A, \gamma)$, where $\mathcal{C}$ is a finite disconnected set of *coordinating* BIP components, P_A is a set of ports, such that $P_{\mathcal{C}} = \bigcup_{C \in \mathcal{C}} P_C \subseteq P_A$, and $\gamma \subseteq 2^{P_A}$ is a *data-agnostic interaction model*. We call ports in $P_A \setminus P_{\mathcal{C}}$ *dangling ports* of A.

Essentially, a BIP architecture is a structured way of combining an interaction model γ with a set of distinguished components, whose only purpose is to control which interactions in γ are applicable at which point in time (which depends on the states of the coordinating components).

Definition 2.1.3 (BIP architecture application [ABB+14])**.** Let $A = (\mathcal{C}, P_A, \gamma)$ be a BIP architecture, and $\mathcal{B}$ a set of components, such that $\mathcal{B} \cup \mathcal{C}$ is finite and disconnected, and that $P_A \subseteq P_{\mathcal{B}} \cup P_{\mathcal{C}}$. Write $\mathcal{B} \cup \mathcal{C} = \{B_i \mid i \in I\}$, with $B_i = (Q_i, q_i^0, P_i, \rightarrow_i)$. Then, the *application* $A(\mathcal{B})$ *of* A *to* $\mathcal{B}$ is the BIP component $(\prod_{i \in I} Q_i, (q_i^0)_{i \in I}, P_{\mathcal{B}} \cup P_{\mathcal{C}}, \rightarrow)$, where $\rightarrow$ is the smallest relation satisfying: $(q_i)_{i \in I} \xrightarrow{N} (q_i')_{i \in I}$ whenever

1. $N = \emptyset$, and there exists an $i \in I$ such that $q_i \xrightarrow{\emptyset}_i q_i'$ and $q_j' = q_j$ for all $j \in I \setminus \{i\}$; or

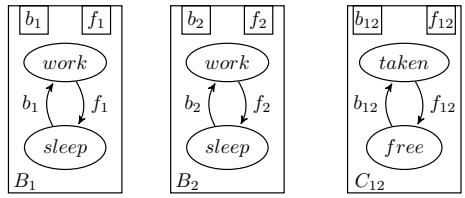


Figure 2.1: BIP components (a); the coordinating component (b) of the BIP architecture A_{12}.

2. $N \cap P_A \in \gamma$, and for all $i \in I$ we have $N \cap P_i \neq \emptyset$ implies $q_i \xrightarrow{N \cap P_i}_i q_i'$, and $N \cap P_i = \emptyset$ implies $q_i' = q_i$.

The application $A(\mathcal{B})$, of a BIP architecture A to a set of BIP components $\mathcal{B}$, enforces coordination constraints specified by that architecture on those components [ABB+14]. The *interface* P_A of A contains all ports $P_{\mathcal{C}}$ 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 participate only in 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.

Intuitively, an architecture can also be viewed as an incomplete system: the application of an architecture consists in "attaching" its dangling ports to the operand components. The operational semantics is that of composing all components (operands and coordinators) with the interaction model as described in the previous paragraph. The intuition behind transitions labelled by $\emptyset$ is that they represent *observable idling* (as opposed to internal transitions). This allows us to "desynchronize" combined architectures (see Definition 2.1.4) in a simple manner, since coordinators of one architecture can idle, while those of another performs a transition. Note that, if $N = \emptyset$, in item 2 of Definition 2.1.3, $N \cap P_i = \emptyset$, hence also, $q_i' = q_i$, for all i. Thus, intuitively, one can say that none of the components moves. Item 1, however, does allow one component to make a real move labelled by $\emptyset$, if such a move exists. Thus, the transitions labelled by $\emptyset$ interleave, reflecting the idea that in BIP synchronization can happen only through ports.

Example 2.1.1 (Mutual exclusion [ABB+14])**.** Consider the components B_1 and B_2 in Figure 2.1(a). In order to ensure mutual exclusion of their `work` states, we apply the BIP architecture $A_{12} = (\{C_{12}\}, P_{12}, \gamma_{12})$ with C_{12} from Figure 2.1(b), $P_{12} = \{b_1, b_2, b_{12}, f_1, f_2, f_{12}\}$ and $\gamma_{12} = \big\{\emptyset, \{b_1, b_{12}\}, \{b_2, b_{12}\}, \{f_1, f_{12}\}, \{f_2, f_{12}\}\big\}$. 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 that explicitly belong 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 (`taken`, `work`, `work`) is reachable, i.e. the mutual exclusion property $(q_1 \neq \texttt{work}) \vee (q_2 \neq \texttt{work})$—where q_1 and q_2 are state variables of B_1 and B_2 respectively—holds in $A_{12}(B_1, B_2)$. ◊

Definition 2.1.4 (Composition of BIP architectures [ABB+14])**.** Let $A_1 = (\mathcal{C}_1, P_1, \gamma_1)$ and $A_2 = (\mathcal{C}_2, P_2, \gamma_2)$ be two BIP architectures. Recall that $P_{\mathcal{C}_i} = \bigcup_{C \in \mathcal{C}_i} P_C$, for

$i = 1, 2$. If $P_{\mathcal{C}_1} \cap P_{\mathcal{C}_2} = \emptyset$, then $A_1 \oplus A_2$ is given by $(\mathcal{C}_1 \cup \mathcal{C}_2, P_1 \cup P_2, \gamma_{12})$, where $\gamma_{12} = \{N \subseteq P_1 \cup P_2 \mid N \cap P_i \in \gamma_i, \text{ for } i = 1, 2\}$. In other words, γ_{12} is the interaction model defined by the conjunction of the characteristic predicates of γ_1 and γ_2.

Data-sensitive semantics Recently, the data-agnostic formalization of BIP interaction models was extended with data transfer, using the notion of *interaction expressions* [BSBJ14].

Let $\mathcal{P}$ be a global set of ports. For each port $p \in \mathcal{P}$, let $x_p : \mathsf{D}_p$ be a typed variable used for the data exchange at that port. For a set of ports $P \subseteq \mathcal{P}$, let $X_P = (x_p)_{p \in P}$. An interaction expression models the effect of an interaction among ports in terms of the data exchanged through their corresponding variables.

Definition 2.1.5 (Interaction expression [BSBJ14])**.** An *interaction expression* is an expression of the form

$$(P \leftarrow Q).[g(X_Q, X_L) : \ (X_P, X_L) := up(X_Q, X_L) \,//\, (X_Q, X_L) := dn(X_P, X_L)]$$

where $P, Q \subseteq \mathcal{P}$ are *top* and *bottom* sets of ports; $L \subseteq \mathcal{P}$ is a set of *local* variables; $g(X_Q, X_L)$ is the boolean *guard*; $up(X_Q, X_L)$ and $dn(X_P, X_L)$ are respectively the *up-* and *downward data transfer* expressions.

For an interaction expression α as above, we define by $top(\alpha) = P$, $bot(\alpha) = Q$ and $supp(\alpha) = P \cup Q$ the sets of top, bottom and all ports in α, respectively. We denote g_α, up_α and dn_α the guard, upward and downward transfer corresponding expressions in α.

The first part of an interaction expression, $(P \leftarrow Q)$, describes the control flow as a dependency relation between the bottom and the top ports. The expression in the brackets describes the data flow, first "upward"—from bottom to top ports—and then "downward". The guard $g(X_Q, X_L)$ relates these two parts: interaction is enabled only when the values of the local variables together with those of variables associated to the bottom ports satisfy a boolean condition. As a side effect, an interaction expression may also modify local variables in X_L. Intuitively, such an interaction expression can *fire* only if its guard is true. When it fires, its upstream transfer is computed first using the values offered by its participating BIP components. Then, the downstream transfer modifies all of its port variables with updated values. These upstream and downstream data transfers execute atomically, which means that an interaction expression behaves as a stateless connector.

Definition 2.1.6 (BIP interaction models [BSBJ14])**.** A *(data-sensitive) BIP interaction model* is a set Γ of *simple BIP connectors* α that are BIP interaction expressions of the form

$$(\{w\} \leftarrow A).[g(X_A) : \ (x_w, X_L) := up(X_A) \,//\, X_A := dn(x_w, X_L)],$$

where $w \in P$ is a single top port, $A \subseteq P$ is a set of ports, such that $w \notin A$, and neither *up* nor g involves local variables.

Example 2.1.2 (Maximum)**.** Let $\mathcal{P} = \{a, b, w, l\}$ be a set of ports of type integer, i.e., $x_p : \mathsf{D}_p = \mathbb{Z}$, for all $p \in \mathcal{P}$, and consider the interaction expression (simple BIP connector)

$$\alpha_{\max} = (\{w\} \leftarrow \{a, b\}).[\mathtt{tt} : \ x_l := \max(x_a, x_b) \,//\, x_a, x_b := x_l],$$

where $\mathtt{tt}$ is true. First, the connector takes the values presented at ports a and b. Then, the simple BIP connector $\alpha_{\max}$ computes atomically the maximum of x_a and x_b and assigns it to its local variable x_l. Finally, $\alpha_{\max}$ assigns atomically the value of x_l to both x_a and x_b. ◊

BIP interaction expressions capture complete information about all aspects of component interaction—i.e., synchronization and data transfer possibilities—in a structured and concise manner. Thus, by examining interaction expressions, one can easily understand, on the one hand, the interaction model used to compose components and, on the other hand, how the valuations of data variables affect the enabledness of the interactions and how these valuations are modified. Furthermore, a formal definition of a composition operator on interaction expressions is provided in [BSBJ14], which allows combining such expressions hierarchically to manage the complexity of systems under design. Since any BIP system can be flattened, this hierarchical composition of interaction expressions is not relevant for the semantic comparison of BIP and Reo in this chapter. Nevertheless, the possibility of concisely capturing all aspects of component interaction in one place is rather convenient.

2.1.2 Reo

We briefly recall the basics of the Reo language and refer to [Arb04] and [Arb11] for further details. Reo is a coordination language wherein graph like structures express concurrency constraints (e.g., synchronization, exclusion, ordering, etc.) among multiple components. A Reo program, called a *connector*, is a graph-like structure whose edges consist of *channels* that enable synchronous and asynchronous data flow and whose vertices consist of *nodes* that synchronously route data among multiple channels.

A channel in Reo has exactly two *ends*, and each end either accepts data items, if it is a *source end*, or offers data items, if it is a *sink end*. The *type* of a channel is a formal constraint on the dataflow through its two ends that completely defines the behavior of the channel. Beside the established channel types (Figure 2.2 contains some of them) Reo allows arbitrary user-defined channel types.

Reo is agnostic regarding the semantics that expresses the behavior of its channel types, so long as the semantics preserves Reo's compositional construction principle (i.e., the behavior of a connector is computed by composing the behaviors of all channels and nodes). Jongmans [JA12] provides an overview of thirty alternative semantics for Reo channels. Its abstract definition of channels and its notion of channel types make Reo an extensible programming language.

Multiple ends may glue together into *nodes* with a fixed *merge-replicate* behavior: a data item out of a single sink end coincident on a node, atomically propagates to all source ends coincident on that node. This propagation happens only if all their respective channels allow the data exchange. A node is called a *source node*

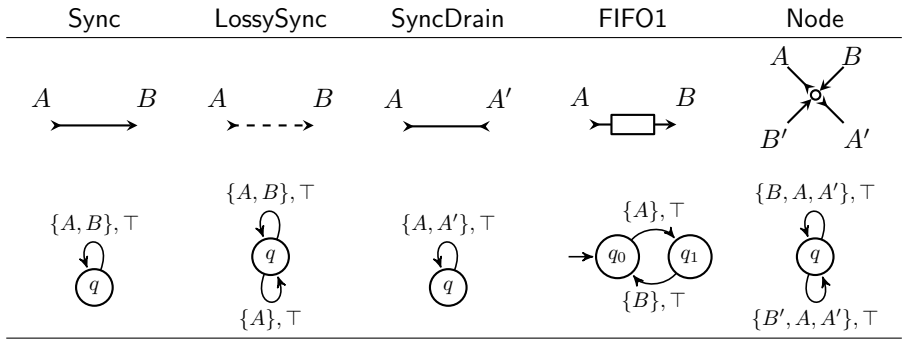


Figure 2.2: Some primitives in the Reo language with CA semantics over a singleton data domain $\mathcal{D}$.

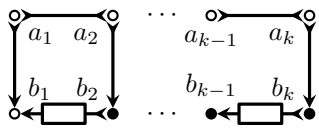


Figure 2.3: Construction of the $\mathsf{Alternator}_k$ Reo connector, for $k \geq 2$.

if it consists of source ends, a *sink node* if it consists of sink ends, and a *mixed node* otherwise. Together, the source and sink nodes of a connector constitute its set of *boundary nodes*.

Example 2.1.3 (Primitive channels)**.** Figure 2.2 shows some typical primitive Reo channels and an example of how these channels can compose at nodes.

A Sync channel accepts a datum from its source end A, when its simultaneous offer of this datum at its sink end B succeeds.

A SyncDrain channel simultaneously accepts a datum from both its source ends A and B and loses this datum.

An empty FIFO_1 accepts data from its source end A and becomes a full FIFO_1. A full FIFO_1 offers its stored data at its sink end B and, when its offer succeeds, it becomes an empty FIFO_1 again.

A Reo node accepts a datum from one of its coincident sink ends (B or B'), when its simultaneous offer to dispense a copy of this datum through every one of its coincident source ends (A and A') succeeds. ◊

The key concept in Reo is composition, which allows a programmer to build complex connectors out of simpler ones.

Example 2.1.4 (Alternator)**.** Using the channels in Figure 2.2, we can construct the $\mathsf{Alternator}_k$ connector, for $k \geq 2$, as shown in Figure 2.3. For $k = 2$, the $\mathsf{Alternator}_2$ consists of four nodes (a_1, a_2, b_1, and b_2) and four channels, namely a SyncDrain channel (between a_1 and a_2), two Sync channels (from a_1 to b_1, and from a_2 to b_2), and a FIFO_1 channel (from b_2 to b_1).

The behavior of the $\mathsf{Alternator}_2$ connector is as follows. Suppose that the environment is ready to offer a datum at each of the nodes a_1 and a_2, and ready

to accept a datum from node b_1. According to Example 2.1.3, nodes a_1 and a_2 both offer a copy of their received datum to the SyncDrain channel. The SyncDrain channel ensures that nodes a_1 and a_2 accept data from the environment only simultaneously. The Sync channel from a_1 to b_1 ensures that node b_1 simultaneously obtains a copy of the datum offered at a_1. By definition, node b_1 either accepts a datum from the connected Sync channel or it accepts a datum from the FIFO_1 channel (but not from both simultaneously), and offers this datum immediately to the environment. Because the FIFO_1 is initially empty, b_1 has no choice but to accept and dispense the datum from a_1. Simultaneously, the Sync channel from a_2 to b_2 ensures that the value offered at a_2 is stored in the FIFO_1 buffer. In the next step, the environment at node b_1 has no choice but to retrieve the datum in the buffer, after which the behavior repeats. ◊

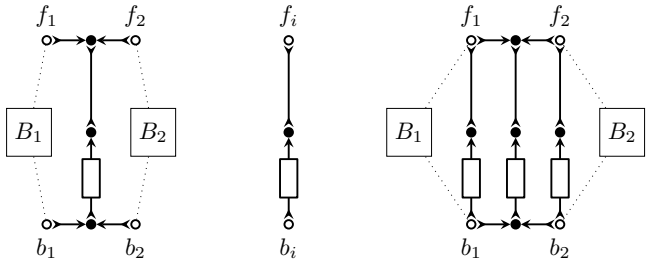
Example 2.1.5. Figure 2.4(a) shows a Reo connector that achieves mutual exclusion of components B_1 and B_2, exactly as the BIP system shown in Figure 2.1 does. This connector consists of a composition of channels and nodes in Figure 2.2. The Reo connector atomically accepts data from either b_1 or b_2 and puts it into the FIFO_1 channel, a buffer of size one. A full FIFO_1 channel means that B_1 or B_2 holds the lock. If one of the components writes to f_1 or f_2, the SyncDrain channel flushes the buffer, and the lock is released, returning the connector to its initial configuration, where B_1 and B_2 can again compete for exclusive access by attempting to write to b_1 or b_2.

The connector in Figure 2.4(a) is not fool-proof. Even if B_1 takes the lock, B_2 may release it, and vice versa. Hence, exactly as the BIP architecture in Figure 2.1, the Reo connector in Figure 2.4(a) relies on the conformance of the coordinated components B_1 and B_2. The expected behavior of B_i, $i = 1, 2$, is that it alternates writes on the b_i and f_i, and that every write on f_i comes after a write on b_i. Depending on such assumptions may not be ideal. The connector, shown in Figure 2.4(b), makes this expected behavior explicit. By composing two such connectors with the connector in Figure 2.4(a), we obtain a fool-proof mutual exclusion protocol, as shown in Figure 2.4(c). Figure 2.6(c) shows the constraint automaton semantics of the connector in Figure 2.4(c). Like the case of the connector in Figure 2.4(a) or the BIP architecture in Figure 2.1, noncompliant writes to b_i or f_i nodes of the connector in Figure 2.4(c) will *block* a renegade component B_i that attempts such writes. However, contrary to the case of the connector in Figure 2.4(a) or the BIP architecture in Figure 2.1, such a renegade component cannot *break* the mutual exclusion protocol that the connector in Figure 2.4(c) implements, as it allows the other component to run undisturbed. ◊

Formal semantics of Reo Reo has a variety of formal semantics [Arb11, JA12]. In this chapter we use its operational *constraint automaton* (CA) semantics [BSAR06].

Definition 2.1.7 (Constraint automata [BSAR06])**.** Let $\mathcal{N}$ be a set of ports and $\mathcal{D}$ a set of data items. A data constraint is a first-order formula g with constants $v \in \mathcal{D}$ and variables d_p, for $p \in \mathcal{N}$, that represent the datum observed at (i.e., exchanged through) port p. More formally, g is defined by the grammar

$$g \ ::= \ \top \mid \neg g \mid g \wedge g \mid \exists d_p(g) \mid d_p = v, \quad \text{with } p \in \mathcal{N}, v \in \mathcal{D},$$



(a) BIP-like mutex (b) (c) Fool-proof mutex

Figure 2.4: Fool-proof (c) mutual exclusion protocol in Reo, composed from a BIP-like (a) mutual exclusion connector and an alternator connector (b).

where $\top$, $\neg$, $\wedge$, $\exists$ and $=$ are respectively tautology, negation, conjunction, existential quantification and equality. Write $DC(\mathcal{N}, \mathcal{D})$ for the set of all data constraints over $\mathcal{N}$, and let $\models$ denote the usual satisfaction relation between data assignments $\delta : N \to \mathcal{D}$, with $N \subseteq \mathcal{N}$, and data constraints $g \in DC(\mathcal{N}, \mathcal{D})$. A constraint automaton (over data domain $\mathcal{D}$) is a tuple $\mathcal{A} = (Q, \mathcal{N}, \to, q_0)$ where Q is a set of states, $\mathcal{N}$ is a finite set of ports, $q_0 \in Q$ is the initial state, and $\to \subseteq Q \times 2^{\mathcal{N}} \times DC(\mathcal{N}, \mathcal{D}) \times Q$ is a transition relation, such that, for any transition $q \xrightarrow{N,g} q'$, we have $g \in DC(N, \mathcal{D})$.[3]

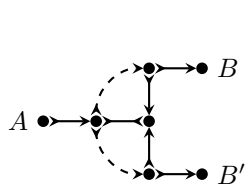
If a constraint automaton $\mathcal{A}$ has only one state, $\mathcal{A}$ is called *stateless*. If the data domain $\mathcal{D}$ of $\mathcal{A}$ is a singleton, $\mathcal{A}$ is called a *port automaton* [KC09]. In that case, we omit data constraints, because all satisfiable constraints reduce to $\top$.

In this chapter, we consider only finite data domains, although most of our results generalize to infinite data domains. Over a finite data domain, the data constraint language $DC(\mathcal{N}, \mathcal{D})$ is expressive enough to define any data assignment. For notational convenience, we relax, in this chapter, the definition of data constraints and allow the use of set-membership and functions in the data constraints (compare the definition of $g(\alpha)$ in Section 2.3.3). However, we preserve the intention that a data constraint describes a set of data assignments.

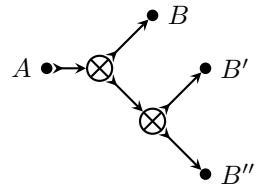
Example 2.1.6 (CA semantics of Reo primitives)**.** Figure 2.2 shows the CA semantics of some typical Reo primitives. Since constraint automata do not model the direction of dataflow, the CA semantics of Sync and SyncDrain coincides. $\Diamond$

Example 2.1.7 (Exclusive router)**.** The fixed merge-replicate behavior of a Reo node propagates an input datum to all of its output ports (i.e., source ends coincident on that node). An *exclusive router* is a connector that propagates an input datum to one of its, non-deterministically selected, output ports. Figure 2.5(a) shows the construction of a binary exclusive router from the primitive channels Sync, SyncDrain, and LossySync. Figure 2.5(b) shows the construction of a ternary

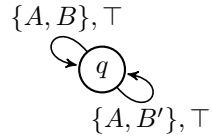
[3]The original definition of constraint automata excludes internal transitions with $\emptyset, \top$ labels [BSAR06]. If necessary, all internal transitions may be removed modulo (weak) language equivalence of constraint automata by merging any state q with every state q' that is reachable from q by a sequence of internal transitions.



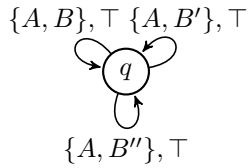
(a) Binary exclusive router



(b) Ternary exclusive router



(c) Semantics of (a)



(d) Semantics of (b)

Figure 2.5: Construction of a binary exclusive router (a); construction of a ternary exclusive router (b) from binary exclusive routers; and the CA semantics (c) and (d) of the exclusive routers in (a) and (b), respectively.

exclusive router by composing two binary exclusive routers, where we abbreviate a binary exclusive router as a crossed node. Figures 2.5(c) and 2.5(c) show the CA semantics of the binary and ternary exclusive router, respectively. ◊

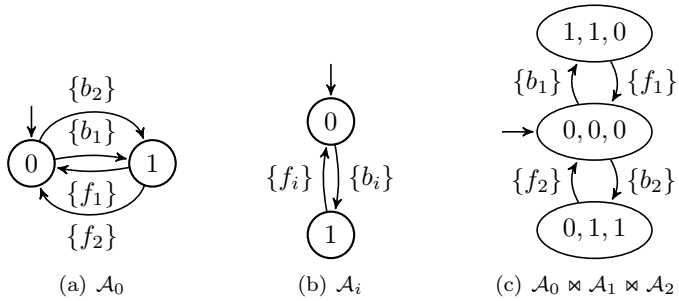
The CA semantics of every Reo connector can be derived as a composition of the constraint automata of its primitives, using the CA product operation in Definition 2.1.8.

The CA semantics for Reo connectors assigns a constraint automaton to every Reo connector. In the other direction, Baier et al. have shown that it is possible to translate every constraint automaton (over a finite data domain) back into a Reo connector [BKK14]. For example, Figure 2.8(c) shows the Reo connector that is generated from the constraint automaton $\mathsf{reo}_1(A_{12})$ in Figure 2.8(b). We refer to Example 2.2.1 for more details. Because of this correspondence, we consider Reo and CA as equivalent and focus on constraint automata only.

Definition 2.1.8 (Product of CA [BSAR06])**.** Let $\mathcal{A}_i = (Q_i, \mathcal{N}_i, \rightarrow_i, q_{0,i})$ be a constraint automaton, for $i = 1, 2$. Then the product $\mathcal{A}_1 \bowtie \mathcal{A}_2$ of these automata is the automaton $(Q_1 \times Q_2, \mathcal{N}_1 \cup \mathcal{N}_2, \rightarrow, (q_{0,1}, q_{0,2}))$, whose transition relation is the smallest relation obtained by the rule: $(q_1, q_2) \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} (q_1', q_2')$ whenever

1. $q_1 \xrightarrow{N_1, g_1}_1 q_1'$, $q_2 \xrightarrow{N_2, g_2}_2 q_2'$, and $N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1$, or
2. $q_i \xrightarrow{N_i, g_i}_i q_i'$, $N_j = \emptyset$, $g_j = \top$, $q_j' = q_j$, and $N_i \cap \mathcal{N}_j = \emptyset$ with $j \in \{1, 2\} \setminus \{i\}$.

It is not hard to see that constraint automata product operator is associative and commutative modulo equivalence of state names and data constraints (e.g., $d_p = v \wedge d_q = w$ is equivalent to $d_q = w \wedge d_p = v$, for $p, q \in \mathcal{N}$ and $v, w \in \mathcal{D}$).



(a) $\mathcal{A}_0$ (b) $\mathcal{A}_i$ (c) $\mathcal{A}_0 \bowtie \mathcal{A}_1 \bowtie \mathcal{A}_2$

Figure 2.6: CA semantics (a), (b), and (c) of Reo connectors in Figures 2.4(a), 2.4(b), and 2.4(c), respectively.

Definition 2.1.9 (Hiding in CA [BSAR06])**.** Let $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, q_0)$ be a constraint automaton, and $P = \{p_1, \ldots, p_n\}$ a set of ports. Then, hiding ports P of $\mathcal{A}$ yields an automaton $\exists P(\mathcal{A}) = (Q, \mathcal{N} \setminus P, \rightarrow_\exists, q_0)$, where $\rightarrow_\exists$ is given by $\{(q, N \setminus P, \exists d_{p_1} \cdots \exists d_{p_n}(g), q') \mid (q, N, g, q') \in \rightarrow\}$.

In addition to removing ports in P from the transition labels, the original definition of hiding merges any two states that become reachable by a sequence of internal $\emptyset$-labelled transitions (Definition 4.3 in [BSAR06] and Footnote 3). Since we allow these internal transitions, we do not bother to remove the internal transitions produced by the hiding operation in Definition 2.1.9. A constraint automaton obtained using our hiding operator is (weak) language equivalent to a constraint automaton obtained using the original hiding operator of [BSAR06].

As hiding of non-shared ports distributes over product, hiding of non-shared ports commutes with constraint automata product.

Example 2.1.8. Figures 2.6(a) and 2.6(b) show the constraint automaton semantics $\mathcal{A}_0$ and $\mathcal{A}_i$, for $i \in \{1, 2\}$, of the Reo connectors in Figures 2.4(a) and (two copies of) 2.4(b). Example 2.1.5 indicates that the fool-proof mutual exclusion protocol in Figure 2.4(c) can be obtain by composing the Reo connectors in Figures 2.6(a) and 2.6(b). Indeed, the constraint automaton semantics of the fool-proof mutual exclusion protocol in Figure 2.4(c) is given by $\mathcal{A} = \mathcal{A}_0 \bowtie \mathcal{A}_1 \bowtie \mathcal{A}_2$. The part of $\mathcal{A}$ that is reachable from initial state $(0, 0, 0)$ is shown in Figure 2.6(c). $\Diamond$

2.2 Port automata and BIP architectures

To study the relation between BIP and Reo with respect to synchronization, we start by defining a correspondence between them in the data-agnostic domain. This correspondence consists of a pair of mappings between the sets containing semantic models of BIP and Reo connectors. For the data independent semantic model of Reo connectors we choose port automata: a restriction of constraint automata over a singleton set as data domain. We model BIP connectors by BIP architectures introduced in [ABB+14]. In order to compare the behavior of BIP and Reo connectors we interpret them as labelled transition systems. We define a mapping reo_1 that transforms BIP architectures into port automata, and a mapping bip_1

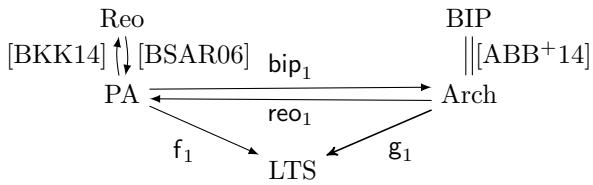


Figure 2.7: Translations and interpretations in the data-agnostic domain.

that transforms port automata into BIP architectures. We then show that these mappings preserve(1) properties closed under bisimulation, and (2) composition structure modulo semantic equivalence.

2.2.1 Interpretation of BIP and Reo

To compare the behavior of BIP and Reo connectors, we interpret all connectors as labelled transitions systems with one initial state and an alphabet 2^P, for a set of ports P. We write LTS for the class of all such labelled transition systems.

Figure 2.7 shows our translations and interpretations. The objects PA and Arch are the classes of port automata and BIP architectures, respectively. The mappings bip_1, reo_1, f_1 and g_1, respectively, translate Reo to BIP, BIP to Reo, Reo to LTS, and BIP to LTS.

We first consider the semantics of connectors in Reo and BIP. Since BIP connectors differ internally from Reo connectors, we restrict our interpretation to their observable behavior. This means that we hide the ports of the coordinating components in BIP architectures. For port automata this means that for our comparison, we implicitly assume that all ports correspond to boundary nodes only.

Interpretation of PA We define the interpretation of a port automaton as

$$\mathsf{f}_1((Q, \mathcal{N}, \to, q_0)) = (Q, 2^{\mathcal{N}}, \to, q_0). \tag{2.1}$$

Hence f_1 acts essentially as an identity function, justifying our choice of interpretation.

Interpretation of Arch We define the interpretation of BIP architectures using their operational semantics obtained by applying them on dummy components and hiding all internal ports. Let $A = (\mathcal{C}, P, \gamma)$ be a BIP architecture with coordinating components $\mathcal{C} = \{C_1, \ldots, C_n\}$, $n \geq 0$, and $C_i = (Q_i, q_i^0, P_i, \to_i)$. Recall that $P_{\mathcal{C}} = \bigcup_i P_i$ is the set of internal ports in A. Define $D = (\{q_D\}, q_D, P, \{(q_D, N, q_D) \mid \emptyset \neq N \subseteq P \setminus P_{\mathcal{C}}\})$ as a dummy component relative to the BIP architecture A. Using Definition 2.1.3, we compute the BIP architecture application $A(\{D\}) = ((\prod_{i=1}^n Q_i) \times \{q_D\}, (\mathbf{q}^0, q_D), P, \to_s)$ of A to its dummy component D. Then,

$$\mathsf{g}_1(A) = ((\textstyle\prod_{i=1}^n Q_i) \times \{q_D\}, 2^{P \setminus P_{\mathcal{C}}}, \to, (\mathbf{q}^0, q_D)) \tag{2.2}$$

where $\to = \{((\mathbf{q}, q_D), N \setminus P_{\mathcal{C}}, (\mathbf{q}', q_D)) \mid (\mathbf{q}, q_D) \xrightarrow{N}_s (\mathbf{q}', q_D)\}$. In other words, $\mathsf{g}_1(A)$ equals $A(\{D\})$ after hiding all internal ports $P_{\mathcal{C}}$.

Note that we based our interpretation g_1 on the operational semantics of BIP architectures, i.e., BIP architecture application. This justifies the definition of interpretation of architectures.

With a common semantics for BIP and Reo, we can define the notion of preservation of properties expressible in this common semantics. Recall that a property of labelled transition systems corresponds to the subset of labelled transition systems satisfying that property.

Definition 2.2.1. Let $P \subseteq \text{LTS}$ be a property. Then, bip_1 *preserves* P iff $\mathsf{f}_1(\mathcal{A}) \in P \Leftrightarrow \mathsf{g}_1(\mathsf{bip}_1(\mathcal{A})) \in P$ for all $\mathcal{A} \in \text{PA}$. Similarly, reo_1 *preserves* P iff $\mathsf{g}_1(A) \in P \Leftrightarrow \mathsf{f}_1(\mathsf{reo}_1(A)) \in P$ for all $A \in \text{Arch}$.

2.2.2 BIP to Reo

To translate BIP connectors to Reo connectors, we first determine what elements of BIP architectures correspond to Reo connectors. Our interpretations of port automata and BIP architectures show that dangling ports in BIP architectures correspond to boundary port names in port automata. Furthermore, the mutual exclusion of the interactions in an interaction model in a BIP architecture simulates mutually exclusive firing of transitions in port automata. The definition of a coordinating component in a BIP architecture is almost identical to that of a port automaton, yielding an obvious translation.

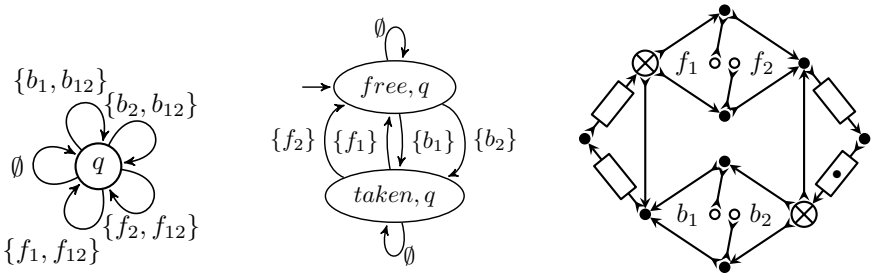
Let $A = (\mathcal{C}, P, \gamma)$ be a BIP architecture, with $\mathcal{C} = \{C_1, \ldots, C_n\}$. Each C_i corresponds trivially to a port automaton C_i^*. Let $\mathcal{A}_\gamma = (\{q\}, P, \to, q)$ be the stateless port automaton over P with transition relation $\to$ defined by $\{(q, N, q) \mid N \in \gamma\}$. Then $\mathcal{A}_\gamma$ can be seen as the port automata encoding of the interaction model γ. Recall that $P_\mathcal{C} = \bigcup_{C \in \mathcal{C}} P_C$. The corresponding port automaton of A is given by

$$\mathsf{reo}_1(A) = \exists P_\mathcal{C}(C_1^* \bowtie \cdots C_n^* \bowtie \mathcal{A}_\gamma). \tag{2.3}$$

Example 2.2.1. We translate the BIP architecture $A_{12} = (\{C_{12}\}, P_{12}, \gamma_{12})$ from Example 2.1.1 using reo_1 defined in Equation (2.3). First, we transform γ_{12} into a port automaton $\mathcal{A}_{\gamma_{12}}$, which is shown in Figure 2.8(a). Then, interpret the coordinating component C_{12} as a port automaton C_{12}^*. Finally, we compute the product of $\mathcal{A}_{\gamma_{12}}$ with the coordinating component C_{12}^* and hide the ports $\{b_{12}, f_{12}\}$ of C_{12}. Figure 2.8(b) shows the resulting port automaton.

As mentioned in Section 2.1.2, we can transform the port automaton in Figure 2.8(b) into a Reo connector, using the method described in [BKK14]. This mechanical translation yields the Reo connector in Figure 2.8(c)[4]. Intuitively, each state is represented by a FIFO buffer, and the current state is indicated by the presence of a token. A transition is represented by synchronous channels that move the token from one buffer to another. The transition is selected by an ternary exclusive router, represented as a crossed node (cf. Example 2.1.7). Note that the port automaton semantics of the connector in Figure 2.4(a) (see Figure 2.6(a)) is similar to the automaton in Figure 2.8(b), up to empty transitions. ◊

[4]For simplicity, we use two FIFO_1 buffers instead of simultaneous FIFO_1 buffers used in [BKK14].



(a) $\mathcal{A}_{\gamma_{12}}$ (b) $\mathsf{reo}_1(A_{12})$ (c) Generated mutex

Figure 2.8: Translation of the interaction model γ_{12} (a) and BIP architecture A_{12} (b) from Figure 2.1, and the Reo connector (c) generated from $\mathsf{reo}_1(A_{12})$.

2.2.3 Reo to BIP

In BIP, interaction is memoryless. This means that a stateful channel in Reo must translate to a coordinating component. In fact, we may encode an entire generic Reo connector as one such component.

The most natural way to translate a port automaton $\mathcal{A}$ into a BIP architecture A is by interpreting $\mathcal{A}$ as the coordinating component of A. However, BIP requires atomic components to synchronize via interactions, rather than directly on shared ports. Indeed, a BIP architecture excludes any two coordinating components to share a port (see Definition 2.1.2).

Since we want a compositional translation of port automata to BIP architectures, we need to interpret each port $p \in \mathcal{N}$ in the interface of $\mathcal{A}$ as a dangling port of A (see Definition 2.1.2). To this end, we rename every port $p \in \mathcal{N}$ in the interface of $\mathcal{A}$ to p', and synchronize p and p' by means of a BIP interaction.

Let $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, q_0)$ be a port automaton. We construct a corresponding BIP architecture for $\mathcal{A}$. Duplicate all ports in $\mathcal{N}$ by defining $\mathcal{N}' = \{n' \mid n \in \mathcal{N}\}$. We do not use a port n', for $n \in \mathcal{N}$, for composition with other BIP architectures. Therefore, the exact names of ports in an $\mathcal{N}'$ are not important, instead only their relation to their dangling siblings $n \in \mathcal{N}$ matters. For every $N \subseteq \mathcal{N}$, define $N' = \{n' \in \mathcal{N} \mid n \in N\}$. Trivially, $\overline{\mathcal{A}} = (Q, q_0, \mathcal{N}', \rightarrow_c)$, with $\rightarrow_c = \{(q, N', q') \mid (q, N, q') \in \rightarrow\}$, is a BIP component (cf. Definition 2.1.1). Essentially, $\mathcal{A}$ and $\overline{\mathcal{A}}$ are the same labelled transition system. Now we define bip_1 as follows:

$$\mathsf{bip}_1(\mathcal{A}) = (\{\overline{\mathcal{A}}\}, \mathcal{N} \cup \mathcal{N}', \{N \cup N' \mid N \subseteq \mathcal{N}\}). \tag{2.4}$$

Thus, bip_1 uses the port automaton as the coordinating component of the generated BIP architecture.

Example 2.2.2. We determine $\mathsf{bip}_1(\mathcal{A})$, where $\mathcal{A}$ is the port automaton in Figure 2.6(b) over the name set $\mathcal{N} = \{b_i, f_i\}$. Obtain $\overline{\mathcal{A}}$ by adding a prime to each port in $\mathcal{A}$. The interaction model of $\mathsf{bip}_1(\mathcal{A})$ consists of $\{N \cup N' \mid N \subseteq \mathcal{N}\} = \big\{\emptyset, \{b_i, b_i'\}, \{f_i, f_i'\}, \{b_i, b_i', f_i, f_i'\}\big\}$. Hence, $\mathsf{bip}_1(\mathcal{A})$ is given by the BIP architecture $(\{\overline{\mathcal{A}}\}, \{b_i, f_i, b_i', f_i'\}, \big\{\emptyset, \{b_i, b_i'\}, \{f_i, f_i'\}, \{b_i, b_i', f_i, f_i'\}\big\})$. ◊

2.2.4 Preservation of properties

To show that translations reo_1 and bip_1 preserve properties, we need to show that the diagram in Figure 2.7 commutes, i.e., $\mathsf{f}_1(\mathsf{reo}_1(A))$ is equivalent to $\mathsf{g}_1(A)$ and $\mathsf{g}_1(\mathsf{bip}_1(\mathcal{A}))$ is equivalent to $\mathsf{f}_1(\mathcal{A})$, for all $A \in \text{Arch}$ and $\mathcal{A} \in \text{PA}$.

The following examples show that this equivalence cannot be interpreted as equality or (strong) bisimulation.

Example 2.2.3. Consider the port automaton $\mathcal{A} = (\{q_0\}, \{a\}, \{(q_0, \{a\}, q_0)\}, q_0)$. The translation $\mathsf{bip}_1(\mathcal{A})$ of $\mathcal{A}$ into a BIP architecture is $(\{\overline{A}\}, \{a, a'\}, \{\emptyset, \{a, a'\}\})$, with coordinating component $\overline{A} = (\{q_0\}, q_0, \{a'\}, \{(q_0, \{a'\}, q_0)\})$. Since the interaction model of $\mathsf{bip}_1(\mathcal{A})$ contains the empty set, we find that the semantics $\mathsf{g}_1(\mathsf{bip}_1(\mathcal{A}))$ of $\mathsf{bip}_1(\mathcal{A})$ is given by $(\{q_0\}, 2^{\{a\}}, \{(q_0, \{a\}, q_0), (q_0, \emptyset, q_0)\}, q_0)$. On the other hand, the semantics $\mathsf{f}_1(\mathcal{A})$ of $\mathcal{A}$ does not admit an internal transition $(q_0, \emptyset, q_0)$, which shows that $\mathsf{g}_1(\mathsf{bip}_1(\mathcal{A}))$ and $\mathsf{f}_1(\mathcal{A})$ are not strongly bisimilar. $\Diamond$

Example 2.2.4. Consider the BIP architecture $A = (\{C_1, C_2\}, \emptyset, \emptyset)$ with coordinating components $C_i = (\{q_i, q_i'\}, q_i, \emptyset, \{(q_i, \emptyset, q_i')\})$, for $i = 1, 2$. Since the interaction model of A is empty, its translation $\mathcal{A}_\emptyset$ to a port automaton equals $(\{q_I\}, \emptyset, \emptyset, q_I)$. In addition, $P_{\{C_1,C_2\}} = \emptyset$, which shows that the translation of A to a port automaton equals $\mathsf{reo}_1(A) = \exists P_{\{C_1,C_2\}}(C_1^* \bowtie C_2^* \bowtie \mathcal{A}_\emptyset) = C_1^* \bowtie C_2^*$. Definition 2.1.8 shows that the semantics $\mathsf{f}_1(\mathsf{reo}_1(A))$ of $\mathsf{reo}_1(A)$ contains a transition $((q_1, q_2, q_I), \emptyset, (q_1', q_2', q_I))$.

Let $D = (\{q_D\}, q_D, \emptyset, \emptyset)$ be a dummy component relative to the BIP architecture A. Since BIP architecture application in Definition 2.1.3 requires state-changing internal (i.e., $\emptyset$-labelled) transitions to execute in isolation, we conclude that $A(\{D\})$ does not admit a transition $((q_1, q_2, q_D), \emptyset, (q_1', q_2', q_D))$. This shows that the semantics $\mathsf{g}_1(A)$ of A and $\mathsf{f}_1(\mathsf{reo}_1(A))$ are not strongly bisimilar. $\Diamond$

Since equality or (strong) bisimulation is a too strong semantic equivalence, we use the slightly weaker notion of equivalence called weak bisimulation [Mil89].

Definition 2.2.2 (Weak bisimulation [Mil89])**.** If $L_i = (Q_i, 2^{P_i}, \to_i, q_i^0) \in \text{LTS}$, $i = 1, 2$, then L_1 and L_2 are *weakly bisimilar* ($L_1 \cong L_2$) iff $P_1 = P_2$ and there exists $R \subseteq Q_1 \times Q_2$ such that $(q_1^0, q_2^0) \in R$ and$(q_1, q_2) \in R$ implies for all $N \in 2^{P_0} = 2^{P_1}$ and all $i, j \in \{1, 2\}$ with $i \neq j$, that

1. if $q_i \xrightarrow{\emptyset}_i q_i'$, then $q_j \ (\xrightarrow{\emptyset}_j)^* \ q_j'$ and $(q_1', q_2') \in R$, for some q_j'; and
2. if $q_i \xrightarrow{N}_i q_i'$ and $N \neq \emptyset$, then $q_j \ (\xrightarrow{\emptyset}_j)^* \xrightarrow{N}_j (\xrightarrow{\emptyset}_j)^* \ q_j'$ and $(q_1', q_2') \in R$, for some q_j'.

Definition 2.2.3 (Semantic equivalence)**.** Port automata $\mathcal{A}$ and $\mathcal{B}$ are *semantically equivalent* ($\mathcal{A} \sim \mathcal{B}$) iff $\mathsf{f}_1(\mathcal{A}) \cong \mathsf{f}_1(\mathcal{B})$. BIP architectures A and B are *semantically equivalent* ($A \sim B$) iff $\mathsf{g}_1(A) \cong \mathsf{g}_1(B)$.

Lemma 2.2.1. *Semantic equivalence of port automata satisfies the following properties: for all $\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2 \in \text{PA}$ we have*

1. *associativity:* $\mathcal{A}_0 \bowtie (\mathcal{A}_1 \bowtie \mathcal{A}_2) \sim (\mathcal{A}_0 \bowtie \mathcal{A}_1) \bowtie \mathcal{A}_2$
2. *commutativity:* $\mathcal{A}_0 \bowtie \mathcal{A}_1 \sim \mathcal{A}_1 \bowtie \mathcal{A}_0$

3. congruence: $\mathcal{A}_0 \sim \mathcal{A}_1$ *implies* $\mathcal{A}_0 \bowtie \mathcal{A}_2 \sim \mathcal{A}_1 \bowtie \mathcal{A}_2$.

Proof. Consider (strong) bisimulation of port automata(i.e., constraint automata all of whose data constraints are $\top$) as defined in [BSAR06]. Composition of port automata is commutative and associative up to bisimulation [BSAR06]. Since f_1 acts like the identity and every (strong) bisimulation is also a weak bisimulation, we conclude that composition of port automata is commutative and associative modulo semantic equivalence.

Since f_1 acts as the identity and every (strong) bisimulation is also a weak bisimulation, we conclude that semantic equivalence of port automata corresponds to weak bisimulation of port automata. Let Q_0, Q_1 and Q_2 be the state spaces of $\mathcal{A}_0$, $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. Suppose that $R \subseteq Q_0 \times Q_1$ is a weak bisimulation between $\mathcal{A}_0$ and $\mathcal{A}_1$. Using Definition 2.1.8, it follows that $R' = \{((q_0, q_2), (q_1, q_2')) \mid (q_0, q_1) \in R \text{ and } q_2 = q_2'\} \subseteq (Q_0 \times Q_2) \times (Q_1 \times Q_2)$ is a weak bisimulation between $\mathcal{A}_0 \bowtie \mathcal{A}_2$ and $\mathcal{A}_1 \bowtie \mathcal{A}_2$. □

Theorem 2.2.2. *For all* $\mathcal{A} \in \mathrm{PA}$ *and* $A \in \mathrm{Arch}$ *we have* $\mathsf{g}_1(\mathsf{bip}_1(\mathcal{A})) \cong \mathsf{f}_1(\mathcal{A})$ *and* $\mathsf{f}_1(\mathsf{reo}_1(A)) \cong \mathsf{g}_1(A)$.

Proof. First, we show that $\mathsf{g}_1(\mathsf{bip}_1(\mathcal{A})) \cong \mathsf{f}_1(\mathcal{A})$ for all port automata $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, q_0) \in \mathrm{PA}$. The state space of $\mathsf{g}_1(\mathsf{bip}_1(\mathcal{A}))$ is $Q \times \{q_D\}$, where q_D is the state of the dummy component, and the state space of $\mathsf{f}_1(\mathcal{A})$ is Q. We show that $\sim$ given by $(q, q_D) \sim q$ for all $q \in Q$ is a weak bisimulation.

Trivially, $(q_0, q_D) \sim q_0$. Suppose that $((q, q_D), N, (q', q_D))$ is a transition in $\mathsf{g}_1(\mathsf{bip}_1(\mathcal{A}))$. We show that either $N = \emptyset$ and $q' = q$, or there exists a transition (q, N, q') in $\mathsf{f}_1(\mathcal{A})$ with $(q', q_D) \sim q'$. Using the shape of the interaction model γ, we obtain a transition $((q, q_D), N \cup N', (q', q_D))$ in $\mathsf{bip}_1(\mathcal{A})(\{D\})$, with $N' = \{n' \mid n \in N\}$. Definition 2.1.3, with $\mathcal{C} = \{\overline{\mathcal{A}}\}$ and $\mathcal{B} = \{D\}$, shows that either

1a) $N \cup N' = \emptyset$, $(q, \emptyset, q')$ is a transition in $\overline{\mathcal{A}}$, and $q_D = q_D$; or

1b) $N \cup N' = \emptyset$, $(q_D, \emptyset, q_D)$ is a transition in D, and $q' = q$; or

2) $N \cup N' \in \gamma_{\mathsf{bip}_1(\mathcal{A})}$, and if $N' \neq \emptyset$ then (q, N', q') is a transition in $\overline{\mathcal{A}}$, and if $N' = \emptyset$ then $q' = q$, and if $N \neq \emptyset$ then (q_D, N, q_D) is a transition in D, and if $N = \emptyset$ then $q_D = q_D$.

If (1a) holds, then $N = \emptyset$, and by the definition of f_1 we find a transition (q, N, q') in $\mathsf{f}_1(\mathcal{A})$. Trivially, $(q', q_D) \sim q'$. Case (1b) is impossible, since dummy component D does not have an empty transition. Suppose that (2) holds. If $N = \emptyset$, then we have $q' = q$. If $N \neq \emptyset$, then the definition of f_1 gives a (q, N, q') in $\mathsf{f}_1(\mathcal{A})$, and trivially we have $(q', q_D) \sim q'$. Thus, in each case, either $N = \emptyset$ and $q' = q$, or there exists a transition (q, N, q') in $\mathsf{f}_1(\mathcal{A})$ with $(q', q_D) \sim q'$.

On the other hand, let (q, N, q') be a transition in $\mathsf{f}_1(\mathcal{A})$. We show that there exists a transition $((q, q_D), N, (q', q_D))$ in $\mathsf{g}_1(\mathsf{bip}_1(\mathcal{A}))$. Using the definition of f_1, we find that (q, N', q') is a transition in $\overline{\mathcal{A}}$, with $N' = \{n' \mid n \in N\}$. If $N = \emptyset$, then the first rule of Definition 2.1.3 implies that $((q, q_D), N \cup N', (q', q_D))$ is a transition in $\mathsf{bip}_1(\mathcal{A})(\{D\})$. If $N \neq \emptyset$, then we have that (q_D, N, q_D) is a transition in the dummy component D of the BIP architecture application $\mathsf{bip}_1(\mathcal{A})(\{D\})$. The second rule of Definition 2.1.3 implies that $((q, q_D), N \cup N', (q', q_D))$ is a transition

in $\mathsf{bip}_1(\mathcal{A})(\{D\})$. In either case, we find that $((q, q_D), N, (q', q_D))$ is a transition in $\mathsf{g}_1(\mathsf{bip}_1(\mathcal{A}))$ and trivially that $(q', q_D) \sim q'$. Thus, $\sim$ is a weak bisimulation between $\mathsf{g}_1(\mathsf{bip}_1(\mathcal{A}))$ and $\mathsf{f}_1(\mathcal{A})$.

Second, We show that $\mathsf{f}_1(\mathsf{reo}_1(A)) \cong \mathsf{g}_1(A)$ for any BIP architecture $A = (\{C_i\}_{i\in I}, P, \gamma)$ with components given by $C_i = (Q_i, q_i^0, P_i, \rightarrow_i)$, for all $i \in I$. The state space of $\mathsf{f}_1(\mathsf{reo}_1(A))$ is $(\prod_{i\in I} Q_i) \times \{q_I\}$, where q_I is the state of the port automaton of the interaction model of A. The state space of $\mathsf{g}_1(A)$ is $(\prod_{i\in I} Q_i)\times\{q_D\}$, where q_D is the state of the dummy component. We show that $\sim$ given by $(\mathbf{q}, q_I) \sim (\mathbf{q}, q_D)$ for all $\mathbf{q} = (q_i)_{i\in I} \in \prod_{i\in I} Q_i$, is a weak bisimulation.

Trivially, $(\mathbf{q}^0, q_I) \sim (\mathbf{q}^0, q_D)$. Let $((\mathbf{q}, q_D), N, (\mathbf{q}', q_D))$ be a transition in $\mathsf{g}_1(A)$, for some $N \subseteq P\backslash P_{\mathcal{C}}$. We show that $((\mathbf{q}, q_I), N, (\mathbf{q}', q_I))$ is a transition in $\mathsf{f}_1(\mathsf{reo}_1(A))$. The definition of g_1 shows that there exists some $M \subseteq P$, with $M \setminus P_{\mathcal{C}} = N$, such that $((\mathbf{q}, q_D), M, (\mathbf{q}', q_D))$ is a transition in $A(\{D\})$, where D is the dummy component of A. Definition 2.1.3 implies that either

1a) $M = \emptyset$, $(q_i, \emptyset, q_i') \in \rightarrow_i$ and $q_j' = q_j$, for some $i \in I$ and all $j \in I \setminus \{i\}$; or

1b) $M = \emptyset$, $(q_D, \emptyset, q_D)$ is a transition in D, and $q_j' = q_j$ for all $j \in I$; or

2) $M \in \gamma$, and if $M \cap P_i \neq \emptyset$ then $(q_i, M \cap P_i, q_i') \in \rightarrow_i$, and if $M \cap P_i = \emptyset$ then $q_i' = q_i$, for all $i \in I$.

If (1a), then $(q_i, \emptyset, q_i')$ is a transition in C_i^*. Hence, the second item in Definition 2.1.8 gives a transition $((\mathbf{q}, q_I), N, (\mathbf{q}', q_I))$ in $\mathsf{f}_1(\mathsf{reo}_1(A))$, with $N \subseteq M = \emptyset$. Case (1b) is impossible, since dummy component D does not have an empty transition. If (2), then $M \in \gamma$ implies $(q_I, M, q_I) \in \mathcal{A}_\gamma$. Using Definition 2.1.8 and $M \setminus P_{\mathcal{C}} = N$, we find a transition $((\mathbf{q}, q_I), N, (\mathbf{q}', q_I))$ in $\mathsf{f}_1(\mathsf{reo}_1(A))$.

Let $((\mathbf{q}, q_I), N, (\mathbf{q}', q_I))$ be a transition in $\mathsf{f}_1(\mathsf{reo}_1(A))$, for some $N \subseteq P \setminus P_{\mathcal{C}}$. We show that there exist a sequence of transitions $(\mathbf{q}, q_I)$ $(\xrightarrow{\emptyset})^*$ $\xrightarrow{N}$ $(\mathbf{q}', q_I)$ in $\mathsf{g}_1(A)$. The definition of reo_1 shows that there exists some $M \subseteq P$ such that $M \setminus P_{\mathcal{C}} = N$ and $((\mathbf{q}, q_I), M, (\mathbf{q}', q_I))$ is a transition in $C_1^* \bowtie \cdots C_n^* \bowtie \mathcal{A}_\gamma$. According to Definition 2.1.8, we find that either

1) $(\mathbf{q}, M, \mathbf{q}')$ and (q_I, M, q_I) are transitions in $C_1^* \bowtie \cdots \bowtie C_n^*$ resp. $\mathcal{A}_\gamma$; or

2a) $(\mathbf{q}, M, \mathbf{q}')$ is a transition in $C_1^* \bowtie \cdots \bowtie C_n^*$ and $M \cap P = \emptyset$; or

2b) (q_I, M, q_I) is a transition in $\mathcal{A}_\gamma$, $M \cap P_{\mathcal{C}} = \emptyset$ and $\mathbf{q}' = \mathbf{q}$.

If (1) holds, then $M \in \gamma$, and, for each $i \in I$, we have either $M \cap P_i = \emptyset$ and $q_i' = q_i$ or we find a transition $(q_i, M \cap P_i, q_i')$ in C_i^*. Definition 2.1.3 requires a transition $(q_i, M \cap P_i, q_i')$ in C_i^* that satisfies both $M \cap P_i = \emptyset$ and $q_i' \neq q_i$ to execute in isolation. Therefore, Definition 2.1.3 yields a sequence of transitions $(\mathbf{q}, q_I)$ $(\xrightarrow{\emptyset}$ $)^*$ $(\overline{\mathbf{q}}, q_I)$ $\xrightarrow{N}$ $(\mathbf{q}', q_I)$ in $\mathsf{g}_1(A)$, where $\overline{q_i} = q_i'$, if $M \cap P_i = \emptyset$ and $q_i' \neq q_i$, and $\overline{q_i} = q_i$ otherwise. If (2a) holds, then $N \subseteq M = M \cap P = \emptyset$ and, by Definition 2.1.8, we have for some $i \in I$ that $(q_i, \emptyset, q_i')$ is a transition in C_i^*. Similar to case(1), we obtain a non-empty sequence of transitions $(\mathbf{q}, q_I)$ $(\xrightarrow{\emptyset})^+$ $(\mathbf{q}', q_I)$ in $\mathsf{g}_1(A)$. If (2b) holds, then we have $N = M \in \gamma$, and Definition 2.1.3 shows that there exist a transition $(\mathbf{q}, q_I)$ $\xrightarrow{N}$ $(\mathbf{q}', q_I)$ in $\mathsf{g}_1(A)$. In each case, we found a sequence of

transitions $(\mathbf{q}, q_I)$ $(\xrightarrow{\emptyset})^*$ $\xrightarrow{N}$ $(\mathbf{q}', q_I)$ in $\mathsf{g}_1(A)$, and $(\mathbf{q}', q_I) \sim (\mathbf{q}', q_D)$. Thus, $\sim$ is a weak bisimulation between $\mathsf{f}_1(\mathsf{reo}_1(A))$ and $\mathsf{g}_1(A)$. □

Corollary. bip_1 *and* reo_1 *preserve all properties closed under weak bisimulation, i.e., for all* $P \subseteq$ LTS, $\mathcal{A} \in$ PA *and* $A \in$ Arch *we have* $\mathsf{f}_1(\mathcal{A}) \in P \Leftrightarrow \mathsf{g}_1(\mathsf{bip}_1(\mathcal{A})) \in P$ *and* $\mathsf{g}_1(A) \in P \Leftrightarrow \mathsf{f}_1(\mathsf{reo}_1(A)) \in P$, *whenever* $L \in P$ *and* $L' \cong L$ *implies* $L' \in P$, *for all* $L, L' \in$ LTS.

Section 2.2.4 allows model checking of BIP architectures with Reo model checkers, and vice versa. This is particularly interesting, since tools for BIP and Reo employ different model checking techniques. For example, the D-Finder tool allows for compositional deadlock detection and verification of BIP systems [bip16], while Vereofy allows for linear and branching time model checking of Reo systems [reo16].

Example 2.2.5. Consider the following safety property φ satisfied by the Reo connector in Figure 2.4(c): "if b_1 fires, then b_2 fires only after f_1 fires". The automaton $\mathcal{A}$ in Figure 2.6(c) clearly satisfies this property. Using Section 2.2.4, we conclude that the BIP architecture $\mathsf{bip}_1(\mathcal{A})$ satisfies φ also. ◊

2.2.5 Compatibility with composition

BIP architectures and port automata have their own notions of composition. We show that, under some mild conditions, our translations preserve composition modulo semantic equivalence.

Recall the port automaton representation of the interaction model from Section 2.2.2. The following lemma provides a decomposition of the port automaton representation of the interaction model of a composed BIP architecture.

Lemma 2.2.3. *Let* $A_i = (\mathcal{C}_i, P_i, \gamma_i) \in$ Arch, $i = 1, 2$, *with* $P_{\mathcal{C}_1} \cap P_{\mathcal{C}_2} = \emptyset$ *and* $\emptyset \in \gamma_1 \cap \gamma_2$. *Then, we have that* $\mathcal{A}_{\gamma_{12}} \sim \mathcal{A}_{\gamma_1} \bowtie \mathcal{A}_{\gamma_2}$, *where* γ_{12} *is the interaction model of* $A_1 \oplus A_2$.

Proof. Let (q, N, q) be a transition in $\mathcal{A}_{\gamma_{12}}$. By definition, $N \in \gamma_{12}$, and from Definition 2.1.4 we deduce $N \cap P_i \in \gamma_i$, $i = 1, 2$. Therefore $(q, N \cap P_i, q)$ is a transition in $\mathcal{A}_{\gamma_i}$. Then, Definition 2.1.8, implies that $((q, q), N, (q, q))$ in $\mathcal{A}_{\gamma_1} \bowtie \mathcal{A}_{\gamma_2}$. On the other hand, suppose that $((q, q), N, (q, q))$ is a transition in $\mathcal{A}_{\gamma_1} \bowtie \mathcal{A}_{\gamma_2}$. Then, Definition 2.1.8 gives either that (1) for $i = 1, 2$, $(q, N \cap P_i, q)$ is a transition in $\mathcal{A}_{\gamma_i}$, or (2) for $i, j \in \{1, 2\}$, $i \neq j$, $(q, N \cap P_i, q)$ is a transition in $\mathcal{A}_{\gamma_i}$ and $N \cap P_j = \emptyset$. In the first case, we conclude that $N \cap P_i \in \gamma_i$, for $i = 1, 2$. Hence, Definition 2.1.4 implies $N \in \gamma_{12}$. In the second case, we see that $N \cap P_i \in \gamma_i$ and $N \cap P_j = \emptyset \in \gamma_j$, since $\emptyset \in \gamma_1 \cap \gamma_2$. Thus, Definition 2.1.4 implies $N \in \gamma_{12}$. In both cases we find $N \in \gamma_{12}$, and we conclude that (q, N, q) is a transition of $\mathcal{A}_{\gamma_{12}}$. □

For any two BIP architectures $A_1, A_2 \in$ Arch, consider the equation

$$\mathsf{reo}_1(A_1 \oplus A_2) \sim \mathsf{reo}_1(A_1) \bowtie \mathsf{reo}_1(A_2), \tag{2.5}$$

Recall that reo_1 hides all internal ports $P_{\mathcal{C}_1 \cup \mathcal{C}_2}$ of $A_1 \oplus A_2$, where, for $i \in \{1, 2\}$, $\mathcal{C}_i$ is the set of coordinating components of A_i. This means that internal ports $P_{\mathcal{C}_1 \cup \mathcal{C}_2}$ in $A_1 \oplus A_2$ cannot be used for composition in the right hand side of equation

Equation (2.5). In particular, the BIP architectures cannot share any internal port in $P_{\mathcal{C}_1 \cup \mathcal{C}_2} = P_{\mathcal{C}_1} \cup P_{\mathcal{C}_2}$. Therefore, we need to assume that $P_{\mathcal{C}_1} \cap P_2 = P_{\mathcal{C}_2} \cap P_1 = \emptyset$, where, for $i \in \{1, 2\}$, P_i is the interface of A_i.

Note that shared internal ports can be transformed into shared dangling ports. Let $p \in P_{\mathcal{C}_1} \cap P_2$ be a dangling port of P_2 that is connected to a component in A_1. Change A_1 to A_1' by adding a (dangling) port x to A_1 and synchronizing p with x by changing the BIP interaction model γ_1 of A_1 to $\gamma_1' = \{N \cup \{x\} \mid p \in N \in \gamma_1\} \cup \{N \mid p \notin N \in \gamma_1\}$. Change A_2 to A_2' by renaming p to x in A_2. The resulting architectures A_1' and A_2' satisfy the assumption. This construction shows that $P_{\mathcal{C}_1} \cap P_2 = P_{\mathcal{C}_2} \cap P_1 = \emptyset$ is only a mild assumption.

Theorem 2.2.4. $\mathsf{reo}_1(A_1 \oplus A_2) \sim \mathsf{reo}_1(A_1) \bowtie \mathsf{reo}_1(A_2)$ *for all* $A_i = (\mathcal{C}_i, P_i, \gamma_i) \in$ Arch*, with* $P_{\mathcal{C}_1} \cap P_2 = P_{\mathcal{C}_2} \cap P_1 = \emptyset$ *and* $\emptyset \in \gamma_1 \cap \gamma_2$*.*

Proof. Let $\mathcal{C}_1 \cup \mathcal{C}_2 = \{C_1, \ldots, C_n, \ldots, C_m\}$, with $C_i \in \mathcal{C}_1$ iff $i \leq n$, be the set of coordinating components of A_1 and A_2. By definition, we have $\mathsf{reo}_1(A_1 \oplus A_2) = \exists P_{\mathcal{C}_1 \cup \mathcal{C}_2}(C_1^* \bowtie \cdots C_n^* \bowtie C_{n+1}^* \bowtie \cdots C_m^* \bowtie \mathcal{A}_{\gamma_{12}})$. Using Lemmas 2.2.1 and 2.2.3, we obtain $\mathsf{reo}_1(A_1 \oplus A_2) \sim \exists P_{\mathcal{C}_1} \exists P_{\mathcal{C}_2}(C_1^* \bowtie \cdots C_n^* \bowtie \mathcal{A}_{\gamma_1} \bowtie C_{n+1}^* \bowtie \cdots C_m^* \bowtie \mathcal{A}_{\gamma_2})$. From $P_{\mathcal{C}_1} \cap P_2 = P_{\mathcal{C}_2} \cap P_1 = \emptyset$, we conclude that the port automata $C_1^*, \ldots, C_n^*$ and $\mathcal{A}_{\gamma_1}$ do not use ports from $P_{\mathcal{C}_2}$. Since hiding of non-shared ports distributes over composition of port automata, we find that

$$\mathsf{reo}_1(A_1 \oplus A_2) \sim \exists P_{\mathcal{C}_1}(C_1^* \bowtie \cdots C_n^* \bowtie \mathcal{A}_{\gamma_1}) \bowtie \exists P_{\mathcal{C}_2}(C_{n+1}^* \bowtie \cdots C_m^* \bowtie \mathcal{A}_{\gamma_2}).$$

Hence, we conclude that $\mathsf{reo}_1(A_1 \oplus A_2) \sim \mathsf{reo}_1(A_1) \bowtie \mathsf{reo}_1(A_2)$. □

Theorem 2.2.5. $\mathsf{bip}_1(\mathcal{A}_1 \bowtie \mathcal{A}_2) \sim \mathsf{bip}_1(\mathcal{A}_1) \oplus \mathsf{bip}_1(\mathcal{A}_2)$ *for all* $\mathcal{A}_i \in$ PA*.*

Proof. Applying Theorem 2.2.4, with $A_1 = \mathsf{bip}_1(\mathcal{A}_1)$ and $A_2 = \mathsf{bip}_1(\mathcal{A}_2)$, gives that $\mathsf{reo}_1(\mathsf{bip}_1(\mathcal{A}_1) \oplus \mathsf{bip}_1(\mathcal{A}_2)) \sim \mathsf{reo}_1(\mathsf{bip}_1(\mathcal{A}_1)) \bowtie \mathsf{reo}_1(\mathsf{bip}_1(\mathcal{A}_2))$. Using Theorem 2.2.2, we find, for any $\mathcal{B} \in$ PA, that $\mathsf{f}_1(\mathsf{reo}_1(\mathsf{bip}_1(\mathcal{B}))) \cong \mathsf{g}_1(\mathsf{bip}_1(\mathcal{B})) \cong \mathsf{f}_1(\mathcal{B})$ and $\mathsf{reo}_1(\mathsf{bip}_1(\mathcal{B})) \sim \mathcal{B}$. Since semantic equivalence is a congruence by Lemma 2.2.1, we find that $\mathsf{reo}_1(\mathsf{bip}_1(\mathcal{A}_1) \oplus \mathsf{bip}_1(\mathcal{A}_2)) \sim \mathcal{A}_1 \bowtie \mathcal{A}_2 \sim \mathsf{reo}_1(\mathsf{bip}_1(\mathcal{A}_1 \bowtie \mathcal{A}_2))$. By Theorem 2.2.2, we conclude that $\mathsf{bip}_1(\mathcal{A}_1) \oplus \mathsf{bip}_1(\mathcal{A}_2) \sim \mathsf{bip}_1(\mathcal{A}_1 \bowtie \mathcal{A}_2)$ □

Example 2.2.6. For any two ports x and y, let $\mathcal{A}_{\{x,y\}}$ be the port automaton of a synchronous channel (cf. Figure 2.2), and let $C_{\{x,y\}}$ be its corresponding BIP component. Suppose we need to translate $\mathcal{A}_{\{a,b\}} \bowtie \mathcal{A}_{\{b,c\}}$ to a BIP architecture. Then, we compute $\mathsf{bip}_1(\mathcal{A}_{\{a,b\}}) = (\{C_{\{a',b'\}}\}, \{a, a', b, b'\}, \gamma_{\{a,b\}})$, with

$$\gamma_{\{a,b\}} = \{\emptyset, \{a, a'\}, \{b, b'\}, \{a, a', b, b'\}\}.$$

Next, we compute $\mathsf{bip}_1(\mathcal{A}_{\{b,c\}}) = (\{C_{\{b'',c''\}}\}, \{b, b'', c, c''\}, \gamma_{\{b,c\}})$, with

$$\gamma_{\{b,c\}} = \{\emptyset, \{b, b''\}, \{c, c''\}, \{b, b'', c, c''\}\}.$$

Note that we need to use double primes now, because otherwise b' would be a shared port of $C_{\{a',b'\}}$ and $C_{\{b'',c''\}}$. Using Theorem 2.2.5, we find that $\mathsf{bip}_1(\mathcal{A}_{\{a,b\}} \bowtie \mathcal{A}_{\{b,c\}}) = \mathsf{bip}_1(\mathcal{A}_{\{a,b\}}) \oplus \mathsf{bip}_1(\mathcal{A}_{\{b,c\}})$. Therefore, $\mathcal{A}_{\{a,b\}} \bowtie \mathcal{A}_{\{b,c\}}$ translates to

$$(\{C_{\{a',b'\}}, C_{\{b'',c''\}}\}, \{a, a', b, b', b'', c, c''\}, \gamma_{\{a,b,c\}}),$$

where $\gamma_{\{a,b,c\}}$ is the composition of $\gamma_{\{a,b\}}$ and $\gamma_{\{b,c\}}$. ◊

Example 2.2.7. Consider the port automaton $\mathcal{A}$ from Figure 2.6(c). If we translate $\mathcal{A}$ to BIP, we obtain a BIP architecture $B_1 = \mathsf{bip}_1(\mathcal{A})$, which has only a single coordinating component. From Example 2.1.8, we see that $\mathcal{A} \cong \mathcal{A}_0 \bowtie \mathcal{A}_1 \bowtie \mathcal{A}_2$, where $\mathcal{A}_0$ is the port automaton in Figure 2.6(a), and $\mathcal{A}_i$ is the port automaton in Figure 2.6(b), for $i = 1, 2$. Now consider $B_3 = \mathsf{bip}_1(\mathcal{A}_0) \oplus \mathsf{bip}_1(\mathcal{A}_1) \oplus \mathsf{bip}_1(\mathcal{A}_2)$. Using Definition 2.1.4, we see that B_3 has three coordinating components. Nevertheless, Theorem 2.2.5 shows that B_3 is semantically equivalent to B. Therefore, Theorem 2.2.5 allows to compute translations compositionally. ◊

2.3 Stateless CA's and interaction models

In Section 2.2, we established a correspondence between port automata and BIP architectures. Here, we offer translations between data-sensitive connector models in BIP and Reo.

For BIP connectors we use BIP interaction models, which are tuples consisting of an interface P and a set Γ of interaction expressions α that have:

1. a single top port that is not a bottom port,
2. bottom ports included in their interface P, and
3. guard and up functions that are independent of local variables (Definition 2.1.5).

We assume that every top port occurs only in one interaction expression per BIP interaction model. We denote the class of such BIP interaction models by IM.

For the semantics of Reo connectors, we take a pair consisting of a constraint automaton and a partition of its interface into *input* ports $\mathcal{N}_{in}$ and *output* ports $\mathcal{N}_{out}$[5]. We call such pairs *constraint automata with polarity*. The reason we explicitly distinguish CA port types in this semantics is to give direction to dataflow, similar to BIP connectors. Usually such port type distinctions are implicit within the semantics of Reo connectors, but for preciseness we encode them here as a partition.

A full correspondence of BIP interaction models and constraint automata with polarity in Reo is not possible. Firstly, BIP interaction models are stateless, we need to restrict ourselves here to only stateless constraint automata with polarity [ABB+14, BSBJ14]. Secondly, ports of a BIP interaction expression are *bidirectional* in the sense that input and output through a port happen simultaneously in a single execution step. Ports in a Reo connector are *unidirectional* in the sense that each port is either an input port or an output port. To accommodate this distinction, we split every bidirectional port p in a BIP interaction expression into an input port $p!$, providing write operations to the user of the connector, and an output port $p?$, providing read operations to the user of the connector. Therefore, we consider the class $\mathrm{CA}^{\pm}$ of all stateless constraint automata with polarity, such that, for some set of BIP ports P, we have the set of Reo ports $\mathcal{N}_{in} = \{p! \mid p \in P\}$, $\mathcal{N}_{out} = \{p? \mid p \in P\}$, and, for every $p \in P$, ports $p!$ and $p?$ synchronize (i.e., $p! \in N$ if and only if $p? \in N$ for every transition $(q, N, g, q') \in \rightarrow$).

[5]To simplify notation, we deviate from [DJAB15] by excluding internal ports.

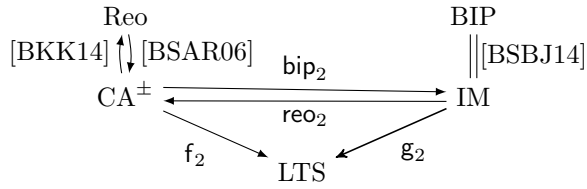


Figure 2.9: Translations and interpretations in the data-sensitive domain.

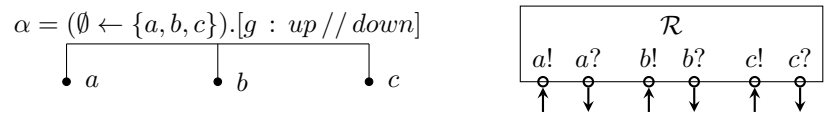


Figure 2.10: Simulating bidirectional ports in BIP with unidirectional ports in Reo.

As in Section 2.2, we interpret all connectors as labelled transition systems. Then, we define translations between Reo connectors ($\mathrm{CA}^{\pm}$) and BIP connectors (IM), and show that they preserve properties.

2.3.1 Interpretation of BIP and Reo

Consider the diagram in Figure 2.9. Classes $\mathrm{CA}^{\pm}$ and IM consist of constraint automata with polarity and BIP interaction models. Morphisms bip_2 and reo_2 are translations of those classes and f_2 and g_2 are interpretations in a common LTS semantics. We do not intend to redefine the semantics of constraint automata with polarity and of BIP interaction models in this section. Hence, we interpret them using their definitions from [BSAR06, BSBJ14].

The class LTS in Figure 2.9 is the class of all labelled transition systems over an alphabet $(D+1)^{2P}$, where D is a set of data items; $1 = \{0\}$, where 0 represents the absence of data (similar to *void* or *null*); and $2P = \{p!, p? \mid p \in P\}$ is the *duplicated (unidirectional) port set* of a set of (bidirectional) ports P. If the environment writes a datum d to bidirectional port p of a connector, then we represent this by an assignment of d to the unidirectional port $p!$. If the environment reads a datum d from a bidirectional port p of a connector, then we represent this by an assignment of d to the unidirectional port $p?$.

Example 2.3.1. Figure 2.10 shows an example of this port duplication. First, the upward data transfer expression in α takes data from the bottom ports a, b and c. In the Reo connector $\mathcal{R}$, this corresponds to taking data from ports $a!$, $b!$ and $c!$. Finally, the downward data transfer expression in the BIP interaction expression α offers data to the bottom ports, which corresponds in Reo connector $\mathcal{R}$ to offering data to ports $a?$, $b?$ and $c?$. ◊

Interpretation of IM We first define the interpretation $\mathsf{g}_2(\Gamma) \in \mathrm{LTS}$ of a BIP interaction model Γ. We define the interface of $\mathsf{g}_2(\Gamma)$ to be $2P = \{p!, p? \mid p \in P\}$, where P is the interface of Γ. We define the data domain of $\mathsf{g}_2(\Gamma)$ to be $\mathcal{D} = \bigcup_{p \in P} \mathsf{D}_p$, where D_p is the data type of port p (cf. Section 2.1.1). We associate to

every interaction expression $\alpha \in \Gamma$ a set $\Delta(\alpha) \subseteq (\mathcal{D}+1)^{2P}$ of data assignments $\delta : 2P \to \mathcal{D}+1$, and we add, for every $\alpha \in \Gamma$ and $\delta \in \Delta(\alpha)$, a transition (q, δ, q) to the stateless labelled transition system $\mathsf{g}_2(\Gamma)$.

We introduce some notation to define the set of data assignments $\Delta(\alpha)$. For every BIP interaction expression α, we write P_α for its bottom ports, g_α for its guard, up^α_w and up^α_L for the restriction of the up function to its top port and its local variables, respectively, and dn^α_{bot} for the restriction of the down function to its bottom ports. For every data assignment $\delta : 2P \to \mathcal{D}+1$, we define $\delta_{up}(p) = \delta(p!)$ and $\delta_{dn}(p) = \delta(p?)$, for all $p \in P_\alpha$.

In this notation, we define

$$\mathsf{g}_2(\Gamma) = (\{q\}, (\mathcal{D}+1)^{2P}, \{(q, \delta, q) \mid \alpha \in \Gamma, \delta \in \Delta(\alpha)\}), \tag{2.6}$$

where $\delta \in \Delta(\alpha)$ iff $\delta(2P \setminus 2P_\alpha) = \{0\}$, $\delta_{dn} = dn^\alpha_{bot}(up^\alpha_w(\delta_{up}), up^\alpha_L(\delta_{up}))$, and $g_\alpha(\delta_{up}) = \mathtt{tt}$. Note that we use the value of $up^\alpha_w(\delta_{up})$ as a local variable, since we consider only non-hierarchical BIP interaction models.

In [BSBJ14], Bliudze et al. encode BIP interaction models in *Top/Bottom (T/B) components*, i.e., an automaton over interaction expressions together with local variables. Furthermore, they define a semantics for T/B components, which indirectly defines an interpretation of interaction models. Equation (2.6) imitates this interpretation without using T/B components explicitly.

Interpretation of $\mathrm{CA}^\pm$ We now define the interpretation of a stateless constraint automaton with polarity $\mathcal{A} = (\{q\}, \mathcal{N}_{in}, \mathcal{N}_{out}, \to, q) \in \mathrm{CA}^\pm$ over a data domain $\mathcal{D}$. By definition, we find a set of unidirectional ports P, such that $\mathcal{N}_{in} = \{p! \mid p \in P\}$, $\mathcal{N}_{out} = \{p? \mid p \in P\}$, and, for every $p \in P$, ports $p!$ and $p?$ synchronize. We use $2P$ as the port names of $\mathsf{f}_2(\mathcal{A})$. We obtain the transitions of $\mathsf{f}_2(\mathcal{A})$ by replacing every transition labelled with N, g in $\mathcal{A}$ with a set of transitions labelled with $\delta \in \Delta(N, g) = \{\delta : 2P \to \mathcal{D}+1 \mid \delta(2P \setminus N) = \{0\}, \delta \models g\}$, where $\Delta(N, g)$ contains all data assignments $\delta : 2P \to \mathcal{D}+1$ that satisfy the synchronization constraint N and data constraint g. Now, define

$$\mathsf{f}_2(\mathcal{A}) = (\{q\}, (\mathcal{D}+1)^{2P}, \{(q, \delta, q) \mid q \xrightarrow{N,g} q, \delta \in \Delta(N, g)\}). \tag{2.7}$$

2.3.2 Reo to BIP

Since BIP interaction models are stateless, we cannot translate an arbitrary constraint automaton (i.e., Reo connector) into BIP. Interaction models in BIP preclude keeping track of the state of a Reo connector. Hence, the translation of the interaction model of a BIP architecture into a port automaton in Section 2.2.2 inspires us for our translation bip_2.

First, we describe intuitively how we translate a stateless constraint automaton $\mathcal{A}$ over a data domain $\mathcal{D}$ to a BIP interaction model. We transform every transition in $\mathcal{A}$ with label N, g into a simple BIP connector with N as its bottom ports, together with a guard, an up and a down function that mimic the data constraint g. We define the corresponding set $\mathsf{bip}_2(\mathcal{A})$ of BIP interaction expressions by the set of all transformed transitions from $\mathcal{A}$.

We now construct an interaction expression for any transition labelled N, g in automaton $\mathcal{A}$ as follows:

$$\alpha(N,g) = (\{w_{N,g}\} \leftarrow P_N).[g_{in}(X_{P_N}) \,:\, Y_{P_N} := \mathrm{solve}(g, X_{P_N}) \,//\, X_{P_N} := Y_{P_N}],$$

where P_N satisfies $2P_N = \{p!, p? \mid p \in P_N\} = N$; the variables $X_{P_N} = \{x_p \mid p \in P_N\}$ model the values assigned to bottom ports; the variables $Y_{P_N} = \{y_p \mid p \in P_N\}$ model some fresh local variables; the guard g_{in} is any quantifier free formula equivalent to $\exists O_N : g(I_N, O_N)$, with input variables $I_N = \{d_{p!} \mid p! \in N\}$ and output variables $O_N = \{d_{p?} \mid p? \in N\}$; and function $\mathrm{solve}(g, X_{P_N})$ returns any vector Y_{P_N} satisfying $g(X_{P_N}, Y_{P_N})$. All variables have data type $\mathcal{D}$ (the data domain of $\mathcal{A}$), i.e., $x_p{:}\mathcal{D}$ for all $p \in \mathcal{N}$.

Let P be the interface of $\mathcal{A}$. Define bip_2 as follows:

$$\mathsf{bip}_2(\mathcal{A}) = (P, \{\alpha(N,g) \mid (q, N, g, q) \in \rightarrow\}). \tag{2.8}$$

Intuitively, the solve function in $\alpha(N,g)$ computes a solution of the guard g, given all input values $d_{p!}$, with $p! \in N$. Note that the solve function in $\alpha(N,g)$ is not deterministic. However, comparing the solve function to the random function in Figure 4 in [BSBJ14], we see that this generality is justified.

Example 2.3.2. Consider a Sync channel from port a to b. To model this channel as a constraint automaton $\mathcal{A} \in \mathrm{CA}^{\pm}$, we duplicate the ports and obtain the interface $P = \{a!, a?, b!, b?\}$. In view of Figure 2.2, we model a Sync channel as $\mathcal{A} = (\{q\}, P, \{(q, P, g, q)\}, q)$, with $g \equiv d_{a!} = d_{b?}$. The translation of $\mathcal{A}$ to a BIP interaction model consist of a single BIP interaction expression

$$\alpha(P,g) = (\{w\} \leftarrow \{a,b\}).[\mathtt{tt} \,:\, (y_a, y_b) := (x_a, x_b) \,//\, (x_a, x_b) := (y_a, y_b)],$$

because $\mathtt{tt} \equiv \exists d_{a?} \exists d_{b?} (d_{a!} = d_{b?})$, for any given $d_{a!}, d_{b!} \in \mathcal{D}$, and the solve function $\mathrm{solve}(g, x_a, x_b) = (x_a, x_b)$ acts as the identity. ◊

2.3.3 BIP to Reo

The correspondence between BIP interaction expressions and automata transitions from Section 2.3.2, provides the main idea for the translation of interaction models into stateless constraint automata. If Γ is a set of simple BIP connectors, we assign to every $\alpha \in \Gamma$ a transition τ_α labelled with $N(\alpha), g(\alpha)$, and subsequently construct the stateless constraint automaton consisting of all such τ_α transitions.

Let α be a simple BIP interaction expression. Define $N(\alpha) = 2P_\alpha = \{p?, p! \mid p \in P_\alpha\}$. Furthermore, let $D? = (d_{p?})_{p \in P}$, $D! = (d_{p!})_{p \in P}$, and define

$$g(\alpha) \;=\; \textstyle\bigwedge_{p \in P} d_{p!}, d_{p?} \in \mathsf{D}_p \;\wedge\; g_\alpha(D!) \;\wedge\; D? = dn^\alpha_{bot}(up^\alpha_w(D!), up^\alpha_L(D!)),$$

where we use our relaxation on the data constraint language from Section 2.1.2 and our notation regarding a BIP interaction expression α from Section 2.3.1. Note that $g(\alpha)$ is independent of the top port w, because we consider only non-hierarchical connectors.

Let Γ be a set of simple BIP connectors with interface P. Recall that $\mathcal{D} = \bigcup_{p \in P} \mathsf{D}_p$. Define the constraint automaton $\mathsf{reo}_2(\Gamma)$ over $\mathcal{D}$ by

$$\mathsf{reo}_2(\Gamma) = (\{q\}, P! \cup P?, \{(q, N(\alpha), g(\alpha), q) \mid \alpha \in \Gamma\}, q). \tag{2.9}$$

Example 2.3.3. Consider the interaction expression $\alpha_{\max}$ from Example 2.1.2, with data domain restricted to $\mathcal{D} = \{0, \ldots, 2^{32} - 1\}$. We translate the interaction model $\Gamma = \{\alpha_{\max}\}$ using Equation (2.9), i.e., we compute $\mathcal{A} = \mathsf{reo}_2(\Gamma)$. Trivially, $\mathcal{A}$ is stateless. Its set of input ports equals $P! = \{a!, b!\}$, and its set of output ports equals $P? = \{a?, b?\}$. $\mathcal{A}$ has a single transition (q, N, g, q), with guard $g \equiv \bigvee_{x,y,z \in \mathcal{D} \,:\, z = \max(x,y)} (d_{a!} = x \wedge d_{b!} = y \wedge d_{a?} = z \wedge d_{b?} = z)$ and synchronization constraint $N = \{a!, b!, a?, b?\}$. ◊

2.3.4 Preservation of properties

To show the faithfulness of translations bip_2 and reo_2, we show that interpretations f_2 and g_2 commute with translations bip_2 and reo_2 in Figure 2.9.

Theorem 2.3.1. *For all* $\mathcal{A} \in \mathrm{CA}^{\pm}$ *and all* $\Gamma \in \mathrm{IM}$ *we have* $\mathsf{g}_2(\mathsf{bip}_2(\mathcal{A})) = \mathsf{f}_2(\mathcal{A})$ *and* $\mathsf{f}_2(\mathsf{reo}_2(\Gamma)) = \mathsf{g}_2(\Gamma)$.

Proof. (Sketch) Let $\mathcal{A} \in \mathrm{CA}^{\pm}$ be a constraint automaton with polarity with interface P, let (q, N, g, q) be a transition in $\mathcal{A}$, and let $\delta : 2P \to \mathcal{D} + 1$ be a data assignment. By definition, we have $\delta \in \Delta(\alpha(N, g))$ if and only if $\delta(2P \setminus 2P_\alpha) = \{0\}$, $\delta_{dn} = dn^{\alpha}_{bot}(up^{\alpha}_{w}(\delta_{up}), up^{\alpha}_{L}(\delta_{up}))$, and $g_\alpha(\delta_{up}) = \mathtt{tt}$, where $\alpha = \alpha(N, g)$. Using the definition of $\alpha(N, g)$, it follows that $\delta \in \Delta(\alpha(N, g))$ if and only if $\delta(2P \setminus N) = \{0\}$ and δ satisfies g. Thus, $\delta \in \Delta(\alpha(N, g))$ if and only if $\delta \in \Delta(N, g)$. Using the definitions of f_2 and g_2, we find that $\mathsf{g}_2(\mathsf{bip}_2(\mathcal{A}))) = \mathsf{f}_2(\mathcal{A})$.

Let $\Gamma \in \mathrm{IM}$ be a BIP interaction model with interface P, let $\alpha \in \Gamma$ be a BIP interaction expression, and let $\delta : 2P \to \mathcal{D}+1$ be a data assignment. By definition, we have $\delta \in \Delta(N(\alpha), g(\alpha))$ if and only if $\delta(2P \setminus N(\alpha)) = \{0\}$ and δ satisfies $g(\alpha)$. Using the definition of $N(\alpha) = 2P_\alpha$ and $g(\alpha)$, it follows $\delta \in \Delta(N(\alpha), g(\alpha))$ if and only if $\delta(2P \setminus 2P_\alpha) = \{0\}$ and $\delta_{dn} = dn^{\alpha}_{bot}(up^{\alpha}_{w}(\delta_{up}), up^{\alpha}_{L}(\delta_{up}))$, and $g_\alpha(\delta_{up}) = \mathtt{tt}$. Thus, $\delta \in \Delta(N(\alpha), g(\alpha))$ if and only if $\Delta(\alpha)$. Using the definitions of f_2 and g_2, we find that $\mathsf{f}_2(\mathsf{reo}_2(\Gamma)) = \mathsf{g}_2(\Gamma)$. □

Corollary. *The translations* bip_2 *and* reo_2 *preserve all properties expressible in* LTS*, i.e.,* $\mathsf{f}_2(\mathcal{A}) \in P \Leftrightarrow \mathsf{g}_2(\mathsf{bip}_2(\mathcal{A})) \in P$ *and* $\mathsf{g}_2(\Gamma) \in P \Leftrightarrow \mathsf{f}_2(\mathsf{reo}_2(\Gamma)) \in P$ *for all* $P \subseteq$ LTS*,* $\mathcal{A} \in \mathrm{CA}^{\pm}$ *and* $\Gamma \in \mathrm{IM}$*.*

Example 2.3.4. Consider the following safety property φ for the interaction expression $\alpha_{\max}$ from Example 2.1.2: "the value retrieved from port a equals zero". Clearly, this safety property does not hold, whenever a or b offers a non-zero integer. Note that φ depends solely on the interpretation of the interaction model $\Gamma = \{\alpha_{\max}\}$ in LTS, and hence φ is expressible in LTS. Using Section 2.3.4 we conclude that φ is false also for $\mathcal{A}_{\max} = \mathsf{reo}_2(\{\alpha_{\max}\})$. Thus, we know any executable code generated from the constraint automaton $\mathcal{A}_{\max}$ does not satisfy φ. More generally, Section 2.3.4 allows us to use the Reo compiler to generate correct code for a BIP interaction model. ◊

2.4 Data-sensitive BIP architectures

Due to the absence[6] of a data-sensitive equivalent of a BIP architecture, our data-sensitive translation presented in Section 2.2 appears restricted in comparison with our data-agnostic translation in Section 2.3. It seems straightforward to extend BIP architectures to the data-sensitive domain by adding coordinating components and replacing the interaction model with a data-sensitive interaction model. However, this extension requires also a composition operator for interaction models, which is not present in the current literature [BSBJ14]. In this section, we propose a data-sensitive extension to BIP architectures and their composition, and we show how this extension relates to Reo connectors.

2.4.1 Composition of BIP interaction expressions

BIP architecture composition in Definition 2.1.4 consists of two parts: it merges the coordinating components into a single set of coordinators, and it composes the BIP interaction models by gluing interactions together. This gluing has not yet been defined for data-sensitive BIP interaction expressions [BSBJ14]. We now propose a possible definition for this gluing of data-sensitive BIP interactions.

Let α_1 and α_2 be two BIP interaction expressions. Intuitively, their composition $\alpha_1 * \alpha_2$ synchronizes α_1 and α_2. That is, both interactions fire in a single atomic step. This means that the composition should evaluate both guards and synchronously execute the upward and downward dataflow of both interaction expressions whenever both guards are satisfied.

Suppose α_1 and α_2 do not share local variables. In that case, we can simulate synchronous execution of the upward data transfer expressions of α_1 and α_2 by sequentially executing both expressions. However, since α_1 and α_2 may share bottom ports, the downward data transfer expressions may write different values to the shared bottom ports. Hence, we cannot simply execute both downward data transfer expressions sequentially.

Generally, the downward data transfer expression of a BIP interaction expression α may depend on the top ports of α. When this is the case, the value produced by the downward data expression becomes known only after hierarchical composition. Thus, at design time we can neither check nor avoid that the downward data transfer expressions of α_1 and α_2 disagree on their shared bottom ports.

Example 2.4.1. Consider the BIP interaction expression

$$\alpha'_{\max} = (\{w\} \leftarrow \{a, b\}).[\mathtt{tt} : \ x_w := \max(x_a, x_b) \,//\, x_a, x_b := x_w],$$

where each port in $\mathcal{P} = \{a, b, w, l\}$ is of type integer, i.e., $x_p : \mathsf{D}_p = \mathbb{Z}$, for all $p \in \mathcal{P}$, and $\mathtt{tt}$ is true. The value of the downward data transfer expression in $\alpha'_{\max}$ depends on the value x_w of its top port w. ♢

When two BIP interaction expressions α_1 and α_2 do not depend on their top ports, we can determine whether α_1 and α_2 agree on shared bottom ports. Indeed, we know the relationship between the values presented to the upward data transfer expression and the values computed by the downward data transfer expression.

[6] This text was written before [BHM19]

This allows us to force agreement already in the guard of the composed BIP interaction expression $\alpha_1 * \alpha_2$. In this way, we can safely execute both downward data transfer expressions sequentially.

Definition 2.4.1 (Composition of interaction expressions)**.** Let α_1 and α_2 be two interaction expressions without shared local variables and for which the downward data transfer expression does not depend on top ports. We define the composition $\alpha_1 * \alpha_2$ of α_1 and α_2 as follows: $\text{top}(\alpha_1 * \alpha_2) = \emptyset$, $\text{bot}(\alpha_1 * \alpha_2) = \text{bot}(\alpha_1) \cup \text{bot}(\alpha_2)$, $up_{\alpha_1 * \alpha_2} = (up_{\alpha_1}, up_{\alpha_2})$, $dn_{\alpha_1 * \alpha_2} = (dn_{\alpha_1}, dn_{\alpha_2})$,

$$g_{\alpha_1 * \alpha_2} = g_{\alpha_1} \wedge g_{\alpha_2} \wedge \Big[dn_{\alpha_1}|_S(up_{\alpha_1}(X^1_Q, X^1_L)) = dn_{\alpha_2}|_S(up_{\alpha_2}(X^2_Q, X^2_L)) \Big],$$

where $dn_{\alpha_i}|_S$ is the restriction of dn_{α_i} to the shared variables X_S over $S = \text{bot}(\alpha_1) \cap \text{bot}(\alpha_2)$, X^i_Q are the variables over $\text{bot}(\alpha_i)$, and X^i_L are the local variables of α_i. The local variables of $\alpha_1 * \alpha_2$ are $X^1_L \cup X^2_L$.

Example 2.4.2. Consider the following BIP interaction expressions $\alpha_1 = (\emptyset \leftarrow \{a, b\}).[\texttt{tt} : x_k := x_a \,//\, x_b := x_k]$, and $\alpha_2 = (\emptyset \leftarrow \{b, c\}).[\texttt{tt} : x_l := x_b \,//\, x_c := x_l]$, which simulate two Sync channels over a, b and b, c respectively (See Figure 2.2). Then, their composition $\alpha_1 * \alpha_2$ is given by $(\emptyset \leftarrow \{a, b, c\}).[\texttt{tt} : x_k := x_a; x_l := x_b \,//\, x_b := x_k; x_c := x_l)$.

This composition merely synchronizes ports a and c, while there is no data exchange between them. On the other hand, the composition of the two Sync channels does transfer data from source a to sink c. Hence, composition of interaction expressions does not correspond directly to composition of Reo channels. ◊

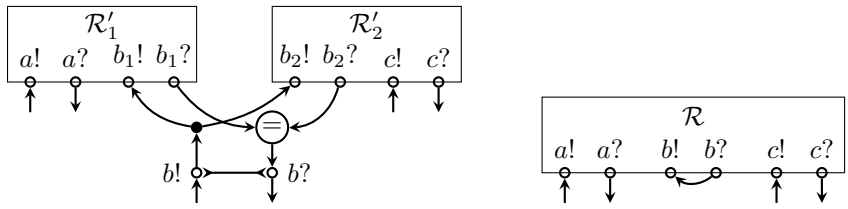
Example 2.4.3. Consider the following BIP interaction expressions $\alpha_1 = (\emptyset \leftarrow \{a, b\}).[\texttt{tt} : x_k := \max(x_a, x_b) \,//\, x_a, x_b := x_k]$, and $\alpha_2 = (\emptyset \leftarrow \{b, c\}).[\texttt{tt} : x_l := \max(x_b, x_c) \,//\, x_b, x_c := x_l]$, which are similar to the BIP interaction expression $\alpha_{\max}$ from Example 2.1.2 (except that we omitted the top port). Intuitively, perhaps, combining $\max(x_a, x_b)$ and $\max(x_b, x_c)$ yields $\max(x_a, x_b, x_c)$. However, the restriction that downward data transfer expressions of α_1 and α_2 must agree on their shared bottom port b, implies that the composition $\alpha_1 * \alpha_2$ takes the following form:

$$\begin{aligned} \alpha_1 * \alpha_2 \quad = \quad & (\emptyset \leftarrow \{a, b, c\}).[\max(x_a, x_b) = \max(x_b, x_c) : \\ & x_k := \max(x_a, x_b); x_l := \max(x_b, x_c) \,//\, x_a, x_b := x_k; x_c := x_l]. \end{aligned}$$

The upward and downward data transfer expressions are composed sequentially. Note that since the downward data transfer does not depend on top ports, the sequential order in this composition is irrelevant. The guard consists of the conjunction of the guards of α_1 and α_2, together with the statement that the downward data transfer expressions agree on the value of x_b. ◊

2.4.2 Abstraction on BIP interaction expressions

Example 2.4.2 shows that the composition of interaction expressions does not correspond directly to composition of Reo connectors. We now investigate the reason for this incompatibility and show that it is possible to simulate composition of Reo connectors by means of an abstraction operator on BIP interaction expressions.



(a) Composition (b) Abstraction

Figure 2.11: Composition (a) and abstraction (b) for interaction expressions.

Consider a Sync channel $\mathcal{R}_1$ over a and b and a Sync channel $\mathcal{R}_2$ over b and c (cf. Figure 2.2). In order to comply with the notation from Section 2.3, we rename every channel end p to $p!$, if it is a source end, or $p?$, if it is a sink end. In this way, we obtain two Reo connectors $\mathcal{R}'_1$ and $\mathcal{R}'_2$ that are Sync channels over $a!$, $b?$ and $b!$, $c?$ respectively.

This renaming splits node b into an output port $b?$ and an input port $b!$. To preserve the intention of composition in Reo, we need to add a Sync channel from $p?$ to $p!$, for every internal port p of the connector. For boundary nodes, there is no need to add a Sync channel.

Using the translation discussed in Section 2.3.2, we obtain from $\mathcal{R}'_1$ a BIP interaction expression α_1 over a and b. Similarly, we find from $\mathcal{R}'_2$ a BIP interaction expression α_2 over b and c. The composition $\alpha_1 * \alpha_2$ of α_1 and α_2 yields a BIP interaction expression over a, b and c.

The composition of BIP interaction expressions may also be described in terms of the Reo connectors $\mathcal{R}'_1$ and $\mathcal{R}'_2$. Figure 2.11(a) shows the construction that simulates this composition. First, we split $\mathcal{R}_1$ and $\mathcal{R}_2$ by renaming their shared ports $b!$ and $b?$ to $b_1!$, $b_2!$ and $b_1?$, $b_2?$ respectively, and we add two fresh ports $b!$ and $b?$. We replicate the data that we observe at $b!$ to both $b_1!$ and $b_2!$. We check the data retrieved from $b_1?$ and $b_2?$ for equality and pass it to $b?$. The node with the equality sign is responsible for this equality check. This node is a Reo component that takes two identical data items from its input and synchronously transfers one of these items to its output. Finally, we synchronize $\mathcal{R}_1$ and $\mathcal{R}_2$ by adding a SyncDrain between $b!$ and $b?$ (cf. Figure 2.2).

As in Example 2.4.2, we see that the BIP interaction expression composition $\mathcal{R}$ of $\mathcal{R}'_1$ and $\mathcal{R}'_2$ yields no dataflow from a to c. Indeed, the depicted composition merely synchronizes $b?$ and $b!$ using a SyncDrain channel. However, the renaming of $\mathcal{R}_1$ and $\mathcal{R}_2$ to $\mathcal{R}'_1$ and $\mathcal{R}'_2$ required an additional Sync channel from $b?$ to $b!$. Hence, in order to simulate composition of Reo connectors, we need to add this Sync channel. We model this addition of the Sync channel by an operation called *abstraction*. Figure 2.11(b) shows the effect of abstraction on the composed Reo connector $\mathcal{R}$.

In terms of Reo connectors, the effect of abstraction is clear. Now, we formulate this abstraction operator in terms of interaction expressions. Consider the interaction expression in Figure 2.11(b). The addition of the Sync channel imposes a restriction on the observed dataflow at b: the data presented as input for the upward data transfer equals the output retrieved from the downward data transfer

expression. This means that the abstraction of b requires us to find a fixed point of the composition of the upward and downward data transfer expressions. Moreover, this fixed point needs to satisfy the guard of the interaction expression. Once we have computed this fixed point, we just use it as input to the interaction.

Since we use our own input at b instead of input obtained from a BIP component, we must hide b from the interface of the interaction. This explains why we call this operation abstraction.

Definition 2.4.2 (Abstraction on interaction models)**.** Let α be the BIP interaction expression $(\emptyset \leftarrow Q).[g : X_L := up(X_Q) \,//\, X_Q := dn(X_L)]$, and let $p \in Q$ be a bottom port of α. Let $ud_p(X_Q) = dn(up(X_Q))|_{x_p}$ be the restriction to x_p of the composition of up and dn. Denote the set of fixed points of the function $x_p \mapsto ud_p(x_p, X_{Q\setminus\{p\}})$ by F. Let $fp(X_{Q\setminus\{p\}}) \in F$ be any partial function that returns, when possible, any fixed point from F such that $g(x_p, X_{Q\setminus\{p\}})$. We call fp a *fixed point function* of α with respect to p. Then, we define the *abstraction* $\alpha \setminus p$ of α with respect to p as

$$(\emptyset \leftarrow Q \setminus \{p\}).[\exists x_p \in F.\, g : X_L := up(X_{Q\setminus\{p\}}, fp(X_{Q\setminus\{p\}})) \,//\, X_{Q\setminus\{p\}} := dn(X_L)]\,.$$

For convenience, we assume that a fixed point function is a random function. However, in practice we care only about the fact that this function returns a fixed point from F that satisfies the guard.

Example 2.4.4. Consider the BIP interaction expressions α_1 and α_2 from Example 2.4.2, and their shared bottom port b. We compute the abstraction $\alpha = (\alpha_1 * \alpha_2) \setminus b$. The mapping $ud_b : x_b \mapsto x_a$ gives the restriction to x_b of the composition of the upward and downward data transfer expressions. The set of fixed points of ud_b consists of $F = \{x_a\}$. Trivially, the guard of α equals $g_\alpha = \texttt{tt}$. Hence, the fixed point function of α is given by $fp(x_a, x_c) = x_a$. Therefore, we find that $\alpha = (\emptyset \leftarrow \{a, c\}).[\texttt{tt} : x_k := x_a; x_l := x_a \,//\, x_c := x_l]$.

We see that the value of x_a flows via x_b to x_c, which simulates the dataflow in the composition of the two Sync channels in Example 2.4.2. ◊

Example 2.4.5. Consider the composed BIP interaction expression $\alpha_1 * \alpha_2$ from Example 2.4.3 and its bottom port b. We compute the abstraction $\alpha = (\alpha_1 * \alpha_2) \setminus b$. The restriction to x_b of the composition of the upward and downward data transfer expressions is given by the mapping $ud_b : x_b \mapsto \max(x_a, x_b)$. The set of fixed points of ud_b is given by $F = \{v \mid v \geq x_a\}$. Since any $x_b \geq x_a, x_c$ can serve as a witness, the guard of α simplifies to $g_\alpha \equiv \exists x_b \geq x_a.(x_b \geq x_c) \vee (x_c \geq x_b \wedge x_b = x_c) \equiv \texttt{tt}$. Thus, the fixed point function $fp(x_a, x_c) = \text{rnd}(\{y \mid y \geq x_a, x_c\})$ may return any value greater than or equal to both x_a and x_c. Finally, we get that $(\alpha_1 * \alpha_2) \setminus b$ is given by

$$(\emptyset \leftarrow \{a, c\}).[\texttt{tt} : x_k := \max(x_a, r);\, x_l := \max(r, x_c) \,//\, x_a := x_k; x_c := x_l],$$

where $r = \text{rnd}(\{v \mid v \geq x_a, x_c\})$. Hence, since r is random, $(\alpha_1 * \alpha_2) \setminus b$ returns the value $\max(x_a, x_c) + C$, where $C \geq 0$ is an arbitrary positive number. ◊

2.4.3 Data-sensitive BIP architectures

The extension of BIP architectures to the data-sensitive domain requires us to combine data-agnostic BIP architectures with interaction expressions that are data-sensitive [ABB+14, BSBJ14].

First, we need to generalize the coordinating components in a BIP architecture. For this, we use a restricted type of constraint automata with polarity.

Definition 2.4.3 (Atomic BIP components)**.** An atomic BIP component is a constraint automaton $\mathcal{A}$ such that every transition $(q, N, g, q') \in \rightarrow$ synchronizes at most one bidirectional port, i.e., $N \in \{\emptyset, \{p!, p?\}\}$, for some bidirectional port p.

Coordinating components in data-agnostic BIP architectures are disconnected (cf. Definition 2.1.1). This notion lifts trivially to sets of atomic BIP components.

Next, we generalize the data-agnostic interaction model γ to a data-sensitive interaction model Γ. Every data-sensitive BIP interaction expression $\alpha \in \Gamma$ reduces to a data-agnostic interaction $N = \mathrm{bot}(\alpha) \in \gamma$.

Definition 2.4.4. A *data-sensitive BIP architecture* is a triple $A = (\mathcal{C}, P, \Gamma)$ consisting of a finite disconnected set $\mathcal{C}$ of atomic BIP components, a finite set P of ports, and an interaction model Γ over P (cf. Definition 2.1.1 and 2.1.6).

Using the operational semantics of atomic components, provided in [BSBJ14, Definition 3.2], and the interpretation $\mathtt{g}_2$ of a data-sensitive interaction model, defined in Section 2.3.1, we define the following semantics for data-sensitive BIP architectures:

Definition 2.4.5 (Semantics of data-sensitive BIP architecture)**.** Consider a data-sensitive BIP architecture $A = (\{C_1, \ldots, C_n\}, P, \Gamma)$. The semantics $\mathtt{g}_3(A)$ of A is given by the labelled transition system $(\prod_{i=1}^{n} Q_i, (\mathcal{D}+1)^{2P}, \rightarrow)$, where Q_i is the state space of atomic component C_i, and $\rightarrow$ is the smallest relation that satisfies the following rule: if $\delta : 2P \rightarrow \mathcal{D}+1$ is a data assignment such that (q, δ, q) is a transition in $\mathtt{g}_2(\Gamma)$, and for all components C_i we have either

1. $q_i' = q_i$ and $\mathrm{dom}(\delta) \cap P_i = \emptyset$; or
2. (q_i, N, g, q_i') is a transition in C_i, $\mathrm{dom}(\delta) \cap P_i = N$, and $\delta \models g$,

then $(q_i)_{i=1}^n \xrightarrow{\delta} (q_i')_{i=1}^n$.

2.4.4 Composition of data-sensitive BIP architectures

Using the concepts introduced in Sections 2.4.1 and 2.4.2, we lift the composition operator of data-agnostic BIP architectures to data-sensitive BIP architectures.

Because the composition of coordinating components consists of set-union, its extension to data-sensitive BIP architectures is trivial. The composition of data-sensitive interaction models is less straightforward. Given two data-sensitive BIP interaction models Γ_1 and Γ_2, the composed data-sensitive interaction model Γ should intuitively consists of composed BIP interaction expressions $\alpha_1 * \alpha_2$, with $\alpha_i \in \Gamma_i$ for both i. However, we cannot allow every combination of α_1 and α_2, because they may synchronize on different shared ports.

Every BIP interaction expression α in the data-sensitive domain, reduces to a BIP interaction $\mathrm{bot}(\alpha)$ in the data-agnostic domain, where $\mathrm{bot}(\alpha)$ are the bottom ports of α. In this way, a BIP interaction model Γ reduces to a data-agnostic interaction model $\gamma = \{\mathrm{bot}(\alpha) \mid \alpha \in \Gamma\}$.

Let γ_1 and γ_2 be the reduced BIP interaction models derived from Γ_1 and Γ_2, and consider the BIP interactions $\mathrm{bot}(\alpha_1)$ and $\mathrm{bot}(\alpha_2)$ in γ_1 and γ_2. Let γ be the composition of γ_1 and γ_2. According to Definition 2.1.4, we have that $N = \mathrm{bot}(\alpha_1 * \alpha_2) \in \gamma$ if and only if $N \cap P_1 \in \gamma_1$ and $N \cap P_2 \in \gamma_2$. It is not hard to see that, in order to ensure that $\mathrm{bot}(\alpha_1 * \alpha_2) \in \gamma$, it suffices to assume that $\mathrm{bot}(\alpha_1) \cap P_2 = \mathrm{bot}(\alpha_2) \cap P_1$.

Definition 2.4.6 (Composition of data-sensitive BIP interaction models)**.** Let Γ_1 and Γ_2 be two interaction models with interfaces P_1 and P_2, respectively, such that no BIP interaction expression has top ports and no local variable is shared. We define the composition of Γ_1 and Γ_2 as $\Gamma_1 * \Gamma_2 = \{\alpha_1 * \alpha_2 \mid \alpha_i \in \Gamma_i, \mathrm{bot}(\alpha_1) \cap P_2 = \mathrm{bot}(\alpha_2) \cap P_1\}$.

Notice that the restriction to interaction expressions that do not have top ports implies that the condition in Definition 2.4.1, which requires that the downward data transfer do not depend on top ports, is trivially satisfied. Hence, the composition operator on data-sensitive BIP interaction models is well-defined.

Moreover, notice that it does not make sense to weaken the condition $\mathrm{bot}(\alpha_1) \cap P_2 = \mathrm{bot}(\alpha_2) \cap P_1$ any further. Suppose that α_1 and α_2 satisfy only $\mathrm{bot}(\alpha_1 * \alpha_2) \cap P_i \in \gamma_i$, for $i = 1, 2$. Then we find $\alpha_1' \in \Gamma_1$ and $\alpha_2' \in \Gamma_2$ such that $\mathrm{bot}(\alpha_1' * \alpha_2') = \mathrm{bot}(\alpha_1 * \alpha_2)$. Although, $\alpha_1' * \alpha_2'$ and $\alpha_1 * \alpha_2$ extend the same data-agnostic interaction, they may behave very differently with respect to data.

Now, Definition 2.4.6 allows us to define our desired composition operator for data-sensitive BIP architectures.

Definition 2.4.7 (Composition of data-sensitive BIP architectures)**.** Let $A_1 = (\mathcal{C}_1, P_1, \Gamma_1)$ and $A_2 = (\mathcal{C}_2, P_2, \Gamma_2)$ be two data sensitive BIP architectures such that $\mathcal{C}_1 \cup \mathcal{C}_2$ is disconnected and no BIP interaction expression has top ports and A_1 and A_2 share no local variables. Then, we define the composition $A_1 \oplus A_2$ as $(\mathcal{C}_1 \cup \mathcal{C}_2, P_1 \cup P_2, \Gamma_1 * \Gamma_2)$.

The composition of data-sensitive BIP interaction models in Definition 2.4.6 can cause an interaction-space explosion. Such an explosion can never occur using hierarchical composition only [BSBJ14]. This makes the data-sensitive BIP architecture composition more expressive than hierarchical composition.

Example 2.4.6. Consider a Reo connector that consist of N parallel Sync channels, i.e., we have a Sync channel $\mathcal{R}_{a_i,b_i}$ from a_i to b_i, for each $i \in \{1, \ldots, N\}$. Since any combination of Sync channels can fire, the associated constraint automaton exhibits 2^N transitions. The direct translation from Section 2.3 requires us to translate every transition into a corresponding BIP interaction expression.

Using BIP architecture composition from Definition 2.4.7, it suffices to translate each Sync channel $\mathcal{R}_{a_i,b_i}$ into a BIP architecture $A_{a_i,b_i} = (\emptyset, \{a_i, b_i\}, \{\alpha_{a_i \to b_i}, \alpha_\emptyset\})$, where $\alpha_{a_i \to b_i} = (\emptyset \leftarrow \{a_i, b_i\}).[\mathtt{tt} : x_l := x_{a_i} \,//\, x_{b_i} := x_l]$ models the Sync channel and $\alpha_\emptyset = (\emptyset \leftarrow \emptyset).[\mathtt{tt} : - \,//\, -]$ models the empty transition. This empty interaction allows the other BIP architectures to proceed independently of this

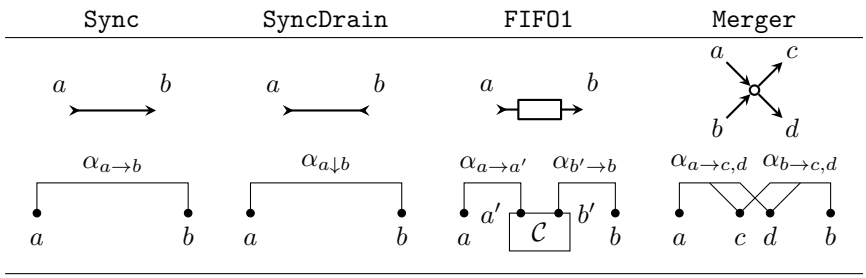


Figure 2.12: Translation of Reo channels and nodes to data-sensitive BIP architectures. The BIP interaction expressions are given by $\alpha_{a\to b,c} = (\emptyset \leftarrow \{a, b, c\}).[\texttt{tt} : x_l := x_a \,//\, x_b, x_c := x_l]$, $\alpha_{a\to b} = \alpha_{a\to b,b}$, and $\alpha_{a\downarrow b} = (\emptyset \leftarrow \{a, b\}).[\texttt{tt} : - \,//\, -]$. The atomic BIP component $\mathcal{C}$ models the behavior of the FIFO_1 channel.

BIP architecture. Hence, Definition 2.4.7 enables us to translate only N channels instead of 2^N transitions. ◊

Definition 2.4.8 (Abstraction of data-sensitive BIP architectures)**.** Let $A = (\mathcal{C}, P, \Gamma)$ be a data-sensitive BIP architecture, and $p \in P$ a dangling port (i.e., $p \notin P_C$, for all $C \in \mathcal{C}$). Then, we define the abstraction $A \setminus p$ as $(\mathcal{C}, P \setminus \{p\}, \{\alpha \setminus p \mid \alpha \in \Gamma\})$.

2.4.5 Incremental translation

The proposed composition operator from Definition 2.4.7 together with the abstraction operator from Definition 2.4.2 allow us to incrementally translate constraint automata to data-sensitive BIP architectures and vice versa. We formalize this by defining two translations, and show that they both preserve the semantics of translated entities.

Reo to BIP Consider a Reo circuit $\mathcal{R}$, and associate to each channel and node in $\mathcal{R}$ its constraint automaton (see Figure 2.2). Rename every input port p of any channel or node in $\mathcal{R}$ to $p!$, and every output port of any channel or node in $\mathcal{R}$ to $p?$. This procedure splits every shared port p into two ports $p!$ and $p?$, which essentially disconnects all channels and nodes. Write $X = \{\mathcal{A}_1, \ldots, \mathcal{A}_m\}$ for the obtained set of constraint automata with polarity. Our goal is to translate each $\mathcal{A}_i \in X$ individually to a data-sensitive BIP architecture, and then compose them using Definitions 2.4.2 and 2.4.7. To this end, we define the translation $\mathsf{bip}_3(\mathcal{A})$ of a BIP-friendly constraint automaton with polarity $\mathcal{A}$.

Let $\mathcal{A}$ be a constraint automaton with polarity over P, which means that $\mathcal{A}$ uses names from $2P = \{p!, p? \mid p \in P\}$. Since atomic components are not allowed to synchronize their ports and since interaction in BIP is stateless, we need to assume that $\mathcal{A}$ is *BIP-friendly*: $\mathcal{A}$ is either stateless (i.e., $Q_{\mathcal{A}} = \{q\}$) or does not synchronize any of its ports (i.e., for every transition (q, N, g, q') we have $N = \{p!, p?\}$ for some $p \in P$). Figure 2.2 shows some examples of BIP-friendly automata.

When $\mathcal{A}$ is stateless, we can translate $\mathcal{A}$ into an interaction model $\mathsf{bip}_2(\mathcal{A})$. We now simply define $\mathsf{bip}_3(\mathcal{A}) = (\emptyset, P, \mathsf{bip}_2(\mathcal{A}))$. See Figure 2.12 for an example. When

$\mathcal{A}$ does not synchronize any of its ports, we can interpret $\mathcal{A}$ as an atomic component $\mathcal{A}'$, where we rename every port $p \in P$ to a port $p' \in P'$. The prime is used only to construct a fresh port name. Now, we interpret every $p \in P$ as a dangling port of the translated data-sensitive BIP architecture and connect p with p' using the interaction $\alpha_{p,p'} = (\emptyset \leftarrow \{p, p'\}).[\mathtt{tt} : x_k := x_p; x_l := x_{p'} \,/\!/\, x_p := x_l; x_{p'} := x_k]$. Thus, we define

$$\mathsf{bip}_3(\mathcal{A}) = \begin{cases} (\emptyset, P, \mathsf{bip}_2(\mathcal{A})) & \text{if } \mathcal{A} \text{ is stateless} \\ (\{\mathcal{A}'\}, P \cup P', \{\alpha_{p,p'} \mid p \in P\}) & \text{if } \mathcal{A} \text{ is non-synchronizing} \end{cases} \tag{2.10}$$

The restriction that the automaton $\mathcal{A}$ should be either stateless or non-synchronizing is not problematic. Every synchronizing stateful automaton $\mathcal{A}$ can be decomposed into a set $\{\mathcal{A}_1, \ldots, \mathcal{A}_m\}$ of stateless and non-synchronizing automata [BKK14]. Indeed, each automaton in the decomposition is the CA representation of a stateless Reo channel or a FIFO_1 buffer.

Using the translation bip_3, we can now translate the Reo circuit $\mathcal{R}$ incrementally. Let $\{\mathcal{A}_1, \ldots, \mathcal{A}_m\}$ be a set of BIP-friendly constraint automata with polarity and $S = \{p \mid \{p!, p?\} \cap \mathcal{N}_{\mathcal{A}_i} \cap \mathcal{N}_{\mathcal{A}_j} \neq \emptyset \text{ for some distinct } i, j\}$ be the set of shared/internal ports of this system of automata. The following diagram illustrates the working of the incremental translation from Reo to BIP:

$$\begin{array}{ccccc} \{\mathcal{A}_1, \ldots, \mathcal{A}_m\} & & \xrightarrow{\quad\mathsf{bip}_3\quad} & & \{\mathsf{bip}_3(\mathcal{A}_1), \ldots, \mathsf{bip}_3(\mathcal{A}_m)\} \\ \downarrow & & & & \downarrow \\ \exists 2S(\mathcal{A}_1 \bowtie \cdots \mathcal{A}_m \bowtie \mathcal{G}) & \xrightarrow{\mathsf{f}_3} & L & \xleftarrow{\mathsf{g}_3} & (\mathsf{bip}_3(\mathcal{A}_1) \oplus \cdots \oplus \mathsf{bip}_3(\mathcal{A}_m)) \setminus S \end{array} \tag{2.11}$$

Here, f_3 is the canonical extension of f_2 defined in equation Equation (2.7), $- \setminus S$ is the abstraction operator defined in Definition 2.4.8, and $\mathcal{G}$ is a stateless gluing automaton that for every subset $P \subseteq S$ of internal ports, has a transition with synchronization constraint $N = \{p!, p? \mid p \in P\}$ and data constraint $g \equiv \bigwedge_{p \in P} d_{p!} = d_{p?}$. Observe that $\mathcal{G}$ essentially models all Sync channels from $p?$ to $p!$ for every $p \in S$. In this way, we reconnect the nodes that were split by our encoding of polarity.

Example 2.4.7. Let $\mathcal{R}$ be the sequential composition of two Sync channels, i.e., $\mathcal{R} = \mathcal{R}_{a,b} \bowtie \mathcal{R}_{b,c}$ where $\mathcal{R}_{x,y}$ is a Sync channel from x to y. First, we associate to $\mathcal{R}_{x,y}$ its constraint automaton with polarity

$$\mathcal{A}_{x,y} = (\{q\}, \{x!, x?, y!, y?\}, \{(q, \{x!, x?, y!, y?\}, d_{x!} = d_{y?}, q)\}, q).$$

Thus, we represent $\mathcal{R}$ by $\{\mathcal{A}_{a,b}, \mathcal{A}_{b,c}\}$. To reconnect the channel ends $b!$ and $b?$, we add a stateless gluing automaton $\mathcal{G}$ with a single transition that has a synchronization constraint $N = \{b?, b!\}$ and data-constraint $g \equiv d_{b?} = d_{b!}$. So now, the semantics of $\mathcal{R}$ is given by $\mathsf{f}_3(\exists b! \exists b?(\mathcal{A}_{a,b} \bowtie \mathcal{A}_{b,c} \bowtie \mathcal{G}))$ and consists of a stateless labelled transition system that encodes that for every observed $\delta : 2\{a, c\} \to \mathcal{D}$, we have $\delta(a!) = \delta(c?)$.

Using the incremental translation from Diagram 2.11 and α_1 and α_2 from Example 2.4.2, we obtain data-sensitive BIP architectures $\mathsf{bip}_3(\mathcal{A}_{a,b})$ and $\mathsf{bip}_3(\mathcal{A}_{b,c})$

given by $(\emptyset, \{a, b\}, \{\alpha_1\})$ and $(\emptyset, \{b, c\}, \{\alpha_2\})$, respectively. Note that b is the only internal node in $\mathcal{R}$, hence $S = \{b\}$. Now, Example 2.4.4 shows that the system $\{\mathsf{bip}_3(\mathcal{A}_{a,b}), \mathsf{bip}_3(\mathcal{A}_{b,c})\}$ composes into a single BIP architecture A given by $(\emptyset, \{a, c\}, \{(\alpha_1 * \alpha_2) \setminus b\})$. It is now easy to see that $\mathsf{f}_3(\exists b! \exists b?(\mathcal{A}_{a,b} \bowtie \mathcal{A}_{b,c} \bowtie \mathcal{G}))$ and $\mathsf{g}_3(A)$ are bisimilar. ♢

In the previous example, we stated that the incremental translation from Diagram 2.11 preserves bisimilarity, but in fact, it preserves even a stronger equivalence: isomorphism. Informally, labelled transition systems are isomorphic if their transition relations are identical modulo state renaming. Consequently, isomorphism implies bisimilarity.

Definition 2.4.9 (Isomorphism)**.** If $L_i = (Q_i, (\mathcal{D}+1)^{2P_i}, \rightarrow_i, q_i^0) \in \text{LTS}$, $i = 1, 2$, then L_1 and L_2 are *isomorphic* iff $P_1 = P_2$ and there exists a bijective function f mapping states from Q_0 to Q_1 such that $f(q_0^0) = q_1^0$ and $q_0 \xrightarrow{\delta}_0 q_0'$, for some $q_0, q_0' \in Q_0$, if and only if $f(q_0) \xrightarrow{\delta}_1 f(q_0')$.

Theorem 2.4.1. *Translation* bip_3 *is correct and compositional, i.e., Diagram 2.11 commutes modulo isomorphism of labelled transition systems.*

Proof. Let $\mathcal{A}_i = (Q_i, \mathcal{N}_i, \rightarrow_i, q_{0i})$, for $i \in \{1, \ldots, m\}$, be BIP-friendly constraint automata with polarity, and let $S = \{p \mid \{p!, p?\} \cap \mathcal{N}_i \cap \mathcal{N}_j \neq \emptyset, \text{ with } i \neq j\}$ be the set of shared ports. The state space of $\mathsf{f}_3(\exists 2S(\mathcal{A}_1 \bowtie \cdots \bowtie \mathcal{A}_m \bowtie \mathcal{G}))$ equals $Q_1 \times \cdots \times Q_m \times \{q_\mathcal{G}\}$, and the state space of $\mathsf{g}_3((\mathsf{bip}_3(\mathcal{A}_1) \oplus \ldots \oplus \mathsf{bip}_3(\mathcal{A}_m)) \setminus S)$ equals $\prod_{j \in J} Q_j$, where $J \subseteq \{1, \ldots, m\}$ is the set of indices of the BIP-friendly components that are non-synchronizing. We show that the mapping $(q_1, \ldots, q_m, q_\mathcal{G}) \mapsto (q_i)_{i \in J}$ constitutes an isomorphism between $K = \mathsf{f}_3(\exists 2S(\mathcal{A}_1 \bowtie \cdots \bowtie \mathcal{A}_m \bowtie \mathcal{G}))$ and $L = \mathsf{g}_3((\mathsf{bip}_3(\mathcal{A}_1) \oplus \ldots \oplus \mathsf{bip}_3(\mathcal{A}_m)) \setminus S)$.

Let $\tau = ((q_1, \ldots, q_m, q_\mathcal{G}), \delta, (q_1', \ldots, q_m', q_\mathcal{G}))$ be a transition in K. Using Definition 2.1.9, if follows that τ is in K if and only if there exists an extension $\delta' : \bigcup_i 2\mathcal{N}_i \to \mathcal{D} + 1$ of δ with $\delta'(p) = \delta(p)$ for all $p \in (\bigcup_i 2\mathcal{N}_i) \setminus 2S$ such that $((q_1, \ldots, q_m, q_\mathcal{G}), \delta', (q_1', \ldots, q_m', q_\mathcal{G}))$ is a transition in $\mathsf{f}_3(\mathcal{A}_1 \bowtie \cdots \bowtie \mathcal{A}_m \bowtie \mathcal{G})$. Write $\delta'|_{2\mathcal{N}_i}$ for the restriction of δ' to $2\mathcal{N}_i$. Using Definition 2.1.8, it follows that τ is in K if and only if $\tau_i = (q_i, \delta'|_{2\mathcal{N}_i}, q_i')$ is a transition in $\mathsf{f}_3(\mathcal{A}_i)$ or $\text{dom}(\delta') \cap 2\mathcal{N}_i = \emptyset$ and $q_i' = q_i$, for all $i \in \{1, \ldots, m\}$, and $\delta'(p!) = \delta'(p?)$, for all $p \in S$, due to the gluing automaton $\mathcal{G}$. Using equations Equation (2.10) and Equation (2.8), we have that τ is in K if and only if $\mathsf{g}_3(\mathsf{bip}_3(\mathcal{A}_i))$ has a transition τ_i or $\text{dom}(\delta') \cap \mathcal{N}_i = \emptyset$ and $q_i' = q_i$, for all $i \in \{1, \ldots, m\}$, and $\delta'(p!) = \delta'(p?)$, for all $p \in S$. By the definition of the composition operator on data-sensitive BIP architectures in Definition 2.4.7 and the definition of g_3 in Definition 2.4.5, it follows that τ is in K if and only if $((q_i)_{i \in J}, \delta', (q_i')_{i \in J})$ is a transition in $\mathsf{g}_3(\mathsf{bip}_3(\mathcal{A}_1) \oplus \ldots \oplus \mathsf{bip}_3(\mathcal{A}_m))$ and $\delta'(p!) = \delta'(p?)$, for all $p \in S$. Using the abstraction operator in Definition 2.4.2, it follows that τ is in K if and only if $((q_i)_{i \in J}, \delta, (q_i')_{i \in J})$ is a transition in L. Since $\mapsto$ trivially preserves initial states, we conclude that $\mapsto$ is an isomorphism which proves the theorem. □

Applying Theorem 2.4.1 for $m = 1$, we obtain, since $S = \emptyset$, correctness of bip_3.

Corollary. $\mathsf{g}_3(\mathsf{bip}_3(\mathcal{A})) \cong \mathsf{f}_3(\mathcal{A})$*, for all CA with polarity* $\mathcal{A}$*.*

BIP to Reo Let $\{A_1, \ldots, A_n\}$ be a set of data-sensitive BIP architectures, and assume no two atomic components share a port. Our goal is to translate the composition $A_1 \oplus \cdots \oplus A_n$ to a constraint automaton with polarity by translating each BIP architecture A_i individually. To this end, we extend the translation reo_2 to data-sensitive BIP architectures.

Let $A = (\{C_1, \ldots, C_n\}, P, \Gamma)$ be a data-sensitive BIP architecture. Trivially, every atomic component C_i constitutes a constraint automaton with polarity. By reusing our translation reo_2, we define

$$\mathsf{reo}_3(A) = \mathsf{reo}_2(\Gamma) \bowtie \prod_{i=1}^{n} C_i. \tag{2.12}$$

Let $\{A_1, \ldots, A_n\}$ be a set of data-sensitive BIP architectures, and assume no two atomic components share a port. The following diagram illustrates the working of the incremental translation from BIP to Reo:

$$\begin{array}{ccccc} \{A_1, \ldots, A_n\} & & \xrightarrow{\quad\mathsf{reo}_3\quad} & & \{\mathsf{reo}_3(A_1), \ldots, \mathsf{reo}_3(A_n)\} \\ \downarrow & & & & \downarrow \\ A_1 \oplus \cdots \oplus A_n & \xrightarrow{\mathsf{g}_3} & L & \xleftarrow{\mathsf{f}_3} & \mathsf{reo}_3(A_1) \bowtie \cdots \bowtie \mathsf{reo}_3(A_n) \end{array} \tag{2.13}$$

Example 2.4.8. Consider the atomic component $C_{42} = (\{q\}, \{b!, b?\}, \rightarrow, q)$, with $\rightarrow = \{(q, \{b!, b?\}, d_{b!} = 42, q)\}$, and let α_1 and α_2 be the BIP interaction expressions from Example 2.4.3. Now, consider the data-sensitive BIP architectures $A_1 = (\{C_{42}\}, \{a, b\}, \{\alpha_1\})$ and $A_2 = (\emptyset, \{b, c\}, \{\alpha_2\})$ over the data domain $\mathcal{D} = \{0, \ldots, 2^{32} - 1\}$. Then, $\mathsf{g}_3(A_1 \oplus A_2)$ is given by a stateless labelled transition system that encodes that for every observed $\delta : 2\{a, b, c\} \to \mathcal{D}$ we have $\delta(a?) = \max(\delta(a!), \delta(b!))$, $\delta(c?) = \max(\delta(b!), \delta(c!))$, $\delta(a?) = \delta(b?) = \delta(c?)$, and $\delta(b!) = 42$. Using Example 2.3.3, it follows that $\mathsf{f}_3(\mathsf{reo}_3(A_1) \bowtie \mathsf{reo}_3(A_2))$, which is equal to $\mathsf{f}_3(\mathsf{reo}_2(\{\alpha_1\}) \bowtie C_{42} \bowtie \mathsf{reo}_2(\{\alpha_2\}))$, amounts to a labelled transition system that is bisimilar to $\mathsf{g}_3(A_1 \oplus A_2)$. ◊

Theorem 2.4.2. *Translation* reo_3 *is correct and compositional, i.e., Diagram 2.13 commutes modulo isomorphism of labelled transition systems.*

Proof. Let $\{A_1, \ldots, A_n\}$ be a set of data-sensitive BIP architectures such that no two atomic components share a port. The state space of $\mathsf{g}_3(A_1 \oplus \cdots \oplus A_n)$ equals $\prod_{C \in \mathcal{C}} Q_C$, where $\mathcal{C} = \bigcup_i \mathcal{C}_{A_i}$ are the atomic components of $A_1 \oplus \cdots \oplus A_n$. The state space of $\mathsf{f}_3(\mathsf{reo}_3(A_1) \bowtie \ldots \bowtie \mathsf{reo}_3(A_n))$ equals $\{q\} \times \prod_{i=1}^{n} \prod_{C \in \mathcal{C}_{A_i}} Q_C$, where $\mathcal{C}_{A_i}$ is the set of atomic components of A_i. We show that the mapping $(q_C)_{C \in \mathcal{C}} \mapsto (q, (q_C)_{C \in \mathcal{C}_{A_i}})_{i=1}^{n}$ constitutes an isomorphism between $K = \mathsf{g}_3(A_1 \oplus \cdots \oplus A_n)$ and $L = \mathsf{f}_3(\mathsf{reo}_3(A_1) \bowtie \ldots \bowtie \mathsf{reo}_3(A_n))$.

Let $\tau = ((q_C)_{C \in \mathcal{C}}, \delta, (q'_C)_{C \in \mathcal{C}})$ be a transition in K. By definition of g_3 in Definition 2.4.5, it follows that τ is in K if and only if δ is accepted by the composed BIP interaction model Γ and $(q_C, \delta|_{P_C}, q'_C)$ in $\mathsf{f}_3(C)$ or $\mathrm{dom}(\delta) \cap P_C = \emptyset$ and $q_C = q'_C$ for all atomic components $C \in \mathcal{C}$. By definition of the composition operator on data-sensitive BIP architectures in Definition 2.4.7, it follows that τ is in K if and only if, for all $i \in \{1, \ldots, n\}$, the following conditions are satisfied: $(q, \delta|_{P_{A_i}}, q)$ is a transition in $\mathsf{g}_2(\Gamma_i)$, with Γ_i the BIP interaction model of A_i,

and $(q_C, \delta|_{P_C}, q'_C)$ in $\mathsf{f}_3(C)$ or $\mathrm{dom}(\delta) \cap P_C = \emptyset$ and $q_C = q'_C$, for all atomic components $C \in \mathcal{C}_{A_i}$. Since $\mathsf{g}_2(\Gamma_i) \cong \mathsf{f}_2(\mathsf{reo}_2(\Gamma_i))$ by Theorem 2.3.1, we conclude that τ is in K if and only if $((q, (q_C)_{C\in\mathcal{C}_{A_i}}), \delta|_{P_{A_i}}, (q, (q'_C)_{C\in\mathcal{C}_{A_i}}))$ is a transition in $\mathsf{f}_3(\mathsf{reo}_3(A_i))$. Using Definition 2.1.8, it follows that that τ is in K if and only if $((q, (q_C)_{C\in\mathcal{C}_{A_i}})_{i=1}^n, \delta, (q, (q'_C)_{C\in\mathcal{C}_{A_i}})_{i=1}^n)$ is a transition in L. Since $\mapsto$ trivially preserves initial states, we conclude that $\mapsto$ is an isomorphism, which proves the theorem. □

By applying Theorem 2.4.2 for $n = 1$, we obtain correctness of reo_3.

Corollary. $\mathsf{f}_3(\mathsf{reo}_3(A)) \cong \mathsf{g}_3(A)$*, for all data-sensitive BIP architectures* A*.*

Thus, Theorems 2.4.1 and 2.4.2 show how our proposed composition operator of Definition 2.4.7 enables us to translate between Reo connectors, modeled by constraint automata with polarity, and data-sensitive BIP architectures.

2.5 Related work

Instead of using labelled transition systems as common semantics (Figures 2.7 and 2.9), we may also choose another model for concurrent systems. The Tile Model offers such an alternative semantics for concurrent systems [GM00]. The basic idea is to associate an m-tuple of terms in n variables $(s_i(x_1, \ldots, x_n))_{i=1}^m$ over the term algebra with signature Σ to an arrow $s : \underline{n} \to \underline{m}$ in the graph with nodes from $\underline{\mathbb{N}}$. Every function symbol $s \in \Sigma$ with arity n is interpreted as an arrow $s : \underline{n} \to \underline{1}$. As Plotkin's structural operational semantics uses terms in an algebra to represent the state of a system, the Tile Model uses the arrows $s : \underline{n} \to \underline{m}$ to describe the configuration of a concurrent system. Transitions from one configuration to another are formulated by means of tiles. A tile α (denoted by $\alpha : s \overset{a}{\underset{b}{\to}} t$) is a diagram

$$\begin{array}{ccc} \underline{n} & \xrightarrow{\ s\ } & \underline{m} \\ a\downarrow & \alpha & \downarrow b \\ \underline{p} & \xrightarrow[\ t\ ]{} & \underline{q} \end{array} \tag{2.14}$$

that represents a rewriting rule that states that trigger a can transform initial configuration s into the final configuration t and produce effect b. The trigger a and effect b are called the observations of α. Tiles may be composed horizontally, vertically, and in parallel, using the monoidal operator $\otimes$ on $\underline{\mathbb{N}}$ given by $\underline{n} \otimes \underline{m} = \underline{n + m}$.

A configuration can be seen as a connector. In this view, the source $\underline{n}$ and target $\underline{m}$ of a configuration $s : \underline{n} \to \underline{m}$ correspond to the interface of the connector. Since the interfaces $\underline{p}$ and $\underline{q}$ in diagram Equation (2.14) may differ from $\underline{n}$ and $\underline{m}$, the Tile Model provides a natural semantics for dynamic reconfiguration in Reo [ABC+08].

Bruni et al. show that Petri nets with boundaries are equally expressive as BIP without priorities [BMM11]. They showed that this formal correspondence indirectly relates BIP to the Tile Model, which resulted in the definition of the Petri calculus. Since boundaries are mainly used for composition, the monolithic

translation by Bruni et al. encodes BIP without priorities into Petri nets without boundaries. A similar encoding exists for Reo, which translates port automata into Petri nets [Kra09].

An indirect comparison of BIP and Reo, in the data-agnostic domain, through their respective comparisons with other models, e.g., Petri nets, is certainly possible. Nevertheless, the direct and formal translations we present in this chapter allow direct translation tools between BIP and Reo, that are otherwise difficult, if not impossible, to construct based on such indirect comparisons.

Beside BIP and Reo, there are many other examples of coordination languages [PA98]. Their relations with BIP and Reo have been studied by others. For instance, Proença and Clarke provide a detailed comparison between Orc and Reo [PC08], Chkouri et al. present a translation of AADL into BIP [CRBS08], and Talcott et al. connect both ARC and PBRD to Reo by providing mappings between their semantic models [TSR11].

2.6 Discussion

In the data-agnostic domain, we showed that BIP architectures and port automata coincide modulo internal transitions, witnessed by the weak simulation in Theorem 2.2.2, and independent progress, witnessed by the condition $\emptyset \in \gamma_1 \cup \gamma_2$ in Theorem 2.2.4. In the data-sensitive domain, we showed by Theorem 2.3.1 that the observable behavior of BIP interaction models and stateless constraint automata is identical. We extended the notion of a data-agnostic BIP architecture to the data-sensitive domain (Definition 2.4.4), and showed that these data-sensitive BIP architectures correspond to constraint automata with polarity (Corollaries 2.4.5 and 2.4.5).

Our formal correspondences between BIP and Reo reveal differences and similarities of their fundamental design principles. One similarity is that both BIP and Reo provide constructs that allow high-level specification of multiparty synchronization, such as a barrier synchronization. Although multiparty synchronization is used in several approaches, such as the bulk-synchronous parallel (BSP) model [Val90] or the Parameterized Networks of Synchronized Automata (pNets) [BAC$^+$09], most of the process algebras lack this feature, expressing multiparty synchronization by a cluttered composition of binary synchronizations. Exceptions include Winskel's synchronization algebra [WN95] and Bergstra & Klop's algebra of communicating processes (ACP) [BK85]. Controlling and constraining multiparty synchronization is, however, more complex in ACP than it is in BIP and Reo (because additional operators, communication and block, need to be used beside parallel composition to specify admissible synchronizations). This is illustrated in work by Krause et al. [KKdV12], who encoded Reo's semantics (i.e., Reo's composition operator and a number of primitives) in mCRL2 [CGK$^+$13], a modern process specification language based on ACP.

The focus of this chapter is on formal relations between BIP and Reo. As such, detailed comparison of BIP or Reo with process algebras or other models that support multi-party synchronization is beyond our scope. However, support for multiparty synchronization in some other models, and the consensus in BIP and Reo to support this notion through first-order constructs confirms the practical

significance of this concept.

On the other hand, BIP and Reo treat the separation between computation and coordination differently. The BIP framework concretely *defines* what separates computation (BIP behavior) from coordination (BIP interaction), while Reo merely *separates* computation (Reo components) and coordination (Reo connector) structurally. Indeed, Reo does not force a fixed universal definition for computation and coordination in all applications. Without giving a fixed definition of separation criterion, Reo's structural separation of computation from coordination (i.e., component versus connector) simply means that, while this separation is always important, the distinction between the two is in the eye of the beholder: in different applications, different, or even the same people, may find it convenient to draw the line that separates computation and coordination at different places to suit their needs. For example, the stateful behavior of a FIFO with capacity of 1 strictly places what this entity does in the behavior layer of BIP, as a (computation) component. In Reo, such stateful components can, of course, be regarded and used as computation as well. However, when deemed appropriate, one can use the same component (i.e., a FIFO_1 channel) in the construction of a Reo connector as well, e.g., to express the stateful, turn-taking interaction between two components, as in Figure 2.4.

The property-preserving translations presented in this chapter enable us to lift the composition operator for data-sensitive Reo circuits to BIP architectures. Besides lifting theoretical results, it seems natural to investigate whether it is possible to transfer also other techniques, such as those used in compilation and model checking. For example, Reo's compositional approach to code generation [Jon16] may yield a very different distributed implementation of a BIP system. Comparing the performance of such a postulated implementation of BIP, can reveal valuable insights for compilation.

The results in this chapter show that both BIP and Reo can be interpreted as a variant of labeled transitions sytems (LTS). In the sequel of this thesis, we work for the large part with the semantics of coordination languages. In fact, one of our main contributions consists in developing coordination language semantics that improve expressiveness (Chapters 3 and 7) and tooling (Chapters 5 and 6). The only exception is Chapter 4, where we develop a syntax to build actual systems with these new semantics. Here, we adopt Reo's compositionally principle as a basis of our syntax.

Chapter 3

Protocols with Preferences

The comparison between BIP en Reo in the Chapter 2 shows that priority layer in BIP has no clear counterpart in graphical Reo language. Although Reo designers recognize the need for language constructs that express priority, the straightforward approach to priority taken in BIP is not satisfactory, because BIP lacks a composition operator that propagates local priorities to global ones. Such a priority composition operator is required for a full priority model in Reo, because the semantics of larger Reo connectors is defined as the composition of the semantics of its channels and nodes. The current chapter aims to partially resolve the discrepancy between BIP and Reo by further developing a (compositional) semantics for Reo connectors that includes a notion preference[1].

The literature offers several semantic formalisms to express the behavior of Reo connectors. The work in [JA12] collects, classifies, and surveys around thirty semantics based on *co-algebraic* or *coloring* techniques, and other models based on, for instance, *constraints* and *Petri nets*. The *operational* models (i.e., automata) are probably the most popular approaches: the main classes are represented by *constraint automata*, and (several) related variants, and *context-sensitive automata*.

The aim of the current work is to generalize *soft constraint automata* [AS12] or *soft component automata* [KAT16, KAT17] (SCA in both cases), which is a variant of constraint automata that is developed after the publication of the survey [JA12]. An SCA is a state-transition system where transitions are labelled with actions and preferences. Higher-preference transitions typically contribute more towards the goal of the component.

The contribution of this chapter is twofold. First, we relax the definition of the underlying structure that models preferences. Instead of semirings (as in [AS12, KAT16, KAT17]), we use complete lattice monoids (see Section 3.1.1) to allow bipolar preferences. Since the unit element, $\mathbf{1}$, is the only sensible preference of an idling transition, any useful transition must have a positive preference $p > \mathbf{1}$.

Second, we extend SCA with a notion of memory (SCAM), as already accomplished for (non-soft) constraint automata [JKA17]. Each transition of a SCAM can also impose a condition on the current data assigned to a finite set of *memory locations*, and update their respective values. Therefore, together with states,

[1] The work in this chapter is based on [DGLS21, DGS18]

memory locations determine the configuration of a connector, and influence its observable behavior.

The outline of the chapter is as follows: Section 3.1 defines soft constraints and shows that they can be compared and composed, and their variables renamed and hidden. Section 3.2 introduces soft constraint automata with memory (SCAMs) and their interpretation as soft constraints. We define their composition and hiding, and show their correctness with respect to the soft constraint semantics. Section 3.3 presents a case study illustrating the composition and hiding operations on SCAMs. Section 3.4 offers a novel encoding of context-sensitive behavior based on SCAMs. Finally, Section 3.5 summarizes the related work on different semantics proposed for CA, and Section 3.6 wraps up with conclusive thoughts and hints about future research.

3.1 Preliminaries on soft constraints

In contrast to Boolean constraints, a *soft constraint* is a constraint that need not be fully satisfied [BMR97]. Instead, such a constraint assigns a *preference value* that measures the degree of satisfaction of a solution. The goal is to find a solution that maximizes this preference value.

The structure of this section is as follows: Section 3.1.1 proposes *complete lattice monoids* (CLMs) to serve as a structured domain of preference values, which allows us to compare and compose preference values. Section 3.1.2 develops a complete lattice monoid of streams equipped with a lexicographic order. We use this construction in the semantics of soft constraint automata in Section 3.2. Section 3.1.3 presents our personal take on cylindric and diagonal operators [SRP91]: they are mostly drawn with minor adjustments from [GSPV17]. Section 3.1.4 shows that the set of soft constraints is itself a complete lattice monoid that admits cylindric and diagonal operators. As such, soft constraints can be compared and composed, and variables in a soft constraint can be renamed and hidden.

3.1.1 Complete lattice monoids

The first step is to define an algebraic structure that models preference values. We refer to [GS17] for the missing proofs as well as for an introduction to bipolar preferences and a comparison with other proposals.

Definition 3.1.1 (Partial order)**.** A partial order (PO) is a pair $\langle A, \leq \rangle$, such that A is a set and $\leq \subseteq A \times A$ is a reflexive, transitive, and anti-symmetric relation. A complete lattice (CL) is a PO, such that any subset of A has a least upper bound (LUB).

The LUB of a subset $X \subseteq A$ is denoted as $\bigvee X$, and it is unique by anti-symmetry of $\leq$. Note that $\bigvee A$ and $\bigvee \emptyset$ correspond respectively to the *top*, denoted as $\top$, and to the *bottom*, denoted as $\bot$, of the CL.

Definition 3.1.2 (Complete lattice monoid)**.** A (commutative) monoid is a triple $\langle A, \otimes, \mathbf{1} \rangle$, such that $\otimes : A \times A \to A$ is a commutative and associative operation and $\mathbf{1} \in A$ is its identity element.

A partially ordered monoid (POM) is a 4-tuple $\langle A, \leq, \otimes, \mathbf{1}\rangle$, such that $\langle A, \leq\rangle$ is a PO and $\langle A, \otimes, \mathbf{1}\rangle$ a commutative monoid. A complete lattice monoid (CLM) is a POM, such that its underlying PO is a CL.

As usual, we use the infix notation: $a \otimes b$ stands for $\otimes(a, b)$.

According to Definition 3.1.2, the partial order $\leq$ and the product $\otimes$ can be unrelated. This is not the case for monotone CLMs.

Definition 3.1.3 (Monotonicity)**.** A CLM $\langle A, \leq, \otimes, \mathbf{1}\rangle$ is monotone if and only if, for all $a, b, c \in A$, we have that $a \leq b$ implies $a \otimes c \leq b \otimes c$.

Our framework (Lemma 3.1.6) requires a condition that is slightly stronger than monotonicity:

Definition 3.1.4 (Distributivity)**.** A CLM $\langle A, \leq, \otimes, \mathbf{1}\rangle$ is distributive if and only if, for every element $a \in A$ and every subset $X \subseteq A$, we have

$$a \otimes \bigvee X = \bigvee \{a \otimes x \mid x \in X\}.$$

Note that $a \leq b$ is equivalent to $\bigvee\{a, b\} = b$, for all $a, b \in A$. Hence, a distributive CLM is monotone and $\bot$ is its zero element (i.e., $a \otimes \bot = \bot$, for all $a \in A$).

Example 3.1.1 (Boolean CLM)**.** The Boolean CLM $\mathbb{B} = \langle\{0, 1\}, \leq, \times, 1\rangle$, with the usual order and multiplication, is a distributive CLM. ◊

Distributive CLMs generalize *tropical* semirings, which define their (idempotent) sum operator as $a \oplus b = \bigvee\{a, b\}$, for all $a, b \in A$. If, moreover, $\mathbf{1}$ is the top of the CL, we end up with *absorptive* semirings [Gol03](in the algebraic literature) or *c*-semirings [BMR97] (in the soft constraint literature). See [BG06] for a brief survey on residuation for such semirings. Together with monotonicity, imposing $\mathbf{1}$ to coincide with $\top$ means that preferences are *negative*(i.e., $a \leq \mathbf{1}$, for all $a \in A$). Since we allow the top of the CL to be strictly *positive*(i.e., $\mathbf{1} < \top$), our approach based on complete lattice monoids falls into the category of *bipolar* approaches.

Example 3.1.2 (Bipolar CLM)**.** The bipolar CLM $\mathbb{K} = \langle\{0, 1, \infty\}, \leq, \times, 1\rangle$, with the usual order and multiplication (extended to ∞ by defining $0 \times \infty = 0$ and $1 \times \infty = \infty \times \infty = \infty > 1$), is a distributive CLM. ◊

Example 3.1.3 (Power set)**.** Given a (possibly infinite) set V of variables, we consider the monoid $\langle 2^V, \cup, \emptyset\rangle$ of (possibly empty) subsets of V, with union as the monoidal operator. Since the operator is idempotent ($a \otimes a = a$, for all $a \in A$), the natural order ($a \leq b \Leftrightarrow a \otimes b = b$, for all $a, b \in A$) is a partial order, and it coincides with subset inclusion: in fact, the power set $\langle 2^V, \subseteq, \cup, \emptyset\rangle$ is a CLM. Moreover, since the LUB, $\bigvee$, and the product, $\otimes$, both model set-union, the power set CLM is distributive. ◊

Example 3.1.4 (Extended integers)**.** The extended integers $\langle \mathbb{Z} \cup \{\pm\infty\}, \leq, +, 0\rangle$, where $\leq$ is the natural order, such that, for all $k \in \mathbb{Z}$,

$$-\infty \;\leq\; k \;\leq\; +\infty,$$

$+$ is the natural addition, such that, for all $k \in \mathbb{Z} \cup \{+\infty\}$,

$$\pm\infty + k \;=\; \pm\infty, \qquad\qquad +\infty + (-\infty) \;=\; -\infty,$$

and 0 the identity element, constitutes a distributive CLM. Here, $+\infty$ and $-\infty$ are respectively the top and the bottom element of the CL. ◊

The following construction allows us to compose primitive CLMs (such as those in Examples 3.1.1 to 3.1.4) into more complex CLMs:

Definition 3.1.5 (Cartesian product)**.** The Cartesian product of two CLMs $\langle A_1, \leq_1, \otimes_1, \mathbf{1}_1\rangle$ and $\langle A_2, \leq_2, \otimes_2, \mathbf{1}_2\rangle$ is the CLM $\langle A_1 \times A_2, \leq, \otimes, (\mathbf{1}_1, \mathbf{1}_2)\rangle$, such that, for all $(a_1, a_2), (b_1, b_2) \in A_1 \times A_2$,

1. $(a_1, a_2) \leq (b_1, b_2)$ if and only if $a_1 \leq_1 b_1$ and $a_2 \leq_2 b_2$;
2. $(a_1, a_2) \otimes (b_1, b_2) = (a_1 \otimes_1 b_1, a_2 \otimes_2 b_2)$.

Note that the Cartesian product is a well-defined CLM.

Lemma 3.1.1. *The Cartesian product of distributive CLMs is distributive.*

3.1.2 Streams of preferences

We now introduce the CLM of streams, which we use for our semantics of SCAMs in Definition 3.2.5. Here we generalize the results in [GHMW13] on binary lexicographic operators. In the following, we denote by A^ω the set of streams(infinite sequences) of elements of A.

Definition 3.1.6 (Lexicographic order)**.** Let $\langle A, \leq\rangle$ be a PO. The lexicographic order $\leq_l$ on A^ω is given by

$$a_0 a_1 \cdots \leq_l b_0 b_1 \cdots \quad \text{iff} \quad \begin{cases} \forall i.\, a_i = b_i & \vee \\ \exists j.\, a_j < b_j \wedge \forall i < j.\, a_i = b_i \end{cases}$$

We write $<_l$ for the usual strict version of the lexicographic order $\leq_l$.

The following lemma provides a recursive description of LUBs in lexicographically ordered streams:

Lemma 3.1.2. *Let $\langle A, \leq\rangle$ be a CL and $X \subseteq A^\omega$ a subset of the PO $\langle A^\omega, \leq_l\rangle$. Then, $\bigvee X = x_0 x_1 x_2 \cdots \in A^\omega$ exists and satisfies, for all $i \geq 0$, the recursion*

$$x_i = \bigvee \{b_i \mid x_0 x_1 \cdots x_{i-1} b_i b_{i+1} \cdots \in X\}.$$

Proof. Define $x_0 x_1 \cdots \in A^\omega$ using the recursion in the lemma.

We prove that $x_0 x_1 \cdots$ is an upper bound of X. Let $a_0 a_1 \cdots \neq x_0 x_1 \cdots$ be in X. Find the smallest $i \geq 0$, such that $a_i \neq x_i$. Then, we have $a_i \in \{b_i \mid x_0 x_1 \cdots x_{i-1} b_i b_{i+1} \cdots \in X\}$, and $a_i < x_i$. Thus, $a_0 a_1 \cdots \leq_l x_0 x_1 \cdots$.

We prove that $x_0 x_1 \cdots$ is minimal. Let $u_0 u_1 \cdots \neq x_0 x_1 \cdots$ be an upper bound of X. Find the smallest $i \geq 0$, such that $u_i \neq x_i$. For every $x_0 \cdots x_{i-1} b_i \cdots \in X$, we have $x_0 \cdots x_{i-1} b_i \cdots \leq_l u_0 u_1 \cdots = x_0 \cdots x_{i-1} u_i \cdots$, which implies $b_i \leq u_i$. Hence, $x_i = \bigvee \{b_i \mid x_0 \cdots x_{i-1} b_i \cdots \in X\} \leq u_i$, and $x_0 x_1 \cdots \leq_l u_0 u_1 \cdots$.

We conclude that $\bigvee X = x_0 x_1 \cdots$, which proves the result. □

Let $\otimes^\omega$ be the operator on data stream given by the point-wise application of $\otimes$, and let $\mathbf{1}^\omega$ the data stream composed just by $\mathbf{1}$. Lemma 3.1.2 shows that $\langle A^\omega, \leq_l, \otimes^\omega, \mathbf{1}^\omega\rangle$ is a CLM. However, it turns out (Lemma 3.1.3) that this CLM is not distributive due to the presence of *non-cancellative* (or *collapsing*) elements in A:

Definition 3.1.7 (Cancellative elements)**.** An element c in a CLM $\langle A, \leq, \otimes, \mathbf{1}\rangle$ is cancellative if and only if there $a \otimes c = b \otimes c$ implies $a = b$, for all $a, b \in A$.

In any distributive CLM, $\bot$ is a non-cancellative element.

Lemma 3.1.3. *The CLM $\langle A^\omega, \leq_l, \otimes^\omega, \mathbf{1}^\omega\rangle$ is not distributive.*

Proof. Let c be a non-cancellative element in a distributive CLM $\langle A, \leq, \otimes, \mathbf{1}\rangle$. By definition, we find distinct elements $a, b \in A$, with $a \otimes c = b \otimes c$. Without loss of generality, we may assume that $a < b$ (otherwise, take $b' = \bigvee\{a, b\}$ and use distributivity to show that $a \otimes c = b' \otimes c$). Point-wise multiplication of the inequality $a\top^\omega <_l b\bot^\omega$ by $c1^\omega$ yields

$$b\bot^\omega \otimes^\omega c1^\omega \;=\; (b \otimes c)\bot^\omega \;<_l\; (a \otimes c)\top^\omega \;=\; a\top^\omega \otimes^\omega c1^\omega,$$

which contradicts monotonicity (and, hence, distributivity). □

To obtain a distributive CLM of streams, we restrict its carrier, A^ω, to a suitable subset. Let A_ι be the set of cancellative elements, and let A_c be the set of non-cancellative elements. The following example shows that the subset $A_\iota^\omega \subseteq A^\omega$ of streams of cancellative elements is not a suitable domain for the CLM of streams, as it is not closed under LUBs.

Example 3.1.5. Let $\langle A, \leq, +, 0\rangle$, with $A = \mathbb{Z} \cup \{\pm\infty\}$, be the CLM of extended integers from Example 3.1.4. Observe that $A_\iota = \mathbb{Z}$ and $A_c = \{\pm\infty\}$. Although $\emptyset$ and $\{1^\omega, 2^\omega, 3^\omega, \cdots\}$ are subsets of A_ι^ω, their respective LUBs are (by Lemma 3.1.2) equal to $(-\infty)^\omega$ and $(+\infty)(-\infty)^\omega$, and not included in A_ι^ω. ◊

To ensure that the set of streams is closed under LUBs, we further include elements of the shape $A_\iota^* A_c \bot^\omega$: streams prefixed by a (possibly empty) finite sequence of cancellative elements, then followed by a single occurrence of a non-cancellative element, and then closed by an infinite sequence of $\bot$.

Theorem 3.1.4 (Lexicographic CLM)**.** *If $\mathbb{S} = \langle A, \leq, \otimes, \mathbf{1}\rangle$ is a distributive CLM, then $\mathbb{S}^\omega = \langle A_\iota^\omega \cup A_\iota^* A_c \bot^\omega, \leq_l, \otimes^\omega, \mathbf{1}^\omega\rangle$ is so.*

Proof. The POM $\mathbb{S}^\omega$ is a CLM, because its carrier $B = A_\iota^\omega \cup A_\iota^* A_c \bot^\omega$ is closed with respect to LUBs: Let $X \subseteq B$ and $\bigvee X = x_0 x_1 \cdots$. Suppose that $x_i \in A_c$, for some $i \geq 0$. Let $j > i$ be arbitrary, and consider the set $X_j = \{b \mid x_0 \cdots x_i \cdots x_{j-1} b \cdots \in X\}$. Since $X \subseteq B$ and $x_i \in A_c$, we have either $X_j = \emptyset$ or $X_j = \{\bot\}$. By Lemma 3.1.2, we have $x_j = \bigvee X_j = \bot$, and we conclude that $\bigvee X \in B$.

Next, we show that the CLM $\mathbb{S}^\omega$ is distributive, i.e., that $a \otimes^\omega \bigvee X = \bigvee\{a \otimes^\omega x \mid x \in X\}$. Let $X \subseteq B$ be a subset of the carrier, and $a \in B$. Let also $p = \bigvee X$ and $q = \bigvee\{a \otimes^\omega x \mid x \in X\}$: we show by induction that $a_i \otimes p_i = q_i$ for all $i \geq 0$. So, let us suppose that $a_j \otimes p_j = q_j$ for all $0 \leq j < i$, which is vacuously true for $i = 0$.

Lemma 3.1.2 shows that $q_i = \bigvee S$, with $S = \{b \mid q_0 \cdots q_{i-1} b \cdots \in \{a \otimes^\omega x \mid x \in X\}\}$. We distinguish two cases:

Case 1: Suppose that some a_j, $0 \leq j < i$, is non-cancellative. Then, $a \in B$ implies that $a_i = \bot$. Hence, we have either $S = \emptyset$ or $S = \{\bot\}$, and we find $q_i = \bigvee S = \bot = a_i \otimes p_i$.

Case 2: Suppose that all a_j, $0 \leq j < i$, are cancellative. Then,

$$q_i = \bigvee S = \bigvee \{a_i \otimes x_i \mid x_0 x_1 \cdots \in X \wedge a_j \otimes x_j = q_j \text{ for all } j < i\}.$$

Distributivity of the original CLM and the induction hypothesis implies

$$q_i = a_i \otimes \bigvee \{x_i \mid x_0 x_1 \cdots \in X, a_j \otimes x_j = a_j \otimes p_j, \text{ for all } j < i\}.$$

Applying Lemma 3.1.2 yields

$$q_i = a_i \otimes \bigvee \{x_i \mid p_0 \cdots p_{i-1} x_i \cdots \in X\} = a_i \otimes p_i.$$

In both cases, we find $q_i = a_i \otimes p_i$, which completes the proof. □

Example 3.1.6. Looking at the CLM $\langle \mathbb{Z} \cup \{\pm\infty\}, \leq, +, 0\rangle$ of extended integers from Example 3.1.4 and Example 3.1.5, the set of elements of the associated lexicographic CLM is $\mathbb{Z}^\omega \cup \mathbb{Z}^*\{\bot, \top\}\bot^\omega$. ◊

3.1.3 Cylindric operators for ordered monoids

We introduce two families of operators on CLMs, which enable hiding and renaming of variables (cf., [GSPV17, GSPV15]). The first family is parameterized by a cylindric operator, and models existential quantification. The second family is parameterized by a diagonal operator, and models of equality of variables. Cylindric and diagonal operators originate in the context of cylindric algebras [HMT81], and entered the constraint literature via [SRP91].

Definition 3.1.8 (Pomonoid action)**.** Let $\mathbb{S} = \langle A, \mathit{popl}91\ \leq, \otimes, \mathbf{1}\rangle$ be a CLM and let $\mathbb{P} = \langle E, \leq\rangle$ a PO. An action of $\mathbb{S}$ on $\mathbb{P}$ is a function $\phi : A \times E \to E$, such that, for all $a, b \in A$ and all $e \in E$,

1. $\phi(\mathbf{1}, e) = e$,
2. $\phi(a, \phi(b, e)) = \phi(a \otimes b, e)$,
3. $a \leq b \implies \phi(a, e) \leq \phi(b, e)$.

The first two requirements state that ϕ is a monoid action of $\mathbb{S}$ on E, and the third one states that ϕ is monotone in the first argument.

Let V be a set of variables, and recall the power set CLM $\langle 2^V, \subseteq, \cup, \emptyset\rangle$ from Example 3.1.3. Consider a CLM $\langle A, \leq, \otimes, \mathbf{1}\rangle$, whose elements $a \in A$ can be thought of as expressions with variables from V. The partial order, $\leq$, the product, $\otimes$, and the identity, $\mathbf{1}$, can be thought of as implication, conjunction, and tautology, respectively. The following definition axiomatizes existential quantification for these expressions.

Definition 3.1.9 (Cylindric operator and support)**.** A cylindric operator $\exists$ over a CLM $\langle A, \leq, \otimes, \mathbf{1}\rangle$ and set of variables V is an action $\exists : 2^V \times A \to A$, such that, for all $X \subseteq V$, all $a, b \in A$, and all $C \subseteq A$,

1. $\exists(X, \mathbf{1}) = \mathbf{1}$,
2. $\exists(X, a \otimes \exists(X, b)) = \exists(X, a) \otimes \exists(X, b)$,
3. $\exists(X, \bigvee C) = \bigvee\{\exists(X, c) \mid c \in C\}$.

The *support* of $a \in A$ is the set of variables $\mathrm{supp}(a) = \{x \mid \exists(\{x\}, a) \neq a\}$.

In the following, we use $\exists_X a$ for $\exists(X, a)$ and $\exists_x a$, when $X = \{x\}$.

Item 3 in Definition 3.1.9 is required for the correctness proofs of SCAM operations on SCAMs (Theorems 3.2.1 and 3.2.2). Item 3 implies monotonicity of $\exists$ in the second argument ($a \leq b$ implies $\exists_X a \leq \exists_X b$, for all $X \subseteq V$ and all $a, b \in A$). By Definition 3.1.8 it holds that $a = \exists_\emptyset a \leq \exists_X a$ and $X \cap \mathrm{supp}(\exists(X, a)) = \emptyset$.

Next, we axiomatize expressions that equate two variables:

Definition 3.1.10 (Diagonalisation)**.** Let $\exists$ be a cylindric operator over a CLM $\langle A, \leq, \otimes, \mathbf{1}\rangle$ and a set of variables V. A diagonal operator δ for $\exists$ is a family of idempotent elements $\delta_{x,y} \in A$, indexed by pairs of variables in V, such that, for all $x, y, z \in V$ and $a \in A$,

1. $\delta_{x,x} = \mathbf{1}$,
2. $\delta_{x,y} = \delta_{y,x}$,
3. $z \notin \{x, y\} \implies \delta_{x,y} = \exists_z(\delta_{x,z} \otimes \delta_{z,y})$,
4. $x \neq y \implies \delta_{x,y} \otimes \exists_x(a \otimes \delta_{x,y}) \leq a$.

Axioms 1, 2, and 3 plus idempotency of $\delta_{x,y}$ imply $\exists_x \delta_{x,y} = \mathbf{1}$, which in turn implies (using also idempotency of $\exists_X$) $\mathrm{supp}(\delta_{x,y}) = \{x, y\}$ for $x \neq y$.

Diagonal operators can be used for modelling variable substitution [GSPV17]: substituting y for $x \neq y$ in a yields $\exists_x(a \otimes \delta_{x,y})$.

3.1.4 Soft constraints (on infinite domains)

We define the notion of soft constraints, following the approach in [BMR06], but generalizing the preference structure, as in [GSPV17, GSPV15]. Soft constraints are expressions that evaluate to a value in a given CLM. They generalize *crisp* constraints, which are expressions that evaluate into the Boolean CLM.

Definition 3.1.11 (Soft constraints)**.** Let V be a set of variables, D a domain of interpretation and $\mathbb{S} = \langle A, \leq, \otimes, \mathbf{1}\rangle$ a CLM. A *soft constraint* is a function $c : (V \to D) \to A$ associating a value in A with each assignment $\eta : V \to D$ of the variables.

We write $\mathcal{C}(V, D, \mathbb{S})$ for the set of all such soft constraints.

We write $c\eta$ to denote the application of a constraint $c : (V \to D) \to A$ to a variable assignment $\eta : V \to D$.

Definition 3.1.11 does not impose any restriction on the number of variables and the size of the domain of interpretation. In fact, our framework requires infinitely many *timed variables*, as introduced in Section 3.2.1. Different from standard practice in soft constraint literature, we also consider a possibly infinite domain of interpretation, D, which necessitates the introduction of memory locations in Definition 3.2.3.

In the following example, we introduce notation to view preference values and Boolean constraints as soft constraints.

Example 3.1.7 (Constant constraints)**.** A preference value $a \in A$ induces a soft constraint $[a]$ defined as $[a]\eta = a$, for every assignment $\eta : V \to D$. ◊

Example 3.1.8 (Boolean constraints)**.** A Boolean constraint B induces a soft constraint $[B]$ defined, for every assignment $\eta : V \to D$, as

$$[B]\eta = \begin{cases} \mathbf{1} & \text{if } \eta \text{ satisfies } B \\ \bot & \text{otherwise} \end{cases}$$

For example, for a variable $v \in V$, a datum $d \in D$, and an assignment $\eta : V \to D$, we have $[v = d]\eta = \mathbf{1}$ if and only if $\eta(v) = d$. Since conjunction with a tautology should act as the identity, we choose $\mathbf{1}$ instead of $\top$. ◊

The set of constraints forms a CLM, with the structure lifted from $\mathbb{S}$.

Lemma 3.1.5 (The CLM of constraints)**.** *The set $\mathcal{C}(V, D, \mathbb{S})$ of soft constraints, endowed with partial order $\leq$, composition $\otimes$, and unit $[\mathbf{1}]$ defined as*

1. *$c_1 \leq c_2$ if $c_1\eta \leq c_2\eta$ for all $\eta : V \to D$*
2. *$(c_1 \otimes c_2)\eta = c_1\eta \otimes c_2\eta$*

is a CLM denoted as $\mathbb{C}(V, D, \mathbb{S})$. The LUB of a subset $C \subseteq \mathbb{C}(V, D, \mathbb{S})$ satisfies $(\bigvee C)\eta = \bigvee\{c\eta \mid c \in C\}$, for all assignments $\eta : V \to D$. The CLM $\mathbb{C}(V, D, \mathbb{S})$ is distributive if $\mathbb{S}$ is so.

Combining constraints using the $\otimes$ operator builds a new constraint whose support involves at most the variables of the original ones. The composite constraint associates, with every assignment, a preference that is equal to the product of the preferences of its constituents.

Note that, in a bipolar setting, we do not have *conjunction elimination*: for constraints $c_1, c_2 \in \mathcal{C}(V, D, \mathbb{S})$, $c_2 > \mathbf{1}$, monotonicity implies $c_1 \otimes c_2 > c_1$, where $\leq$ is interpreted as implication.

Given a function $\eta : V \to D$ and a set $X \subseteq V$, we denote by $\eta|_X : X \to D$ the usual restriction.

Lemma 3.1.6 (Cylindric and diagonal operators for constraints)**.** *If $\mathbb{S}$ is a distributive CLM, then the CLM of constraints $\mathbb{C}(V, D, \mathbb{S})$ admits a diagonal operator $\delta_{x,y} = [x = y]$, for all $x, y \in V$, and a cylindric operator $\exists_X$, defined, for all soft constraints $c \in \mathcal{C}(V, D, \mathbb{S})$ and all subsets $X \subseteq V$ of variables, as*

$$(\exists_X c)\eta = \bigvee\{c\rho \mid \rho|_{V\setminus X} = \eta|_{V\setminus X}\}.$$

Proof. Using the definitions from Example 3.1.8, Lemma 3.1.5, and distributivity of $\mathbb{S}$, it is straightforward to verify that all axioms in Definitions 3.1.8 to 3.1.10 are satisfied. For example, for all soft constraints $c, d \in \mathcal{C}(V, D, \mathbb{S})$ and all $X \subseteq V$, we have, for all assignments $\eta : V \to D$, that

$$\begin{aligned}(\exists_X(c \otimes \exists_X d))\eta &= \bigvee\{c\rho \otimes (\exists_X d)\rho \mid \rho|_{V\setminus X} = \eta|_{V\setminus X}\} \\ &= \bigvee\{c\rho \otimes (\bigvee\{d\xi \mid \xi|_{V\setminus X} = \eta|_{V\setminus X}\}) \mid \rho|_{V\setminus X} = \eta|_{V\setminus X}\} \\ &= \bigvee\{c\rho \mid \rho|_{V\setminus X} = \eta|_{V\setminus X}\} \otimes \bigvee\{d\xi \mid \xi|_{V\setminus X} = \eta|_{V\setminus X}\} \\ &= (\exists_X c \otimes \exists_X d)\eta,\end{aligned}$$

which shows that $\exists_X(c \otimes \exists_X d) = \exists_X c \otimes \exists_X d$. □

Hiding removes variables from the support: $\mathrm{supp}(\exists_X c) \subseteq \mathrm{supp}(c) \setminus X$.[2] Note that both the infinite number of variables and the infinite domain of interpretation necessitate the existence of LUBs in the definition of $\exists_X c$, which motivates the introduction of complete lattices in the previous section.

Although a soft constraint c evaluates mappings $\eta : V \to D$ that assign a value in D to every variables in V, the evaluation $c\eta$ may depend on the assignment of a (finite) subset of them, called its support. The cylindric operator from Lemma 3.1.6, together with Definition 3.1.9, provides a precise characterization of the support of a soft constraint. For instance, a binary constraint c with $\mathrm{supp}(c) = \{x, y\}$ is a function $c : (V \to D) \to A$ that depends only on the assignment of variables $\{x, y\} \subseteq V$, meaning that two assignments $\eta_1, \eta_2 : V \to D$ that differ only for the image of variables $z \notin \{x, y\}$ coincide (i.e., $c\eta_1 = c\eta_2$). The support corresponds to the classical notion of *scope* of a constraint.

3.2 Soft constraint automata with memory

Constraint automata have been introduced in [BSAR06] as a formalism to describe the behavior and data flow in coordination models(such as the Reo language [BSAR06]); they can be considered as acceptors of *timed data streams* [BSAR06, DGS18, AR02]. Constraint automata have been enriched with new features to create more expressive formalisms. On the one hand, *constraint automata with memory* (CAM) enrich constraint automata with a finite set of memory locations [JKA17]. This extension allows one to handle infinite state spaces by enabling the values of each memory location to range over an infinite data domain. On the other hand, *soft constraint automata* (SCA) enrich constraint automata with soft constraints [AS12]. This extension allows one to express preference amongst different executions.

We present *soft constraint automata with memory* (SCAM): a generic framework that captures both CAM and SCA in a single formalism. Our approach differs significantly from both existing works with respect to its semantics. Originally, SCA are acceptors of tuples of *weighted timed data streams* [AS12, DGS18]. In the current work, we interpret a SCAM as a special kind of soft constraint, encoding the same information in an alternative way.

[2] The operator is called *projection* in constraints literature, and $\exists_X c$ is denoted $c \Downarrow_{V\setminus X}$.

3.2.1 Soft languages

Memory is the capacity to preserve information through time. Therefore, given a finite set of memory locations L, we model the behavior of a location $v \in L$ as an infinite sequence of *timed variables*

$$(v,0),(v,1),\ldots,(v,i),\ldots,$$

where variable (v,i) represents the value of memory location $v \in L$ at time step $i \in \mathbb{N}_0$.

We write $\widehat{L} = L \times \mathbb{N}_0$ for the set of timed variables. We define the *k-th derivative* of a variable $x = (v,i) \in \widehat{L}$ as $x^k = (v,i)^k = (v, k+i)$. We define the *k-th derivative* of a set of variables $X \subseteq \widehat{L}$ as $X^k = \{x^k \mid x \in X\}$.

For notational convenience, we treat a timed variable $(v,0) \in \widehat{L}$ and a plain variable v as equal, and we write a prime for the first derivative. For example, the expression $m' = a$ expands to an expression $(m,1) = (a,0)$ on timed variables.

Next, we extend the data domain $\mathcal{D}$ with a special symbol $* \notin \mathcal{D}$ that denotes "no-data", and write $\mathcal{D}_* = \mathcal{D} \cup \{*\}$. We model a single execution of a SCAM as an assignment to timed variables.

Definition 3.2.1 (Data stream)**.** A *data stream* is a map $\eta : \widehat{L} \to \mathcal{D}_*$.

Intuitively, $\eta(v,i) \in \mathcal{D}_*$ represents the data observed at location $v \in L$ and time step $i \geq 0$. If $\eta(v,i) = *$, no data is observed at location v and time step i. We define the k-th derivative $\eta^k : \widehat{L} \to \mathcal{D}_*$ of a data stream η as $\eta^k(v,i) = \eta(v, k+i)$, for all $v \in L$ and $i \geq 0$.

We can visualize a data stream η as an infinite table, with columns indexed by variables $v \in L$, rows indexed by non-negative integers $i \in \mathbb{N}_0$, and entries containing either $*$ or data from $\mathcal{D}$. For a time step $i \geq 0$, we can represent the i-th row of η as the partial map $\eta_i : L \rightharpoonup \mathcal{D}$, with $\mathrm{dom}(\eta_i) = \{v \in L \mid \eta(v,i) \neq *\}$ and $\eta_i(v) = \eta(v,i)$, for all $v \in \mathrm{dom}(\eta_i)$. We refer to η_i as the i-th *data assignment*. The empty function $\tau : L \rightharpoonup \mathcal{D}$, with $\mathrm{dom}(\tau) = \emptyset$, is also a valid data assignment. We use τ to represent an explicit silent step.

Definition 3.2.2 (Soft languages)**.** A soft language over a CLM $\langle A, \leq, \otimes, \mathbf{1}\rangle$ is a function $c : (\widehat{L} \to \mathcal{D}_*) \to A$.

Suppose that we have a morphism $h : A \to B$. Then, we can view any soft language c over A as a soft constraint over B. Indeed, the composition $h \circ c$ that maps a data stream η to the value $h(c\eta) \in B$ constitutes a soft constraint over B. In particular, if B is the Boolean CLM $\mathbb{B}$, we can view a soft constraint as a crisp constraint that defines a set of accepted executions. Hence, such a constraint corresponds naturally to a constraint automaton [BSAR06], which are thus subsumed by Definition 3.2.2.

Lemma 3.1.6 shows that soft constraints form a cylindric algebra. Thus, relevant notions, such as composition and hiding, carry over from soft constraints to SCAMs. It is straightforward to verify that all these notions correspond to their classical definitions in the literature.

3.2.2 Syntax

We fix a finite set of memory locations $\mathcal{X}$, a data domain $\mathcal{D}$, and a distributive and cancellative CLM $\mathbb{S}$. Recall the CLM of constraints $\mathbb{C}(V, D, \mathbb{S})$ from Lemma 3.1.5, for some set of variables V and domain of interpretation D.

Definition 3.2.3 (SCAM)**.** A *soft constraint automaton with memory* over $\mathcal{D}$ and $\mathbb{S}$ is a 6-tuple $\langle \mathcal{Q}, \mathcal{N}, \mathcal{X}, \longrightarrow, \mathcal{Q}_0, c_0 \rangle$, such that

1. $\mathcal{Q}$ is a finite set of states,
2. $\mathcal{N}$ is a finite set of port variables,
3. $\mathcal{X}$ is a finite set of memory locations,
4. $\longrightarrow \subseteq \mathcal{Q} \times 2^{\mathcal{N}} \times \mathbb{C}(\widehat{\mathcal{N} \cup \mathcal{X}}, \mathcal{D}_*, \mathbb{S}) \times \mathcal{Q}$ is a finite set of transitions,
5. $\mathcal{Q}_0 \subseteq \mathcal{Q}$ is a set of initial states, and
6. $c_0 \in \mathbb{C}(\widehat{\mathcal{N} \cup \mathcal{X}}, \mathcal{D}_*, \mathbb{S})$ is an initial constraint

such that $\mathcal{X} \cap \mathcal{N} = \emptyset$, $\mathrm{supp}(c_0) \subseteq \mathcal{X}$, and $(q, N, c, p) \in \longrightarrow$ implies that $\mathrm{supp}(c) \subseteq N \cup \mathcal{X} \cup \mathcal{X}'$ (where $\mathcal{X}' = \{x' \mid x \in \mathcal{X}\}$ is the set of first derivatives).

We usually write $q \xrightarrow{N,c} p$ instead of $(q, N, c, p) \in \longrightarrow$ and we call N the synchronization constraint and c the guard of the transition, respectively. We say that a transition is *invisible*, whenever $N = \emptyset$.

Different from [DGS18], the condition $\mathrm{supp}(c) \subseteq N \cup \mathcal{X} \cup \mathcal{X}'$ means that the guards are soft constraints with a single time step look-ahead for memory locations. This is just a simplifying assumption: the following results would carry over smoothly.

Definition 3.2.4 (Runs)**.** Let $\mathcal{T} = \langle \mathcal{Q}, \mathcal{X}, \mathcal{N}, \longrightarrow, \mathcal{Q}_0, c_0 \rangle$ be a SCAM. A *run* λ of $\mathcal{T}$ from $q \in \mathcal{Q}$ is an infinite sequence in $\longrightarrow^{\omega}$, with $\lambda_i = (p_i, N_i, c_i, q_i) \in \longrightarrow$, such that $p_0 = q$ and $q_i = p_{i+1}$, for all $i \geq 0$. We write $R(\mathcal{T}, q)$ for the set of runs of $\mathcal{T}$ from q and $R(\mathcal{T}) = \bigcup_{q \in \mathcal{Q}_0} R(\mathcal{T}, q)$ for the set of runs of $\mathcal{T}$.

The intuitive meaning of a SCAM $\mathcal{T}$ as an operational model for service queries is similar to the interpretation of labelled transition systems as models for reactive systems. The states represent the configurations of a service. The transitions represent the possible one-step behavior, where the meaning of a transition $p \xrightarrow{N,c} q$ is that we can move from configuration p to q, whenever

1. all ports in $n \in N$ perform an I/O operation,
2. all other ports in $\mathcal{N} \backslash N$ perform no I/O operation,
3. all ports/memory locations in $N \cup \mathcal{X} \cup \mathcal{X}'$ satisfy the guard c.

Each assignment to ports in N represents the data exchanged by the I/O operations through these ports, while assignments to variables in $\mathcal{X}$ and $\mathcal{X}'$ represent the data in memory locations before and after the transition.

For example, a transition $p \xrightarrow{\{a,b\},[x=a]\otimes[x'=b]} q$ from state p to q fires ports a and b, and the value at port a is equal to the current value of the memory x, and the

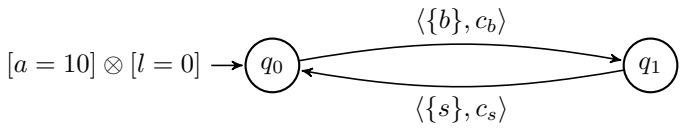


Figure 3.1: A SCAM over the data domain $\mathbb{N}_0$ and CLM $\langle \mathbb{Z} \cup \{\pm\infty\}, \leq, +, 0\rangle$, where $c_b = [-b] \otimes [a' = a - b] \otimes [b \leq a] \otimes [l' = b]$ buys an affordable item and saves its price, and $c_s = [s] \otimes [a' = a + s] \otimes [l \leq s]$ sells that item for a higher price.

next value at the memory x is equal to the current value at port b. Port variables, if not hidden, can be shared with other SCAM (cf., Definition 3.2.6), while memory variables are not shared (cf., Theorem 3.2.1).

Example 3.2.1 (A SCAM for buying and selling)**.** We describe an agent that prefers to buy an item as cheap as possible, and prefers to maximize its profit. We use the set $\mathbb{N}_0$ of natural numbers as a data domain, and we use the extended integers $\langle \mathbb{Z} \cup \{\pm\infty\}, \leq, +, 0\rangle$ as preference values. In particular, every datum can be viewed as a preference value.

Figure 3.1 shows a (deterministic) SCAM for buying and selling. The set of ports, $\mathcal{N}$, is $\{b, s\}$, where the value at b is the purchase price of an item, and the value at s is the selling price of an item. The set of variables, $\mathcal{X}$, is $\{a, l\}$, where a is the current balance, and l is the price of current item. The soft constraint c_b buys $(a' = a - b)$ an affordable $(b \leq a)$ item, and stores its value $(l' = b)$. The preference of $-b$ ensures that maximizing preference amounts to minimizing purchase price. The soft constraint c_s sells the current item $(a' = a + s)$ for a higher price $(l \leq s)$. The preference of s ensures that maximizing preference amounts to maximizing selling price. ◊

3.2.3 Semantics

Recall the lexicographically-ordered CLM of streams $\mathbb{S}^\omega$ from Theorem 3.1.4. We interpret a SCAM $\mathcal{T}$ as a soft language

$$L(\mathcal{T}) : (\widehat{\mathcal{N} \cup \mathcal{X}} \to \mathcal{D}_*) \to \mathbb{S}^\omega$$

that assigns a preference stream from $\mathbb{S}^\omega$ to every possible execution. The constraint $L(\mathcal{T})$ can be seen as the language of the SCAM $\mathcal{T}$.

We describe the intuitive semantics of a SCAM. Let $\eta : \widehat{\mathcal{N} \cup \mathcal{X}} \to \mathcal{D}_*$ be a data stream. First, we define the preference stream $c_\lambda\eta \in \mathbb{S}^\omega$ of η with respect to a run $\lambda = t_0 t_1 t_2 \cdots \in R(\mathcal{T}, q)$ from a state $q \in \mathcal{Q}$. We compute the initial preference $c_0\eta \in \mathbb{S}$ and, for every transition $t_i = (p_i, N_i, c_i, q_i)$ in the run λ, we compute the preference $c_{t_i}\eta^i \in \mathbb{S}$ of transition t_i, where η^i is the ith derivative of η, and c_{t_i} is the soft constraint composed from the guard c_i and the synchronization constraint N_i. For $i \geq 0$, consider the composition $a_i = c_0\eta \otimes c_{t_i}\eta^i \in \mathbb{S}$. If the initial condition and all transition guards and synchronization constraints are satisfied (i.e., a_i cancellative, for all $i \geq 0$), then the preference stream of η equals $c_\lambda\eta = a_0 a_1 a_2 \cdots \in \mathbb{S}^\omega$. Otherwise, we set $c_\lambda\eta = \bot^\omega$.

Next, we define the preference value of the data stream η assigned by the SCAM $\mathcal{T}$ as the least upper bound (in the lexicographically-ordered CLM of streams $\mathbb{S}^\omega$) over all possible runs that start from an initial state. The lexicographic order implies that, at any given state, the SCAM prefers to take the outgoing transition of maximal preference. Indeed, any run that starts with an outgoing transition of suboptimal preference results in a preference stream that is suboptimal in the lexicographic order.

Finally, we hide all memory locations, which prevents SCAMs from synchronizing on shared memory locations (Theorem 3.2.1).

Definition 3.2.5 (SCAM semantics)**.** Let $\mathcal{T} = \langle \mathcal{Q}, \mathcal{N}, \mathcal{X}, \longrightarrow, \mathcal{Q}_0, c_0 \rangle$ be a SCAM. The semantics of a transition $t = (p, N, c, q) \in \longrightarrow$ is a soft constraint $c_t \in \mathbb{C}(\widehat{\mathcal{N} \cup \mathcal{X}}, \mathcal{D}_*, \mathbb{S})$ defined as

$$c_t = c \otimes \bigotimes_{n \in N} [n \neq *] \otimes \bigotimes_{n \in \mathcal{N} \setminus N} [n = *].$$

The semantics of a run $\lambda = t_0 t_1 t_2 \cdots \in R(\mathcal{T}, q)$ from a state $q \in \mathcal{Q}$ is a soft constraint c_λ that maps a data stream $\eta : \widehat{\mathcal{N} \cup \mathcal{X}} \to \mathcal{D}_*$ to the preference stream $c_\lambda \eta$ defined as

$$c_\lambda \eta = \begin{cases} a_0 a_1 a_2 \cdots & \text{if } a_i = c_0 \eta \otimes c_{t_i} \eta^i \text{ is cancellative, for all } i \geq 0, \\ \bot^\omega & \text{otherwise} \end{cases}$$

The accepted language of $\mathcal{T}$ at $q \in \mathcal{Q}$ is defined as

$$L(\mathcal{T}, q) = \bigvee \{ c_\lambda \mid \lambda \in R(\mathcal{T}, q) \}.$$

The language of $\mathcal{T}$ is defined as

$$L(\mathcal{T}) = \exists_{\widehat{\mathcal{X}}} \bigvee \{ L(\mathcal{T}, q) \mid q \in \mathcal{Q}_0 \}.$$

Definition 3.2.5 deals exclusively with infinite paths in a SCAM $\mathcal{T}$: if a state q has no outgoing transitions, then $c(\mathcal{T}, q)\eta = \bot$, for every data stream η.

Example 3.2.2 (The language of business)**.** Let $\mathcal{T}$ be the SCAM from Example 3.2.1. Consider a data stream $\eta : \{\widehat{b, s, a, l}\} \to \mathbb{N}_0$ whose prefix is defined in Figure 3.2. From $\eta(a, 0) = 10$ and $\eta(l, 0) = 0$, it follows that

$$c_0 \eta = ([a = 10] \otimes [l = 0])\eta = \mathbf{1},$$

which means that the initial condition is satisfied. There exists only one possible run $\lambda = t_0 t_1 \cdots$ in $\mathcal{T}$ from the initial state q_0. Hence, the stream of preferences associated with η satisfies

$$L(\mathcal{T})\eta = (\exists_{\widehat{\{a,l\}}} \bigvee \{c_\lambda\})\eta = (\exists_{\widehat{\{a,l\}}} c_\lambda)\eta = c_\lambda \eta,$$

where the last equality follows from the fact that the preferences are independent of the memory locations a and l. Concretely, the stream of preferences $L(\mathcal{T})\eta = a_0 a_1 \cdots$ satisfies

$$a_0 = c_0 \eta \otimes c_{t_0} \eta^0 = \mathbf{1} \otimes c_b \eta^0 = -\eta^0(b, 0) = -\eta(b, 0) = -6$$
$$a_1 = c_0 \eta \otimes c_{t_1} \eta^1 = \mathbf{1} \otimes c_s \eta^1 = \eta^1(s, 0) = \eta(s, 1) = 7$$

x	b	s	a	l
$\eta(x,0)$	6	$*$	10	0
$\eta(x,1)$	$*$	7	4	6
$\eta(x,2)$	$\vdots$	$\vdots$	11	$\vdots$

Figure 3.2: Data stream for the SCAM from Example 3.2.1, wherein the agent starts with 10 units of money, buys an item for 6 units, and sells it for 7 units.

The lexicographic order on preference streams ensures that any other data stream $\rho : \{\widehat{b,s,a,l}\} \to \mathbb{N}_0$, for which $\rho(b,0) < 6$, satisfies $L(\mathcal{T})\eta <_l L(\mathcal{T})\rho$, which means that the data stream ρ is preferred over η. In other words, the SCAM $\mathcal{T}$ prefers to minimize the purchase price. ◊

3.2.4 SCAM composition

We now introduce the product of automata, extending [AS12, Definition 5].

Definition 3.2.6 (Soft join)**.** Let $\mathcal{T}_i = (\mathcal{Q}_i, \mathcal{X}_i, \mathcal{N}_i, \to_i, \mathcal{Q}_{0i}, c_{0i})$ for $i \in \{0,1\}$ be two SCAMs over $\mathcal{D}$ and $\mathbb{S}$, with $(\mathcal{N}_0 \cup \mathcal{N}_1) \cap (\mathcal{X}_0 \cup \mathcal{X}_1) = \emptyset$. Then, their soft product $\mathcal{T}_0 \bowtie \mathcal{T}_1$ is the tuple $\langle \mathcal{Q}_0 \times \mathcal{Q}_1, \mathcal{X}_0 \cup \mathcal{X}_1, \mathcal{N}_0 \cup \mathcal{N}_1, \longrightarrow, \mathcal{Q}_{00} \times \mathcal{Q}_{01}, c_{00} \otimes c_{01} \rangle$ where $\longrightarrow$ is the smallest relation that satisfies the rule

$$\frac{q_0 \xrightarrow{N_0,c_0}_0 p_0, \quad q_1 \xrightarrow{N_1,c_1}_1 p_1, \quad N_0 \cap \mathcal{N}_1 = N_1 \cap \mathcal{N}_0}{\langle q_0, q_1 \rangle \xrightarrow{N_0 \cup N_1, c_0 \otimes c_1} \langle p_0, p_1 \rangle}$$

The rule applies when there is a transition in each automaton such that they can fire together. This happens only if the two local transitions agree on the subset of shared ports that *fire* (which is empty, if no ports are shared). The transition in the resulting automaton is labelled with the union of the name sets on both transitions, and the constraint is the conjunction of the constraints of the two transitions.

Note that the new automaton may include asynchronous executions: it suffices that the SCAM is *reflexive*, i.e., every q has an idling transition $q \xrightarrow{\emptyset,\mathbf{1}} q$. To avoid such idling transitions to be of maximal preference, we must use a bipolar CLM of preferences, wherein $\top > \mathbf{1}$.

We now express the composition of SCAM in Definition 3.2.6 in terms of composition of languages as defined in Lemma 3.1.5.

Theorem 3.2.1 (Correctness of soft join)**.** *Let $\mathcal{T}_0$ and $\mathcal{T}_1$ be two SCAMs sharing no memory location. Then, $L(\mathcal{T}_0 \bowtie \mathcal{T}_1) = L(\mathcal{T}_0) \otimes L(\mathcal{T}_1)$.*

Proof. We first show that, for all $(q_0, q_1) \in \mathcal{Q}_0 \times \mathcal{Q}_1$, we have

$$L(\mathcal{T}_0 \bowtie \mathcal{T}_1, (q_0,q_1)) = \bigvee \{ c_{\rho_0} \otimes c_{\rho_1} \mid \rho_i \in R(\mathcal{T}_i, q_i), i \in \{0,1\} \}. \tag{3.1}$$

Let $\rho \in R(\mathcal{T}_0 \bowtie \mathcal{T}_1, (q_0,q_1))$ be a run from (q_0,q_1). By construction of $\mathcal{T}_0 \bowtie \mathcal{T}_1$, we find runs $\rho_i \in R(\mathcal{T}_i, q_i)$, for $i \in \{0,1\}$, such that $c_\rho = c_{\rho_0} \otimes c_{\rho_1}$. Hence, c_ρ is less

than or equal to the right-hand side of Equation (3.1). Since, ρ is arbitrary, we conclude the $\leq$ part of Equation (3.1).

For $i \in \{0,1\}$, let $\rho_i \in R(\mathcal{T}_i, q_i)$ be, a run from q_i. If ρ_0 and ρ_1 are not compatible according to the rule in Definition 3.2.6, we have $c_{\rho_0} \otimes c_{\rho_1} = [\bot^\omega] \leq L(\mathcal{T}_0 \bowtie \mathcal{T}_1, (q_0, q_1))$. If ρ_0 and ρ_1 are compatible according to the rule in Definition 3.2.6, we find a run $\rho \in R(\mathcal{T}_0 \bowtie \mathcal{T}_1, (q_0, q_1))$ from (q_0, q_1), such that $c_{\rho_0} \otimes c_{\rho_1} = c_\rho \leq L(\mathcal{T}_0 \bowtie \mathcal{T}_1, (q_0, q_1))$. Since ρ_0 and ρ_1 are abritrary, we conclude the $\geq$ part of Equation (3.1), which proves Equation (3.1).

Using Equation (3.1) and Theorem 3.1.4, we find, for $(q_0, q_1) \in \mathcal{Q}_0 \times \mathcal{Q}_1$, that

$$\begin{aligned} L(\mathcal{T}_0 \bowtie \mathcal{T}_1, (q_0, q_1)) &= \bigotimes_{i=0}^{1} \bigvee \{c_{\rho_i} \mid \rho_i \in R(\mathcal{T}_i, q_i)\} \\ &= L(\mathcal{T}_0, q_0) \otimes L(\mathcal{T}_1, q_1) \end{aligned}$$

Since no memory is shared, we have $\mathcal{X}_0 \cap \mathcal{X}_1 = \emptyset$. Then,

$$\begin{aligned} L(\mathcal{T}_0 \bowtie \mathcal{T}_1) &= \exists_{\widehat{\mathcal{X}}} \bigvee \{L(\mathcal{T}_0, q_0) \otimes L(\mathcal{T}_1, q_1) \mid q_0 \in \mathcal{Q}_{00}, q_1 \in \mathcal{Q}_{01}\} \\ &= \exists_{\widehat{\mathcal{X}_0}} \exists_{\widehat{\mathcal{X}_1}} \bigotimes_{i=0}^{1} \bigvee \{L(\mathcal{T}_i, q_i) \mid q_i \in \mathcal{Q}_{0i}\} \\ &= \bigotimes_{i=0}^{1} \exists_{\widehat{\mathcal{X}_i}} \bigvee \{L(\mathcal{T}_i, q_i) \mid q_i \in \mathcal{Q}_{0i}\} = L(\mathcal{T}_0) \otimes L(\mathcal{T}_1), \end{aligned}$$

which proves the result. □

3.2.5 SCAM hiding

The hiding operator [BSAR06] abstracts the details of the internal communication in a constraint automaton. For SCA [AS12, Definition 6], the hiding operator $\exists_O \mathcal{T}$ removes from the transitions all the information about the ports in $O \subseteq \mathcal{N}$, including those in the(support of the) constraints. The definition smoothly extends over SCAMs: in fact, since we allow silent transitions, our definition is much more compact.

Definition 3.2.7 (Soft hiding)**.** Let $\mathcal{T} = \langle \mathcal{Q}, \mathcal{X}, \mathcal{N}, \longrightarrow, \mathcal{Q}_0 \rangle$ be a SCAM and $O \subseteq \mathcal{N}$ a set of ports. Then, $\exists_O \mathcal{T}$ is the SCAM $\langle \mathcal{Q}, \mathcal{X}, \mathcal{N} \setminus O \longrightarrow_*, \mathcal{Q}_0 \rangle$ where $\longrightarrow_*$ is defined by $q \xrightarrow{N \setminus O, \exists_O c}_* p$ iff $q \xrightarrow{N,c} p$.

We express the correctness of hiding in terms of the cylindric operator on soft constraints from Lemma 3.1.6.

Theorem 3.2.2 (Correctness of soft hiding)**.** *Let $\mathcal{T}$ be a SCAM and O a set of its ports. Then, $L(\exists_O \mathcal{T}) = \exists_O L(\mathcal{T})$.*

Proof. We prove that, for all $q \in \mathcal{Q}$, we have

$$L(\exists_O \mathcal{T}, q) = \bigvee \{\exists_O c(\rho) \mid \rho \in R(\mathcal{T}, q)\}. \tag{3.2}$$

By construction of $\exists_O \mathcal{T}$ in Definition 3.2.7, we have a natural 1-1 correspondence between runs $\rho \in R(\exists_O \mathcal{T}, q)$ in $\exists_O \mathcal{T}$ from q and runs $\rho' \in R(\mathcal{T}, q)$ in $\mathcal{T}$ from q, which satisfies $c_\rho = \exists_O c_{\rho'}$. Using the same approach used in the proof of Theorem 3.2.1 (i.e., $\leq$ and $\geq$ on LUBs), we conclude that Equation (3.2) holds. For all $q \in \mathcal{Q}$, we now find that $L(\exists_O \mathcal{T}, q) = \exists_O L(\rho, q)$. Hence, $L(\exists_O \mathcal{T}) = \exists_O L(\mathcal{T})$, which proves the result. □

3.3 Case study

We present an example that illustrates the operations of composition and hiding for SCAMs. The example consists of an interrupt management-system tied to a data-flow of information. Even if academic, it is rooted into concepts widely adopted by several real-world examples, e.g., a computer CPU receiving hardware and software interrupts.

We show that, even if the machine has the ability to keep executing a process, in the presence of a *kill* signal sent by the operator, the machine chooses to stop. The given construction could be adapted to express the case where more than one machine is controlled by an operator.

As a carrier for the preferences of the soft constraints, we use the CLM of extended integers $\mathbb{S} = \langle \mathbb{Z} \cup \{\pm\infty\}, \leq, +, 0 \rangle$ from Example 3.1.4. Recall that a tautology has preference $\mathbf{1}$, which is the element $0 \in \mathbb{S}$, and a false constraint has the least preference $-\infty \in \mathbb{S}$. We refer to $+\infty$ as the element $\top$, and $-\infty$ as the element $\bot$.

3.3.1 Operator

Let $\mathcal{A} = \langle \{q_0, q_1\}, \{k, s, i, ack\}, \{c, id\}, \longrightarrow, \{q_0\}, [id = *] \otimes [c = 0] \rangle$ be the SCAM representation of the operator, with transition relation $\longrightarrow$ as defined in Figure 3.3. Initially, the memory id is empty. The operator, in state q_0, waits to receive a signal i and stores the value carried by the signal in the memory location id. Then, the operator waits for a signal $s \neq *$ to arrive, and takes the outgoing transition from q_0 to q_1 only if the value of s equals the current value of memory location id. Simultaneously, the operator starts a counter by setting the memory location c to 10. Being in state q_1, the operator repeatedly decreases its counter c. When the value of memory location c becomes negative, the operator sends a kill signal carrying the value stored in memory location id. If the operator receives, in state q_1, an acknowledge signal ack with the value stored in the memory location id, the operator sets the counter memory location c to 0, and returns to its initial state q_0.

The preference values in Figure 3.3 ensure that, if the guards of the self transition at q_1 and the transitions from q_1 to q_0 are satisfied by the same assignment, the operator prefers to send a kill or acknowledge signal (transitions from q_1 to q_0) instead of decreasing its counter (self transition at q_1).

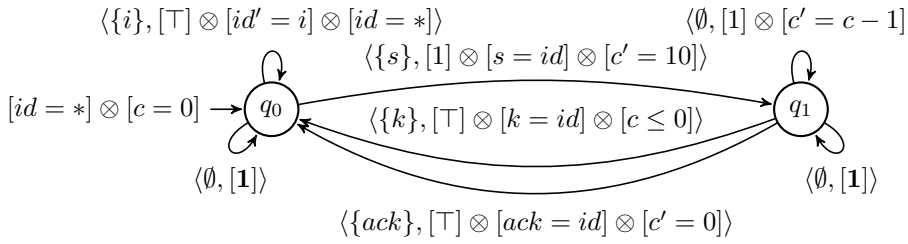


Figure 3.3: The operator's SCAM $\mathcal{A}$ over an arbitrary data domain and the CLM $\langle \mathbb{Z} \cup \{\pm\infty\}, \leq, +, 0\rangle$.

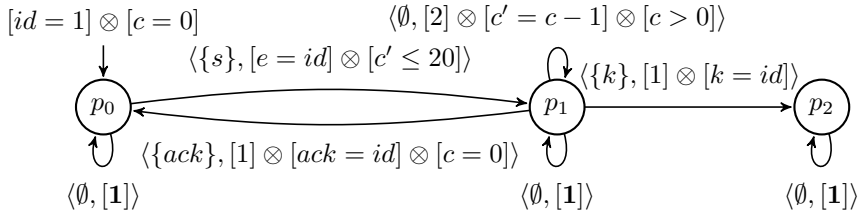


Figure 3.4: The machine's SCAM $\mathcal{B}$ over an arbitrary data domain and the CLM $\langle \mathbb{Z} \cup \{\pm\infty\}, \leq, \otimes, \mathbf{1}\rangle$.

3.3.2 Machine

Let $\mathcal{B} = \langle\{p_0, p_1, p_2\}, \{k, s, ack\}, \{c, id\}, \longrightarrow, \{p_0\}, [id = 1] \otimes [c = 0]\rangle$ be the SCAM representation of a machine, whose transition relation $\longrightarrow$ is shown in Figure 3.4. The machine starts in p_0, with the identity value 1 stored in the memory id. Whenever the value observed on port s corresponds to its identity id, the machine can start executing and moves from state p_0 to p_1. In state p_1, the execution of the machine is simulated by decreasing a counter from a non-deterministically selected initial value of at most 20. Once the counter reaches 0, the machine sends an acknowledgement with its own id value, and gets back to state p_0. At any point, however, the machine can be interrupted by a kill signal and goes to state p_2.

The constraints of the machine ensure that the machine terminates, if a kill signal is received. In absence of a kill signal, the machine prefers to execute the process before sending the acknowledgement.

3.3.3 Composition

Figure 3.5 shows the SCAM of the composition $\mathcal{A} \bowtie \mathcal{B}$ of the operator, $\mathcal{A}$, and the machine, $\mathcal{B}$. Their composition synchronizes on the shared ports k, ack, and s between $\mathcal{A}$ and $\mathcal{B}$. While the I/O direction of a port is not explicitly mentioned, one should think of the port k as an input port for the machine and output port for the operator. Similarly, the port ack is used as an output port for the machine and input port for the operator.

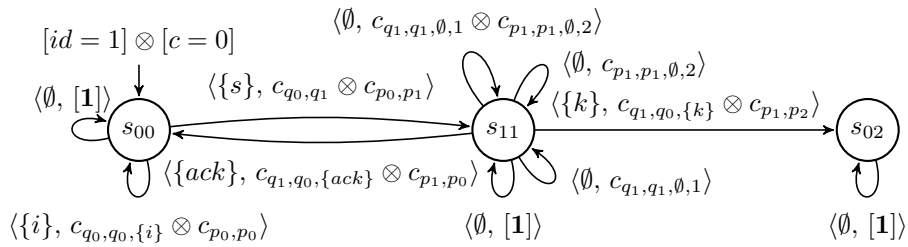


Figure 3.5: The product $\mathcal{A} \bowtie \mathcal{B}$ of the machine's SCAM and operator's SCAM, where $s_{ij} = (q_i, p_j)$ and $c_{x,y,N,e}$ is the constraint of the underlying SCAM labelling transition from state x to state y with synchronization set N and constant preference e. If clear from the context, some elements from x, y, N, e are omitted to identify the constraint.

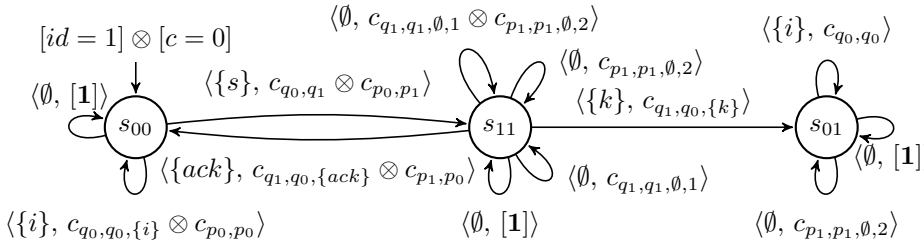


Figure 3.6: The product $\mathcal{A} \bowtie \exists_k(\mathcal{B})$ of the machine's SCAM and operator's SCAM after hiding k in the machine, using the same notation for states and guards as in Figure 3.5.

If satisfied, the soft constraint $c_{q_1,q_1,\emptyset,1} \otimes c_{p_1,p_1,\emptyset,2}$[3] evaluates to the preference $2 + 1 = 3$, and the soft constraint $c_{q_1,q_0,\{k\}} \otimes c_{p_1,p_2}$ evaluates to the preference $\top$. Since $3 < \top$, when the counter memory of the operator reaches 0, the run where the kill signal is sent has higher preference than the run where the machine keeps executing its process.

3.3.4 Hiding

We compare the product $\mathcal{A} \bowtie \mathcal{B}$ with the product $\mathcal{A} \bowtie \exists_k(\mathcal{B})$, wherein we first hide the port k in $\mathcal{B}$. As displayed in Figure 3.6, the composite system goes to the state s_{01}, where the transition $\langle \emptyset,\ c_{p_1,p_1,\emptyset,2} \rangle$ can still be taken (i.e. the machine is still running). Hence, hiding the kill signal in $\mathcal{B}$ does not force the machine to terminate its execution. Note that state s_{01} is a deadlock, because the operator cannot receive an acknowledge signal or send a kill signal.

[3] See the label of Figure 3.5 for clarification on the notation.

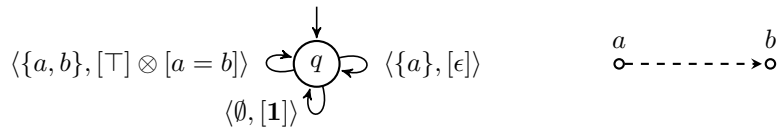


Figure 3.7: SCAM representation of the context-sensitive lossysync channel, and its Reo notation. The passing transition, $\langle \{a, b\}, [\top] \otimes [a = b] \rangle$, has priority over the losing transition, $\langle \{a\}, [\epsilon] \rangle$, and the losing transition has priority over the idling transition, $\langle \emptyset, [\mathbf{1}] \rangle$.

3.4 Application to context-sensitivity

We apply our SCAM framework to model context-sensitivity, which is also known as context-dependency or as context-awareness.

Definition 3.4.1. A component is context-sensitive iff an I/O request by the environment can disable an action of the component.

One source of context-sensitivity is priority. If an I/O request by the environment enables a high-priority action, then previously enabled actions of lower priority become disabled. Although other sources of context-sensitivity must exist in theory, we do not know of any convincing example.

The notion of context-sensitivity received considerable attention in the Reo community. The primal example of a context-sensitive Reo connector is a lossysync channel, which accepts a datum d from its input end, and either atomically offers d at its output end, or loses d if the output is not ready to accept. The literature offers a variety of semantic models that encode context-sensitive behavior, namely *coloring semantics* [CCA07], *augmented Büchi automata of records* [IBC08], *intentional automata* [CNR11], and *guarded automata* [BCS12]. Context-sensitivity can be encoded in context-insensitive models by adding dual ports [JKA11]. Although we consider context-sensitivity in the realm of Reo, we stress that context-sensitivity is a fundamental concept that applies to languages other than Reo.

The environment of a connector can be represented in at least two ways. On the one hand, augmented Büchi automata of records, intentional automata, and guarded automata represent the environment as the subset of ports of the connector that have pending requests. On the other hand, coloring semantics and the encoding in [JKA11] represent the environment as another connector of identical type that composes with current connector.

All existing context-sensitive models for Reo [CCA07, IBC08, CNR11, BCS12, JKA11] have special syntax to detect the presence or absence of pending I/O requests. Intentional automata, guarded automata, and augmented Büchi automata of records query the presence of I/O request via a Boolean guard. The coloring semantics uses two colors for the absence of data flow, which allows the connector to detect the presence of I/O requests. The dual ports in the encoding in [JKA11] serve the same purpose as the two colors in the coloring semantics.

We now propose a context-sensitive semantics without any syntax to detect the presence or absence of pending I/O requests. As such, our approach is arguably

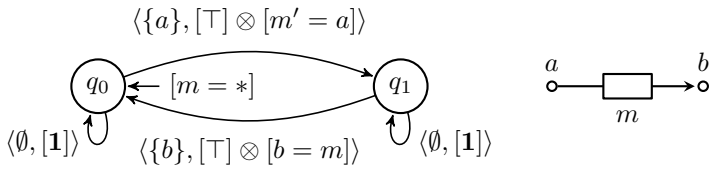


Figure 3.8: SCAM representation of the fifo channel, and its Reo notation.

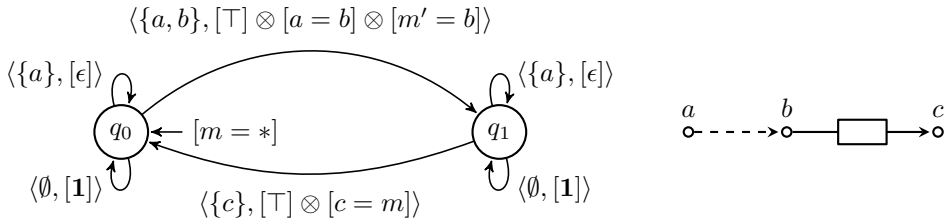


Figure 3.9: Composition of the lossysync and fifo in Figures 3.7 and 3.8. If the fifo channel can drain its buffer, then the lossysync channel cannot lose any datum.

simpler than existing approaches. The basic idea is to distinguish four types of transitions, namely

1. *illegal* transitions with unsatisfiable soft constraint.
2. *idling* transition (i.e., a silent self-loop transition).
3. *losing* transition (as in the lossysync).
4. *regular* transitions (i.e., legal, non-idling, non-losing transitions).

We assign to each transition type a unique preference value from the CLM $\mathcal{E} = \mathbb{K} \times \mathbb{B}$, where $\bot = (\bot, \bot)$, $\mathbf{1} = (\mathbf{1}, \top)$, $\epsilon = (\top, \bot)$, and $\top = (\top, \top)$ correspond respectively with illegal, idling, losing, and regular transitions.

The partial order $\leq$ on $\mathcal{E}$ induces a priority relation on the set of enabled transitions in a SCAM over $\mathcal{E}$. If present, the connector fires any enabled transition of highest priority. The multiplication, $\otimes$, is used to propagate the types through composition.

Figure 3.7 shows the SCAM representation of the lossysync channel. We verify that the SCAM representation of lossysync behaves as desired, if it operates in isolation. Note that $\bot < \mathbf{1} < \top$ and $\bot < \epsilon < \top$, but ϵ and $\mathbf{1}$ are incomparable. If the lossysync has no pending I/O operations on a or b, then the idling transition, $\langle \emptyset, [\mathbf{1}] \rangle$, is the only enabled transition. If there is a pending input at port a, then the losing transition, $\langle \{a\}, [\epsilon] \rangle$, and the idling transition, $\langle \emptyset, [\mathbf{1}] \rangle$, are enabled. Since $\mathbf{1}$ and ϵ are incomparable, the choice between losing and idling is non-deterministic. If there are pending I/O operations on both a and b, then all transitions are enabled. In particular, the passing transition, $\langle \{a, b\}, [\top] \otimes [a = b] \rangle$, has priority over all other transitions.

We now verify that the SCAM representation of lossysync behaves as desired, if it operates in a composition. Our approach crucially relies on the correct identification of the three different transition types, namely the illegal, idling, and losing transitions. We define the type of a global transition $\tau = \tau_1 \mid \cdots \mid \tau_n$ as follows:

1. τ is illegal iff one local transitions τ_i is illegal,
2. τ is idling iff all local transitions τ_i are idling,
3. τ is losing iff τ is not illegal and one transitions τ_i is losing.
4. τ is regular iff τ is not idling and all τ_i are idling or regular.

The following result ensures that the transition types are correctly propagated through the composition of SCAMs.

Lemma 3.4.1. *Consider the CLM $\mathcal{E}$, and let $x = a_1 \otimes \cdots \otimes a_n$, for some $n \geq 2$, and $a_1, \ldots, a_n \in \mathcal{E}$. Then, for $I = \{1, \ldots, n\}$, we have*

1. *$x = \bot$ if and only if $\exists i \in I.\ a_i = \bot$.*
2. *$x = \mathbf{1}$ if and only if $\forall i \in I.\ a_i = \mathbf{1}$.*
3. *$x = \epsilon$ if and only if $(\exists i \in I.\ a_i = \epsilon) \wedge (\forall i \in I.\ a_i \neq \bot)$.*
4. *$x = \top$ if and only if $(\exists i \in I.\ a_i = \top) \wedge (\forall i \in I.\ a_i \in \{\mathbf{1}, \top\})$.*

Proof. Follows immediately from the definition. □

Example 3.4.1. Consider the composition C of the lossysync channel and the fifo channel, as depicted in Figure 3.9. Suppose that C is in state q_1, which means that the fifo channel is full. If there is a pending I/O request on port c, then the data can be taken out of the can drain its buffer, then the lossysync channel cannot lose any datum. ◊

It is important to observe that the bipolar approach is essential for the construction of our context-sensitive model. To see this, note that $\mathbf{1}$ is the only sensible preference value for an idling transition. Otherwise, composition with an idling transition would change the preferences. If $\mathbf{1} = \top$ holds (as is the case for c-semirings), then the priority of the losing transition is necessarily lower than the priority of the idling transition. Consequently, any component would prefer idling (which is always possible) over losing, which is clearly undesirable.

3.5 Related work on constraint automata

The closest related work to what is discussed in this chapter concerns other extensions of constraint automata (CAs), previously advanced in the ample literature about Reo.

Quantitative CAs (QCAs) are introduced in [ACMM07, MA09] with the aim of describing the behavior of connectors tied to their *quality of service* (QoS), e.g., a reliability measure or the shortest transmission time. Similarly to CAs, the states

of a QCAs correspond to the internal states of the connector it models. The label on a transition consists instead of a firing set, a data constraint, and a cost that represents a QoS metric. Hence, QCAs differ from *timed* [ABdBR07] and *probabilistic* [Bai05] constraint automata, because these latter two classes of models describe functional aspects of connectors, while QoS represents non-functional properties.

As applications, SCAs have been already used in [AS12, SSAA13] and [KAT16, KAT17, TNAK16]. Different from previous related work, the main motivation behind SCAs is to associate an action with a preference. In [AS12, SSAA13] the authors present a formal framework that is able to discover stateful web services, and to rank the results according to a similarity score expressing the affinities between the query, asked by a user, and the services in a database. Preference for the similarity between the query and each service is modeled through SCAs. In the second group of works instead, the authors advance a framework that facilitates the construction of autonomous agents in a compositional fashion; these agents are 'soft', in that their actions are associated with a preference value, and agents may or may not execute an action depending on a threshold preference. Hence, at design-time, SCAs can be used to reason about the behavior of the components in an uncertain physical world, i.e., to model and verify the behavior of cyber-physical systems.

Research on SCAs is currently a trending topic among all the different lines concerning Reo. An example is [Tal18], where the authors describe two complementary approaches to the specification and analysis of robust cyber-physical agent systems: the first one focuses on abstract theoretical concepts based on automata and temporal logics, called *soft component automata*; the second approach describes a concrete experimental approach based on executable rewriting logic specifications, simulation, search, and model checking, called *soft agents* [Tal18]. The soft agents framework combines ideas from several previous works: *i)* the use of soft constraints and SCAs for specifying robust and adaptive behavior, *ii)* partially ordered knowledge sharing for communication in disrupted environments, *iii)* and the real-time Maude approach to modelling timed systems.

The work in [JKA17] extensively presents a kind of CA (there named as `W/MC`) consisting of a finite set of states, a finite set of transitions, three sets of directed ports, and a set of memory cells. The presence of memory cells in `W/MC` allows one to explicitly model the content of buffers, instead of using states. The main difference is that constraints are not soft in [JKA17], and consequently they do not allow for representing preference values, as needed by application summarized in the following paragraph.

Along the same line concerning cyber-physical systems, the related literature is represented by several works, as for example is [TNAK16] and [BMMS16]. In [TNAK16] the authors formalize soft agents in the *Maude* rewriting logic system [CDE$^+$02]. The most important features of this framework are the explicit representation of the physical state, the cyber perception of this state, the robust communication via sharing of partially ordered knowledge, and the robust behavior based on soft constraints. In [BMMS16] the authors address the problem of finding what local properties the agents in a cyber-physical system have to satisfy to guarantee a global required property ϕ; preferences are modeled via semirings on actions, and verified through a model checking function. Note that also all

the examples in [KAT16] use SCAs (with preferences) to model the behavior of cyber-physical systems.

The feature of enhancing automata with memory has roots in the dawn of computer science. In this way, an automaton can base its transition on both the current symbol being read and values stored in memory; moreover, it can issue commands to the memory device whenever it makes a transition. For example, *pushdown automata* (*PDAs*) employ a stack through which operations can be determined by the first element on such a data structure; a transition rule optionally pops the top of the stack, and optionally pushes new symbols onto the stack. *Stack automata* allow access to and operations on deeper elements instead, and can recognize a strictly larger set of languages than PDAs [HU67]. Applications may concern also computational models in biology: e.g., automata can use memory to stabilize the behavior of modeled proteins [AA16].

A conclusive related work is represented by [MSKA14], where Reo channels are annotated with stochastic values for data arrival rates at channel ends and processing delay rates at channels. Automata are thus stochastically extended in order to compositionally derive a QoS-aware semantics for Reo. The semantics is given by translating a component into *continuous-time Markov chains*. Our approach deals with preferences by using a more general approach: we do not only consider time but different systems of preferences (even bipolar ones), as long as they can be cast in the algebraic structure we present in Section 3.2.

3.6 Discussion

We have reworked soft constraint automata as originally proposed in [AS12, KAT16], with the dual purpose of first, extending the underlying algebraic structure in order to model both positive and negative preferences, and second, adding memory locations as originally provided for 'standard' constraint automata [JKA17].

As future work, we have many directions in mind. First, we would like to extend existing Reo compilers [Jon16, DA18a] to a SCAM-based compiler. Our results allow the user to conveniently compile context-sensitive connectors.

Next, we would like to exploit the properties of soft constraints to give additional operators on SCAMs, such as operators for port *renaming* or for *determinizing* guards by adding $[m' = m]$, whenever m' is unbound.

Finally, we would like to encode the behavior of SCAMs into a *concurrent constraint programming* language [GSPV17]. Such languages provide agents with actions to *tell* (i.e., add) and *ask* (i.e., query) constraints to a centralized store of information; this store represents a *constraint satisfaction problem*, and standard heuristic-based techniques might be applied to find a solution to complex conditions on *filter channels* [Arb11].

Part II

Compilation

Chapter 4

Protocol Syntax

The protocol specifications introduced in Part I offer explicit information on the interactions of tasks in a given application, and we can use this information for scheduling. However, scheduling operates on concrete tasks, rather than their formal specifications. Therefore, we now study compilation of these protocol specification, which allows us to develop scheduling techniques for the generated executable.

History shows that the number of compilers for Reo is proportional to the number of PhD students that worked on Reo, and each tool uses its own input language. For example, the Dreams framework [PCdVA12] uses a graphical editor in Eclipse, The Lykos compiler [Jon16] uses FOCAML, and Vereofy uses the RSL [Klü12]. The current chapter aims to unify these toolsets by proposing a syntax for Reo protocols called Treo[1]. A key feature of the Treo language is that it allows tool developers to extend Treo with their own semantics of primitive components, which is an essential feature in view of plethora of semantics models for Reo [JA12].

Many tools for Reo have been implemented as a collection of Eclipse plugins called the ECT [ECT]. The main plugin in this tool set consists of a graphical editor that allows a user to draw a connector on a canvas. The graphical editor has an intuitive interface with a flat learning curve. However, it does not provide constructs to express parameter passing, iteration, recursion, or conditional construction of connector graphs. Such language constructs are more easily offered by familiar programming language constructs in a textual representation of connectors.

In the context of Vereofy (a model checker for Reo), Baier, Blechmann, Klein, and Klüppelholz developed the Reo Scripting Language (RSL) and its companion language, the Constraint Automata Reactive Module Language (CARML) [BBKK09, Klü12]. RSL is the first textual language for Reo that includes a construct for iteration, and a limited form of parameter passing. Primitive channels and nodes are defined in CARML, a guarded command language for specification of constraint automata. Programmers then combine CARML specified constraint automata as primitives in RSL to construct complex connectors and/or complete systems. In contrast to the declarative nature of the graphical syntax of Reo, RSL is imperative.

[1]The work in this chapter is based on [DA18b]

Jongmans developed the First-Order Constraint Automata with Memory Language (FOCAML) [Jon16], a textual declarative language that enables compositional construction of connectors from a (pre-defined set of) primitive components. As a textual representation for Reo, however, FOCAML has poor support for its primary design principle: Reo channels are user-defined, not tied to any specific formalism to express its semantics, and compose via shared nodes with predefined merge-replicate behavior. Although FOCAML components are user-defined, FOCAML requires them to be of the same predefined semantic sort (i.e., constraint automata with memory [BSAR06]). The primary concept of Reo nodes does not exist in FOCAML, which forces explicit construction of their 'merge-replicate' behavior in FOCAML specifications.

Jongmans et al. have shown by benchmarks that compiling Reo specifications can produce executable code whose performance competes with or even beats that of hand-crafted programs written in languages such as C or Java using conventional concurrency constructs [JHA14, JA15, JA16b, JA16a, JA18]. A textual syntax for Reo that preserves its declarative, compositional nature, allows user-defined primitives, and faithfully complies with the semantics of its nodes can significantly facilitate the uptake of Reo for specification of protocols in large-scale practical applications.

In this chapter, we introduce Treo, a declarative textual language for component-based specification of Reo connectors with user-defined semantic sorts and predefined node behavior. We describe the structure of a Treo file by means of an abstract syntax (Section 4.1). In Listing 4.1, we provide a concrete syntax of Treo as an ANTLR4 grammar [Par13]. In on-going work, we currently use Treo to compile Reo into target languages such as Java, Promela, and Maude [Reo]. The construction of the Treo compiler is based on the theory of stream constraints [DA18a].

In order to preserve the agnosticism of Reo regarding the concrete semantics of its primitives, Treo uses the notion of user-defined *semantic sorts*. A user-defined semantic sort consist of a set of component instances together with a composition operator $\wedge$, a substitution operator $[\,/\,]$, and a trivial component $\top$ (Section 4.2). The composition operator defines the behavior of composite components as a composition of its operands. The substitution operator binds nodes in the interface or passes values to parameters.

For a given semantic sort, we define the meaning of abstract Treo programs (Section 4.3). Treo is very liberal with respect to parameter values. A component definition not only accepts the usual (structured) data as actual parameters, but also other component instances and other component definitions. Among other benefits, this flexible parameter passing supports *component sharing*, which is useful to preserve component encapsulation [BCL$^+$06, Figure 2].

A given semantics sort may possibly distinguish between inputs and outputs. Thus, not all combinations of components may result in a valid composite component. For example, the composition may not be defined, if two components share an output. In Treo, however, it is safe to compose components on their outputs, because, complying with the semantics of Reo, the compiler inserts special *node components* to ensure well-formed compositions (Section 4.4).

We conclude by discussing related work (Section 4.5), and pointing out future work (Section 4.6).

4.1 Treo syntax

We now present a textual representation for the graphical Reo connectors in Section 2.1.2. Table 4.1 shows the abstract syntax of Treo.

$$
\begin{array}{llll}
K & ::= & I \mid KND & D ::= V \mid \langle U_0 \rangle (U_1) \{C\} \\
L & ::= & \epsilon \mid L, T \mid L, T_0..T_1 & C ::= V \mid A \mid C_0 C_1 \mid \{C \mid P\} \mid D\langle L \rangle (U) \\
U & ::= & \epsilon \mid U, V & T ::= V \mid C \mid D \mid [L] \mid T_0 : T_1 \mid T[L] \mid F(L) \\
V & ::= & N \mid V[L] & P ::= V \in T \mid R(L) \mid \neg P \mid P_0 \wedge P_1 \mid P_0 \vee P_1 \mid (P)
\end{array}
$$

Table 4.1: Abstract syntax of Treo, with start symbol K (a source file), and terminal symbols for imports (I), primitive components (A), functions (F), relations (R), names (N), and the empty list (ϵ). The bold vertical bar in $\{C \mid P\}$ is just text.

We introduce the symbols in the abstract syntax by identifying them in some concrete examples. These concrete examples are Treo programs that can be parsed using the concrete Treo syntax shown in Listing 4.1.

```
grammar Treo;
file     : sec? imp* assg* EOF;
sec      : 'section' name ';';
imp      : 'import' name ';';
assg     : ID defn;
defn     : var | params? nodes comp;
comp     : defn vals? args | var | '{' atom+ '}' | '{' comp* ('|' pred)? '}'
         | 'for' '(' ID 'in' list ')' comp
         | 'if' '(' pred ')' comp ('else' '(' pred ')' comp)* ('else' comp)?;
atom     : STRING ; /* Example syntax for primitive components */
pred     : 'true' | 'false' | '(' pred ')' | var 'in' list
         | term op=('<=' | '<' | '>=' | '>' | '=' | '!=') term
         | var | 'forall' ID 'in' list ':' pred
         | 'exists' ID 'in' list ':' pred | 'not' pred | pred ('and'|',') pred
         | pred 'or' pred | pred 'implies' pred;
term     : var | NAT | BOOL | STRING | DEC | comp | defn | list | 'len(' term ')'
         | '(' term ')' | <assoc=right> term list | <assoc=right> term '^' term
         | '-' term | term op=('*' | '/' | '%' | '+' | '-') term;
vals     : '<' '>' | '<' term (',' term)* '>';
list     : '[' ']' | '[' item (',' item)* ']';
item     : term | term '..' term | term ':' term;
args     : '(' ')' | '(' var (',' var)* ')';
params   : '<' '>' | '<' var (',' var)* '>';
nodes    : '(' ')' | '(' node (',' node)* ')';
node     : var (io=('?' | '!' | ':') ID?)?;
var      : name list*;
name     : (ID '.')* ID;
NAT      : ('0' | [1-9][0-9]*);
DEC      : ('0' | [1-9][0-9]*) '.' [0-9]+;
BOOL     : 'true' | 'false';
ID       : [a-zA-Z_][a-zA-Z0-9_]*;
STRING   : '\"' .*? '\"';
SPACES   : [ \t\r\n]+ -> skip;
SL_COMM  : '//' .*? ('\n'|EOF) -> skip;
ML_COMM  : '/*' .*? '*/' -> skip;
```

Listing 4.1: Concrete ANTLR4 syntax of Treo (Treo.g4).

Consider the following Treo file (K in Table 4.1) representing the $\mathsf{Alternator}_2$:

```
import syncdrain;
import sync;
import fifo1;
```

```
alternator2(a1,a2,b1) {
  sync(a1,b1) syncdrain(a1,a2) sync(a2,b2) fifo1(b2,b1)
}
```

On the first line, we import (I) three different *component definitions*. On the second line, we define the `alternator2` component (ND). Its definition (D) has no parameters ($\langle U_0 \rangle$), and three nodes, `a1`, `a2`, and `b1`, in its interface ((U_1)). The body ($\{C\}$) of this definition consists of a set of *component instances* that interact via shared nodes. The first component instance `sync(a1,b1)` is an instantiation ($D\langle L \rangle(U)$) of the imported `sync` definition (D) with nodes `a1` and `b1` ((U)) and without any parameters ($\langle L \rangle$).

All nodes that occur in the body, but not in the interface, are hidden. Hiding renames a node to a fresh inaccessible name, which prevents it from being shared with other components. In the case of `alternator2`, node `b2` is not part of the interface, and hence hidden.

Constructed from existing components, `alternator2` is a *composite* component (C_0C_1). However, not every component is constructed from existing components, and we call such components *primitive* (A). The following Treo code shows a possible (primitive) definition of the`fifo1` component.

```
fifo1(a?,b!) { empty -{a},true-> full; full -{b},true-> empty; }
```

The definition of the `fifo1` differs from the definition of the `alternator2` in two ways.

The first difference is that the `fifo1` component is (in this case) defined directly as a *constraint automaton* [BSAR06]. Constraint automata constitute a popular semantic sort for specification of Reo component types, and forms the basis of the Lykos compiler [Jon16]. However, constraint automata are not the *de facto* standard: the literature offers more than thirty different semantic sorts for specification of Reo components [JA12], such as the coloring semantics and timed data stream semantics. To accommodate the generality that disparate semantics allow, Treo features *user-defined semantic sorts*, which means that the syntax for primitive components is user-defined. For example, this means that we may also define the `fifo1` component by referring to a Java file via `fifo1(a?,b!){ "MyFIFO1.java" }`.

The second difference is that the nodes `a` and `b` in the interface are *directed*. That is, each of its interface nodes is either of type input or output, designated by the markers `?` and `!`, respectively. In Reo, it is safe to join two channels on a shared sink node (e.g., node b_1 in Figure 2.3). However, the composition operators in most Reo semantics do not automatically produce the correct behavior for such nodes (e.g., see [BSAR06, Section 4.3] for further details). Therefore, most Reo semantics require *well-formed* compositions, wherein each node has at most one input channel end and at most one output channel end.

The restriction of well-formed compositions can be very inconvenient in practice. To ensure well-formed compositions, a programmer must implement every Reo node with more than one input or output channel end as a *node component*. The interface of this node component is determined by its *degree*, which is a pair (i, o) giving the numbers of its coincident source and sink ends. Such explicit node components make component constructions verbose and hard to maintain. For convenience, the Treo compiler uses the above input/output annotations to compute the degree of

each node in a composition, and subsequently inserts the correct node components in the construction. We may view the input/output annotations as syntactic sugar that ensures well-formed compositions. This feature allows programmers to remain oblivious to these annotations and well-formed composition.

The ellipses in Figure 2.3 signify the parametrized construction of the $\mathsf{Alternator}_k$ connector, for $k > 2$. This notation is informal and not supported in the graphical Reo editor [ECT], which offers no support for parametrized constructions. In Treo, however, we can define the $\mathsf{Alternator}_k$ connector as:

```
alternator<k>(a[1:k],b[1]) {
  sync(a[1],b[1])
  {
    syncdrain(a[i-1],a[i])
    sync(a[i],b[i])
    fifo1(b[i],b[i-1])
    | i in [2..k]
  }
}
```

The definition of the `alternator` depends on a parameter `k`. Since Treo is a strongly typed language with type-inferencing, there is no need to specify a type for the (integer) parameter `k`. The interface consists of an array of nodes `a[1:k]` and the single node `b[1]`. Here, `[1:k]` is an abbreviation for the list `[[1..k]]` that contains a single list of length `k`. The array `a[1:2]` stands for the *slice* `[a[1],a[2]]` of `a`, while the expression `a[1..2]` stands for the element `a[1][2]` in `a` (cf., Equation (4.2)). For iteration, we write `{ ... | i in [2..k] }` using set-comprehension ($\{C \mid P\}$ in Table 4.1).

Instead of defining `alternator` iteratively, we may also provide a recursive definition as follows:

```
recursive_alternator(a[1:k],b[1],b[k]) {
  recursive_alternator(a[1:k-1],b[1],b[k-1])
  {syncdrain(a[k-1],a[k]) sync(a[k],b[k]) fifo1(b[k-1],b[k]) | k > 1}
}
```

Here, the value of `k` is defined by the size of `a[1:k]`, and we use set-comprehension `{ ... | k > 1 }` for conditional construction, as well. Indeed, the resulting set of component instances is non-empty, only if `k > 1` holds. Although Treo syntax allows recursive definitions, the semantics presented in Section 4.3 does not yet support recursion, which we leave as future work.

We illustrate the practicality of Treo by providing code for a chess playing program [Jon16, Figure 3.29]. In this program, two teams of chess engines compete in a game of chess. We define a chess team as the following Treo component:

```
import parse; /* and the other imports */
team<engine[1:n]>(inp,out) {
  for (i in [1..n]) {
    engine[i](inp,best[i]) parse(best[i],p[i])
    if (i > 1) concatenate(a[i-1],p[i],a[i])
  }
```

```
  sync(best[1],a[1]) majority(a[n],b) syncdrain(b,c)
  fifo1(inp,c) move(b,d) concatenate(c,d,out)
}
```

The for-loop `for (i in [1..n]) ...` and if-statement `if (i > 1) ...` are just syntactic sugar for set-comprehensions `{... | i in [1..n]}` and `{... | i > 1}`, respectively. The `team` component depends on an array `engine[1:n]` of parameters. This array does not contain the usual data values, but consists of Treo component definitions. In the body of the `team` component, these definitions are instantiated via `engine[i](inp,best[i])`. In RSL [BBKK09, Klü12] and FOCAML [Jon16], it is impossible to pass a component as a parameter, which makes these languages less expressive than Treo.

We may view the `team` component as an example of role-oriented programming [CDB+16]. Indeed, the `team` component encapsulates a list of chess engines in a component, so that they can collectively be used as a single participant in a chess match:

```
match() {
  fifo1full<"">(a,b) fifo1(c,d)
  team<[eng1, eng2]>(a,d) team<[eng3]>(b,c)
}
```

Treo treats not only component definitions, but also component instances as values. By passing a single component instance as a parameter to multiple components, this feature allows *component (instance) sharing* (cf., [BCL+06, Figure 2]). Hence, it is straightforward to implement a chess match, wherein a single instance of a chess engine plays against itself.

4.2 Semantic sorts

As noted in Section 4.1, Reo channels can be defined in many different semantic formalisms [JA12], such as the constraint automaton semantics, the coloring semantics, or the timed data stream semantics. Although each sort of Reo semantics has its unique properties, each of them can be used to define a collection of composable components with parameters and nodes, which we call a *semantic sort*:

Definition 4.2.1 (Semantic sort)**.** A semantic sort over a set of names $\mathcal{N}$ with values from $\mathcal{V}$ is a tuple $(\mathcal{C}, \wedge, [\,/\,], \top)$ that consists of a set of components $\mathcal{C}$, a composition operator $\wedge : \mathcal{C} \times \mathcal{C} \longrightarrow \mathcal{C}$, a substitution operator $[\,/\,] : \mathcal{C} \times (\mathcal{N} \cup \mathcal{V}) \times \mathcal{N} \longrightarrow \mathcal{C}$, and a trivial component $\top \in \mathcal{C}$.

We assume that the set of names and the set of values are disjoint, i.e., $\mathcal{N} \cap \mathcal{V} = \emptyset$. For convenience, we write $C \wedge C'$ for $\wedge(C, C')$, and $C[y/x]$ for $[\,/\,](C, y, x)$. For any semantic sort T, we write $\mathcal{C}_T$ for its set of components, $\wedge_T$ for its composition operator, $[\,/\,]_T$ for its substitution operator, and $\top_T$ for its trivial component. The composition operator $\wedge_T$ ensures that the behavior of finite non-empty compositions is well-defined. To empty compositions we assign the trivial component $\top_T$. The substitution operator $[\,/\,]_T$ allows us to change the interface of a component via renaming or instantiation. Let $C \in \mathcal{C}_T$ be a component and $x \in \mathcal{N}$ a name.

For a name $y \in \mathcal{N}$, the construct $C[y/x]_T$ renames every occurrence of name x in C to y. For a value $y \in \mathcal{V}$, the construct $C[y/x]_T$ instantiates (parameter) x in C to y.(See Example 4.2.3 for an example of the distinction between renaming and instantiations.)

A semantic sort T implicitly defines an interface for each component $C \in \mathcal{C}$ via the map $\text{supp} : \mathcal{C}_T \longrightarrow 2^{\mathcal{N}}$ defined as $\text{supp}(C) = \{x \in \mathcal{N} \mid C[y/x]_T \neq C$, for some name $y \in \mathcal{N}\}$. If name x does not 'occur' in C, substitution of x by any name y does not affect C, i.e., $C[y/x]_T = C$.

Example 4.2.1 (Systems of differential equations)**.** The set ODE of systems of ordinary differential equations with variables from $\mathcal{N}$ and values $\mathcal{V} = \{v : \mathbb{R} \longrightarrow \mathbb{R}\}$ constitute a semantic sort. Composition is union, substitution is binding a name or value to a given name, and the trivial component is the empty system of equations. Using the ODE semantic sort, we can define continuous systems in Treo. ◊

Example 4.2.2 (Process calculi)**.** Consider the process calculus CSP, proposed by Hoare [Hoa78]. The set CSP of all such process algebraic terms comprises a semantic sort. Each process can participate in a number of events, which we can interpret as names from a given set $\mathcal{N}$. We model the composition of CSP processes P and Q by means of the interface parallel operator $P \,|[X]|\, Q$, where $X \subseteq \mathcal{N}$ is the set of event names shared by P and Q. We define substitution as simply (1) renaming the event, if a name is substituted for an event; or (2) hiding the event, if a values is substituted for an event. Since neither STOP nor SKIP shares any event with its environment, we may use either one to denote the trivial component. ◊

Example 4.2.3 (I/O-components)**.** Let T be a semantic sort over $\mathcal{N}$ and $\mathcal{V}$. We define the I/O-component sort IO_T over T using the notion of a primitive I/O-component of sort T.

A *primitive I/O-component* P of sort T is a tuple (C, I, O), where $C \in \mathcal{C}_T$ is a component, $I \subseteq \mathcal{N}$ is a set of input names, $O \subseteq \mathcal{N}$ is a set of output names. For $P \subseteq \mathcal{N}$ and $x \in \mathcal{N}$ and $y \in \mathcal{N} \cup \mathcal{V}$, define

$$P[y/x] = \begin{cases} (P - \{x\}) \cup \{y\} & \text{if } x \in P \text{ and } y \in \mathcal{N} \\ P - \{x\} & \text{if } x \in P \text{ and } y \in \mathcal{V} \\ P & \text{otherwise} \end{cases} \tag{4.1}$$

We define substitution on primitive I/O-components as

$$(C, I, O)[y/x] = (C[y/x], I[y/x], O[y/x]),$$

for all $x \in \mathcal{N}$ and $y \in \mathcal{N} \cup \mathcal{V}$. We denote the set of primitive I/O-components over T as $\mathcal{P}_T$.

An *I/O-component* of sort T is a sequence $P_1 \cdots P_n \in \mathcal{P}_T^*$, with $n \geq 0$, of primitive I/O-components of sort T. Composition of I/O-components is concatenation $\cdot$ of sequences. The trivial I/O-component is the empty sequence ϵ. We define substitution of composite I/O-components as $(P_1 \cdots P_n)[y/x] = P_1[y/x] \cdots P_n[y/x]$, for all $x \in \mathcal{N}$ and $y \in \mathcal{N} \cup \mathcal{V}$. Hence, $\text{IO}_T = (\mathcal{P}_T^*, \cdot, [\,/\,], \epsilon)$ is a semantic sort. ◊

4.3 Denotational semantics

We define the denotational semantics of the Treo language over a fixed, but arbitrary, semantic sort T. The main purpose of this denotational semantics is to provide a clear abstract structure that guides the implementation of Treo parsers. The syntax to which this denotational semantics applies is the abstract syntax in Table 4.1. The general structure of our denotational semantics is quite standard, and adheres to Schmidt's notation [Sch86].

Although Treo syntax allows recursive definitions, the semantics presented in this section does not support this feature. Since not all recursive definitions define finite compositions of components, extending the current semantics with recursion is not straightforward, and we leave it as future work.

Variables and terms in Treo are structured as non-rectangular arrays. The set of all *(ragged) arrays* over a set X is the smallest set $X^\square$ such that both $X \subseteq X^\square$ and $[x_0, \ldots, x_{n-1}] \in X^\square$, if $n \geq 0$ and $x_i \in X^\square$ for all $0 \leq i < n$. For example, the set $\mathbb{N}^\square$ of ragged arrays over integers contains all natural numbers from $\mathbb{N}$ as 'atomic' arrays, as well as the array $[37, [], [[2, [55], 3]]] \in \mathbb{N}^\square$. Every ragged array has a length, which can be computed via the map $\text{len} : X^\square \longrightarrow \mathbb{N}$ defined inductively as $\text{len}(x) = 0$, if $x \in X$, and $\text{len}([x_0, \ldots, x_{n-1}]) = n$, otherwise. If $x = [x_0, \ldots, x_{n-1}] \in X^\square$ is a ragged array, we access its entries via the function application $x(i) = x_i$, for every $0 \leq i < n$. We extend the access map $\mathbb{N}^\square$ by defining $x([i_0, \ldots, i_n])$ as

$$\begin{cases} x(i_0)([i_1, \ldots, i_n]) & \text{if } i_0 \in \mathbb{N} \\ [x(i_{00})([i_1, \ldots, i_n]), \ldots, x(i_{0m})([i_1, \ldots, i_n])] & \text{if } i_0 = [i_{00}, \ldots, i_{0m}] \end{cases}, \tag{4.2}$$

whenever the right-hand side is defined. Two ragged arrays $x \in X^\square$ and $y \in Y^\square$ have the same structure ($x \simeq y$) iff $x \in X$ and $y \in Y$, or $\text{len}(x) = \text{len}(y)$ and $x(i) \simeq y(i)$ for all $0 \leq i < \text{len}(x)$. We can flatten a ragged array from $X^\square$ to a sequence over X via the map $\text{flatten} : X^\square \longrightarrow X^*$ defined as $\text{flatten}(x) = x$, if $x \in X$, and $\text{flatten}([x_0, \ldots, x_{n-1}]) = \text{flatten}(x_0) \cdots \text{flatten}(x_{n-1})$, otherwise.

Suppose that semantic sort T is defined over a set of names $\mathcal{N}$ and a set of values $\mathcal{V}$, with $\mathcal{N} \cap \mathcal{V} = \emptyset$. For simplicity, we assume that, for every component $C \in \mathcal{C}_T$, its support $\text{supp}(C) \subseteq \mathcal{N}$ is finite. Since Treo views components as values, we assume the inclusion $\mathcal{C}_T \subseteq \mathcal{V}$.

We assume that the set of names $\mathcal{N}$ is closed under taking subscripts from $\mathbb{N}$. That is, if $x \in \mathcal{N}$ is a name and $i \in \mathbb{N}$ is a natural number, then we can construct a fresh name $x_i \in \mathcal{N}$. To construct sequences of data with variable lengths, we use a map $\text{lst} : \mathbb{N}^2 \longrightarrow \mathbb{N}^\square$ that constructs from a pair $(i, j) \in \mathbb{N}^2$ of integers a finite ordered list $[i, i+1, \ldots, j]$ in $\mathbb{N}^\square$.

Recall from Section 4.1 that a component accepts an arbitrary but finite number of parameters and nodes. Therefore, we define a component definition as a map $D : \mathcal{V}^\square \times \mathcal{N}^\square \longrightarrow \mathcal{C}_T \cup \{\lightning\}$ that takes an array of parameter values from $\mathcal{V}^\square$ and an array of nodes from $\mathcal{N}^\square$ and returns a component or an *error* $\lightning$. Let $\mathcal{D} = (\mathcal{C}_T \cup \{\lightning\})^{\mathcal{V}^\square \times \mathcal{N}^\square}$ be the set of all definitions. As mentioned earlier, Treo also allows definitions as values, which amounts to the inclusion $\mathcal{D} \subseteq \mathcal{V}$.[2]

[2] Such a set of values $\mathcal{V}$ exists only if $\mathcal{V} \mapsto \mathcal{C}_T \cup (\mathcal{C}_T \cup \{\lightning\})^{\mathcal{V}^\square \times \mathcal{N}^\square}$ admits a pre-fixed point. In this work, we simply assume that such $\mathcal{V}$ exists.

We evaluate every Treo construct in its *scope* $\sigma : N \longrightarrow \mathcal{V}^{\square}$, with $N \subseteq \mathcal{N}$ finite, which assigns a value to a finite collection of locally defined names. We write $\Sigma = \{\sigma : N \longrightarrow \mathcal{V}^{\square} \mid N \subseteq \mathcal{N} \text{ finite}\}$ for the set of scopes. For a name $x \in \mathcal{N}$ and a value $d \in \mathcal{V}^{\square}$, we have a scope $\{x \mapsto d\} : \{x\} \longrightarrow \mathcal{V}^{\square}$ defined as $\{x \mapsto d\}(x) = d$. For any two scopes $\sigma, \sigma' \in \Sigma$, we have a composition $\sigma\sigma' \in \Sigma$ such that for every $x \in \mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma')$ we have $(\sigma\sigma')(x) = \sigma'(x)$, if $x \in \mathrm{dom}(\sigma')$, and $(\sigma\sigma')(x) = \sigma(x)$, otherwise. The composite scope $\sigma\sigma'$ can be viewed as an extension of σ that includes definitions and updates from σ'.

Let Names be the set of parse trees with root N, and let $\mathbf{N}[\![-]\!] : \text{Names} \longrightarrow \mathcal{N}$ be the semantics of names. We define the semantics of variables as a map $\mathbf{V}[\![-]\!] : \text{Variables} \longrightarrow (\mathcal{N}^{\square} \cup \{\lightning\})^{\Sigma}$, where Variables is the set of parse trees with root V. For a scope $\sigma \in \Sigma$, we define $\mathbf{V}[\![-]\!](\sigma)$ as follows:

1. $\mathbf{V}[\![N]\!](\sigma) = \mathbf{N}[\![N]\!]$;
2. $\mathbf{V}[\![V[L]]\!](\sigma) = \begin{cases} x(k) & \text{if } \mathbf{V}[\![V]\!](\sigma) = x \in \mathcal{N}^{\square} \text{ and } \mathbf{L}[\![L]\!](\sigma) = k \in \mathbb{N}^{\square} \\ \lightning & \text{otherwise} \end{cases}$.

Since $\mathcal{N}$ is closed under taking subscripts, we can define $n(i) = n_i$, for all $n \in \mathcal{N}$ and $i \in \mathbb{N}$, which ensures that $x(k) \in \mathcal{N}^{\square}$ is always defined.

The semantics of arguments is a map $\mathbf{U}[\![-]\!] : \text{Arguments} \longrightarrow (\mathcal{N}^{\square} \cup \{\lightning\})^{\Sigma}$, where Arguments is the set of all parse trees with root U. For a scope $\sigma \in \Sigma$, we define $\mathbf{U}[\![-]\!](\sigma)$ as follows:

1. $\mathbf{U}[\![\epsilon]\!](\sigma) = []$;
2. $\mathbf{U}[\![U, V]\!](\sigma) = \begin{cases} [x_1, \ldots, x_{n+1}] & \text{if } \mathbf{U}[\![U]\!](\sigma) = [x_1, \ldots, x_n] \text{ and } \mathbf{V}[\![V]\!](\sigma) = x_{n+1} \\ \lightning & \text{otherwise} \end{cases}$.

Let Functions be the set of parse trees with root F, and let $\mathbf{F}[\![-]\!] : \text{Functions} \longrightarrow \{\mathcal{V}^k \longrightarrow \mathcal{V} \mid k \in \mathbb{N}\}$ be the semantics of functions. The semantics of terms is a map $\mathbf{T}[\![-]\!] : \text{Terms} \longrightarrow (\mathcal{V}^{\square} \cup \{\lightning\})^{\Sigma}$, where Terms is the set of parse trees with root T. For a scope $\sigma \in \Sigma$, we define $\mathbf{T}[\![-]\!](\sigma)$ inductively as follows:

1. $\mathbf{T}[\![V]\!](\sigma) = \begin{cases} \sigma(\mathbf{V}[\![V]\!](\sigma)) & \text{if defined} \\ \lightning & \text{otherwise} \end{cases}$;
2. $\mathbf{T}[\![C]\!](\sigma) = \mathbf{C}[\![C]\!](\sigma)$, which is well-defined since $\mathcal{C}_T \subseteq \mathcal{V}$;
3. $\mathbf{T}[\![D]\!](\sigma) = \mathbf{D}[\![D]\!](\sigma)$, which is well-defined since $\mathcal{D} \subseteq \mathcal{V}$;
4. $\mathbf{T}[\![[L]]\!](\sigma) = \mathbf{L}[\![L]\!](\sigma)$;
5. $\mathbf{T}[\![T_0 : T_1]\!](\sigma) = \begin{cases} \mathrm{lst}(x_0, x_1 - 1) & \text{if } \mathbf{T}[\![T_i]\!](\sigma) = x_i \in \mathbb{N} \text{ for } i \in \{0, 1\} \\ \lightning & \text{otherwise} \end{cases}$;
6. $\mathbf{T}[\![T[L]]\!](\sigma) = \begin{cases} x(k) & \text{if } \mathbf{T}[\![T]\!](\sigma) = x \in \mathcal{V}^{\square} \text{ and } \mathbf{L}[\![L]\!](\sigma) = k \in \mathbb{N}^{\square} \\ \lightning & \text{otherwise} \end{cases}$;

7. $\mathbf{T}[\![F(L)]\!](\sigma) = \begin{cases} \mathbf{F}[\![F]\!](\mathbf{L}[\![L]\!](\sigma)) & \text{if } \mathbf{F}[\![F]\!] : \mathcal{V}^k \longrightarrow \mathcal{V} \text{ and } \text{len}(\mathbf{L}[\![L]\!](\sigma)) = k \\ \lightning & \text{otherwise} \end{cases}$.

The semantics of lists is a map $\mathbf{L}[\![-]\!] : \text{Lists} \longrightarrow (\mathcal{V}^\square \cup \{\lightning\})^\Sigma$, where Lists is the set of parse trees with root L. For a given scope $\sigma \in \Sigma$, we define $\mathbf{S}[\![-]\!](\sigma)$ inductively as follows:

1. $\mathbf{L}[\![\epsilon]\!](\sigma) = []$;

2. $\mathbf{L}[\![L, T]\!](\sigma) = \begin{cases} [x_1, \ldots, x_{n+1}] & \text{if } \mathbf{L}[\![L]\!](\sigma) = [x_1, \ldots, x_n] \in \mathcal{V}^\square \\ & \text{and } \mathbf{T}[\![T]\!](\sigma) = x_{n+1} \in \mathcal{V} \\ \lightning & \text{otherwise} \end{cases}$;

3. $\mathbf{L}[\![L, T_0..T_1]\!](\sigma) = \begin{cases} [x_1, \ldots, x_{n+k}] & \text{if } \mathbf{L}[\![L]\!](\sigma) = [x_1, \ldots, x_n] \in \mathcal{V}^\square, \\ & \mathbf{T}[\![T_i]\!](\sigma) = a_i \in \mathcal{V}, \text{for } i \in \{0, 1\}, \\ & \text{and } \text{lst}(a_0, a_1) = [x_{n+1}, \ldots, x_{n+k}] \\ \lightning & \text{otherwise} \end{cases}$.

Since we use predicates in Treo for list comprehension, we define the semantics of predicates as a map $\mathbf{P}[\![-]\!] : \text{Predicates} \longrightarrow (2^\Sigma)^\Sigma$, where Predicates is the set of all parse trees with root P. For a scope $\sigma \in \Sigma$, we define the semantics $\mathbf{P}[\![-]\!](\sigma)$ of a predicate P as the set of all extensions of σ that satisfy P. We define $\mathbf{P}[\![-]\!](\sigma)$ inductively as follows:

1. $\mathbf{P}[\![V \in T]\!](\sigma) = \begin{cases} \{\sigma\{x \mapsto t_i\} \mid 1 \leq i \leq n\} & \text{if } \mathbf{V}[\![V]\!](\sigma) = x \notin \text{dom}(\sigma), \\ & \text{and } \mathbf{T}[\![T]\!](\sigma) = [t_1, \ldots, t_n] \\ \{\sigma\} & \text{if } \mathbf{T}[\![V]\!](\sigma) \in \mathbf{T}[\![T]\!](\sigma) \\ \emptyset & \text{otherwise} \end{cases}$,

2. If P is $R(L)$, we define $\mathbf{P}[\![R(L)]\!](\sigma) = \{\sigma' \in \Sigma \mid \sigma'\sigma = \sigma', \mathbf{L}[\![L]\!](\sigma') \in \mathbf{R}[\![R]\!]\}$;

3. If P is $\neg P$, we define $\mathbf{P}[\![\neg P]\!](\sigma) = \{\sigma' \in \Sigma \mid \sigma'\sigma = \sigma', \neg\mathbf{P}[\![P]\!](\sigma')\}$;

4. If P is $P_0 \wedge P_1$, we define $\mathbf{P}[\![P_0 \wedge P_1]\!](\sigma) = \mathbf{P}[\![P_0]\!](\sigma) \cap \mathbf{P}[\![P_1]\!](\sigma)$;

5. If P is $P_0 \vee P_1$, we define $\mathbf{P}[\![P_0 \vee P_1]\!](\sigma) = \mathbf{P}[\![P_0]\!](\sigma) \cup \mathbf{P}[\![P_1]\!](\sigma)$;

6. If P is (P), we define $\mathbf{P}[\![(P)]\!](\sigma) = \mathbf{P}[\![P]\!](\sigma)$.

For set and list comprehensions, we can iterate over only a finite subset of scopes $\mathbf{P}[\![P]\!](\sigma)$ of P. We ensure this by restricting the set of scopes to those solutions that are minimal with respect to inclusion of domains. Formally, we write $\min \mathbf{P}[\![P]\!](\sigma)$ for the set of all scopes that are minimal with respect to $\leq$ defined as $\sigma_1 \leq \sigma_2$ iff $\text{dom}(\sigma_1) \subseteq \text{dom}(\sigma_2)$, for all $\sigma_1, \sigma_2 \in \mathbf{P}[\![P]\!](\sigma)$.

The semantics of component instances is a map $\mathbf{C}[\![-]\!] : \text{Components} \longrightarrow (\mathcal{C}_T \cup \{\lightning\})^\Sigma$, where Components is the set of parse trees with root C. Recall that Treo views components as values ($\mathcal{C}_T \subseteq \mathcal{V}$). Given a scope $\sigma \in \Sigma$, we define $\mathbf{C}[\![-]\!](\sigma)$ inductively as follows:

1. $\mathbf{C}[\![V]\!](\sigma) = \begin{cases} \sigma(x) & \text{if } \mathbf{V}[\![V]\!](\sigma) = x \in \mathrm{dom}(\sigma) \text{ and } \sigma(x) \in \mathcal{C}_T \\ \lightning & \text{otherwise} \end{cases}$;

2. $\mathbf{C}[\![A]\!](\sigma) = \mathbf{A}[\![A]\!]$, where $\mathbf{A}[\![-]\!] : \text{Atoms} \longrightarrow \mathcal{C}_T$ is the semantics of primitive components;

3. $\mathbf{C}[\![C_0 C_1]\!](\sigma) = \begin{cases} \mathbf{C}[\![C_0]\!](\sigma) \wedge_T \mathbf{C}[\![C_1]\!](\sigma) & \text{if } \mathbf{C}[\![C_i]\!](\sigma) \in \mathcal{C}_T, \text{ for } i \in \{0,1\} \\ \lightning & \text{otherwise} \end{cases}$;

4. $\mathbf{C}[\![\{C : P\}]\!](\sigma) = \begin{cases} \top_T & \text{if } \min \mathbf{P}[\![P]\!](\sigma) \text{ is empty or infinite} \\ C_1 \wedge_T \cdots \wedge_T C_k & \text{if } \min \mathbf{P}[\![P]\!](\sigma) = \{\sigma_1, \ldots, \sigma_k\} \neq \emptyset, \\ & \text{and } \mathbf{C}[\![C]\!](\sigma_i) = C_i \in \mathcal{C}_T \\ \lightning & \text{otherwise} \end{cases}$;

5. $\mathbf{C}[\![D\langle L\rangle(U)]\!](\sigma) = \begin{cases} \mathbf{D}[\![D]\!](\sigma)(\mathbf{L}[\![L]\!](\sigma), \mathbf{U}[\![U]\!](\sigma)) & \text{if defined} \\ \lightning & \text{otherwise} \end{cases}$.

The semantics of component definitions is a map $\mathbf{D}[\![-]\!] : \text{Definitions} \longrightarrow (\mathcal{D} \cup \{\lightning\})^{\Sigma}$, where Definitions is the set of all parse trees with root D. For a scope $\sigma \in \Sigma$, we define $\mathbf{D}[\![-]\!](\sigma)$ as follows:

1. $\mathbf{D}[\![V]\!](\sigma) = \begin{cases} \sigma(\mathbf{V}[\![V]\!](\sigma)) & \text{if } \mathbf{V}[\![V]\!](\sigma) = x \in \mathrm{dom}(\sigma) \text{ and } \sigma(x) \in \mathcal{D} \\ \lightning & \text{otherwise} \end{cases}$;

2. If D is a component $\langle U_0\rangle(U_1)\{C\}$, then for an array of parameter values $t \in \mathcal{V}^{\square}$ and an array of nodes $q \in \mathcal{N}^{\square}$, we define $\mathbf{D}[\![\langle U_0\rangle(U_1)\{C\}]\!](\sigma)(t,q)$ as follows: Recall from Section 4.1 that the number of parameters and nodes can implicitly define variables. Suppose that there exists a unique 'index-defining' scope $\sigma' \in \Sigma$ such that for $m = \mathrm{len}(t)$ and $n = \mathrm{len}(q)$. Then we have

   (a) $\mathbf{U}[\![U_0]\!](\sigma') = [s_1, \ldots, s_m] \neq \lightning$ satisfies $s_i \simeq t(i)$, for all $1 \leq i \leq m$;
   (b) $\mathbf{U}[\![U_1]\!](\sigma') = [p_1, \cdots, p_n] \neq \lightning$ satisfies $p_i \simeq q(i)$, for all $1 \leq i \leq n$;
   (c) $\mathrm{flatten}([s_1, \ldots s_m, p_1, \ldots p_n]) \in \mathcal{N}^{\square}$ has no duplicates;
   (d) $\mathrm{dom}(\sigma') \subseteq \mathcal{N}$ is minimal such that properties (a)-(c) are satisfied.

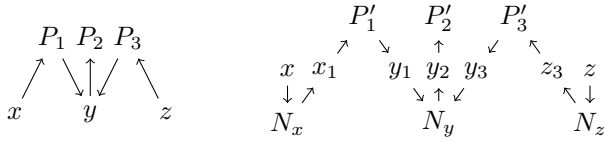
   We evaluate the body C of the component definition to the component $\mathbf{C}[\![C]\!](\sigma\sigma')$, where $\sigma\sigma'$ is the composition of σ and σ'. Define the map

$$r : \mathrm{supp}(\mathbf{C}[\![C]\!](\sigma\sigma')) \longrightarrow \mathcal{N}$$

   as

$$r(x) = \begin{cases} t_i(k_1)\cdots(k_l) & \text{if } x = s_i(k_1)\cdots(k_l) \\ q_i(k_1)\cdots(k_l) & \text{if } x = p_i(k_1)\cdots(k_l) \\ v \text{ fresh} & \text{otherwise} \end{cases}$$

   Map r is well-defined, because $\mathrm{flatten}([s_1, \ldots s_m, p_1, \ldots p_n]) \in \mathcal{N}^{\square}$ has no duplicates. Note that r is finite, since we assume that $\mathrm{supp}(\mathbf{C}[\![C]\!](\sigma\sigma'))$ is



(a) $P_1 \cdot P_2 \cdot P_3$ (b) $\mathrm{surg}(P_1 \cdot P_2 \cdot P_3)$

Figure 4.1: Surgery on an I/O-component to remove mixed nodes.

finite. We define $\mathbf{D}[\![\langle U_0\rangle(U_1)\{C\}]\!](\sigma)(t,q)$ as the simultaneous substitutions $\mathbf{C}[\![C]\!](\sigma\sigma')[r(x)/x : x \in \mathrm{dom}(f)]$. If such 'index-defining' scope σ' does not exists or is not unique, then we simply define $\mathbf{D}[\![\langle U_0\rangle(U_1)\{C\}]\!](\sigma)(t,q) = \lightning$.

We define the semantics of files as a map $\mathbf{K}[\![-]\!] : \mathrm{Files} \longrightarrow \Sigma \cup \{\lightning\}$, where Files is the set of parse trees with root K. Let $\mathbf{I}[\![-]\!] : \mathrm{Imports} \longrightarrow \Sigma$ be the semantics of imports. For a scope $\sigma \in \Sigma$, we define $\mathbf{K}[\![-]\!](\sigma)$ inductively as follows:

1. $\mathbf{K}[\![I]\!](\sigma) = \mathbf{I}[\![I]\!]$;
2. $\mathbf{K}[\![KND]\!](\sigma) = \begin{cases} \sigma_0\{x \mapsto c\} & \text{if } \sigma_0 = \mathbf{K}[\![K]\!](\sigma) \neq \lightning, x = \mathbf{N}[\![N]\!], \\ & \text{and } c = \mathbf{D}[\![D]\!](\sigma_0) \neq \lightning \\ \lightning & \text{otherwise} \end{cases}$.

4.4 Input/output nodes

As mentioned in Section 4.1, nodes of primitive component definitions require input/output annotations. Treo regards such port type annotations as attributes of the primitive component. For a semantic sort T, we model the input nodes and output nodes of its instances via two maps $I, O : \mathcal{C}_T \longrightarrow 2^{\mathcal{N}}$ satisfying $\mathrm{supp}(C) = I(C) \cup O(C)$, for all $C \in \mathcal{C}_T$. If $x \in I(C) \cap O(C)$, then we call x a *mixed* node.

Example 4.4.1 (Mixed nodes)**.** Recall the I/O component sort from Example 4.2.3. Let $P_1 = (C_1, \{x\}, \{y\})$, $P_2 = (C_2, \{y\}, \emptyset)$, and $P_3 = (C_3, \{z\}, \{y\})$ be three primitive I/O components. Figure 4.1(a) shows a graphical representation of composition of P_1, P_2, and P_3. In this figure, an arrow from a node a to a component P indicates that a is an input node of P. An arrow from a component P to a node a indicates that a is an output node of P. Node y is an output node of P_1 and P_3, and it is an input node of P_2. Thus, y is a mixed node in the composition $P_1 \cdot P_2 \cdot P_3$, where $\cdot$ is sequential composition of I/O components. ◊

Most semantic sorts that distinguish input and output nodes assume *well-formed* compositions: each shared node in a composition is an output of one component and an input of the other.

Definition 4.4.1 (Well-formedness)**.** A composition $C_1 \wedge_T \cdots \wedge_T C_n$, with $n \geq 0$, is well-formed if and only if $|\{i \in \{1,\ldots,n\} \mid x \in I(C_i)\}| \leq 1$ and $|\{i \in \{1,\ldots,n\} \mid x \in O(C_i)\}| \leq 1$, for all $x \in \mathcal{N}$.

For well-formed compositions, the behavior of the composition naturally corresponds to the composition of Reo connectors. However, specification of complex components as well-formed compositions is quite cumbersome, because it requires explicit verbose expression of the 'merge-replicate' behavior of every Reo node in terms of a suitable number of binary mergers and replicators. Reo nodes abstract from such detail and yield more concise specifications. Like Reo, Treo does not impose any restriction on the nodes of constituent components in a composition. Indeed, the denotational semantics of components $\mathbf{C}[\![-]\!]$ in Section 4.3 unconditionally computes the composition. To define the semantics of $\mathbf{C}[\![-]\!]$ for a semantic sort T where $\wedge_T$ requires well-formedness, parsing a(non-well-formed) Treo composition needs the degree (i.e., the number of coincident input and output channel ends) of each node to correctly express the 'merge-replicate' semantics of that node. The degree of every node used in a definition can be known only at the end of that definition. The Treo compiler could discover the degree of every node via two-pass parsing.

Alternatively, Treo can delay applying composition $\wedge_T$ in T until parsing completes, Treo accomplishes this by interpreting a Treo program over the I/O-component sort IO_T, as defined in Example 4.2.3, wherein compositions consist of lists of primitive components. First, Treo wraps each primitive component $C \in \mathcal{C}_T$ within a primitive I/O-component $(C, I(C), O(C)) \in \mathcal{P}_T$. Using Section 4.3, Treo parses the Treo program over the semantic sort IO_T as usual, and obtains a single I/O-component $P_1 \cdots P_n \in \mathrm{IO}_T$.

However, the resulting composition $P_1 \cdots P_n$ may not be well-formed. Therefore, the Treo compiler applies some surgery on $P_1 \cdots P_n$ to ensure a well-formed composition. This surgery consists of splitting all shared nodes in X, and reconnecting them by inserting a *node component*. We model these node components (over semantic sort T) as a map node : $(2^{\mathcal{N}})^2 \times \mathcal{N} \longrightarrow \mathcal{C}_T$. For sets of names $I, O \subseteq \mathcal{N}$ and a default name $x \in \mathcal{N}$, the component $\mathrm{node}(I, O, x) \in \mathcal{C}_T$ has input nodes I (or $\{x\}$, if I is empty) and output nodes O (or $\{x\}$, if O is empty).

Definition 4.4.2 (Surgery)**.** The surgery map surg : $\mathrm{IO}_T \longrightarrow \mathrm{IO}_T$ is defined as $\mathrm{surg}(P_1 \cdots P_n) = P'_1 \cdots P'_n \cdot \prod_{x \in \mathrm{supp}(P_1 \cdots P_n)} N_x$, where $P'_i = P_i[x_i/x : x \in \mathrm{supp}(P_i)]$, for all $1 \leq i \leq n$, and $N_x = (\mathrm{node}(I_x, O_x, x), I_x, O_x)$, with $I_x = \{x_i \mid x \in O(P_i)\}$ and $O_x = \{x_i \mid x \in I(P_i)\}$. The composition $\prod$ is ordered arbitrarily.

Intuitively, the surgery map takes a possibly non-well-formed composition and produces a well-formed composition by inserting node components. Although initially, multiple components may produce output at the same node. After applying the surgery map, these components offer data for the same node component via different 'ports'.

Example 4.4.2 (Surgery)**.** Figure 4.1(b) shows the result of applying the surgery map to the I/O-component $P_1 \cdot P_2 \cdot P_3$ from Example 4.4.1. The surgery map consists of two parts. First, the surgery map splits every node $a \in \{x, y, z\}$ by renaming a to a_i in P_i, for every $1 \leq i \leq n$. Second, the surgery map inserts at every node $a \in \{x, y, z\}$ a node component N_a. Clearly, $\mathrm{surg}(P_1 \cdot P_2 \cdot P_3)$ is a well-formed composition. $\Diamond$

4.5 Related work

The Treo syntax offers a textual representation for the graphical Reo language [Arb04, Arb11]. We propose Treo as a syntax for Reo that (1) provides support for parameterization, recursion, iteration, and conditional construction; (2) implements basic design principles of Reo more closely than existing languages; and (3) reflects its declarative nature. The graphical Reo editor implemented as an Eclipse plugin [ECT] does not support parameterization, recursion, iteration, or conditional construction. RSL (with CARML for primitives) [BBKK09, Klü12] is imperative, while Reo is declarative. FOCAML [Jon16], supports only constraint automata [BSAR06], while Treo allows arbitrary user-defined semantic sorts for expressing the behavior of Reo primitives.

Since Treo leaves the syntax for primitive subsystems (i.e., semantic sorts) as user-defined, Treo is a "meta-language" that specifies compositional construction of complex structures (using the common core language defined in this chapter) out of primitives defined in its arbitrary, user-defined sub-languages. As such, Treo is not directly comparable to any existing language. We can, however, compare the component-based system composition of Treo with the system composition of an existing language.

Treo components are similar to proctype declarations in Promela, the input language for the SPIN model checker developed by Holzmann [Hol04]. However, the focus of Promela is on imperative definitions of processes, while Treo is designed for declarative composition of processes.

SysML is a graphical language for specification of systems [FMS14]. SysML offers 9 types of diagrams, including activity diagrams and block diagrams. Each diagram provides a different view on the same system [Kru95]. Diagram types in SysML are comparable to semantic sorts in Treo. The main difference between the two, however, is that Treo requires a well-defined composition operator, using which it allows construction of more complex components, while diagram composition is much less prominent in SysML.

A component model is a programming paradigm based on components and their composition. Our Treo language can be viewed as one such component model with a concrete syntax. Over the past decades, many different component models have been proposed. For example, CORBA [OMG06] is a component model that is flat in the sense that every CORBA component is viewed as a black box, i.e., it does not support composite components. Fractal [BCL$^+$06] is an example of a component model that is hierarchical, which means a component can be a composition of subcomponents. Concrete instances of Fractal consist of libraries (API's) for a variety of programming languages, such as Java, C, and OMG IDL [BCL$^+$06]. Treo components and Fractal component differ with respect to interaction: Treo components interact via shared names, while Fractal component interact via explicit bindings.

4.6 Discussion

We propose Treo as a textual syntax for Reo connectors that allows user-defined semantic sorts, and incorporates Reo's predefined node behavior. These features

are not present in any of the existing alternative languages for Reo. We provided an abstract syntax for Treo and its denotational semantics based on this abstract syntax. We identify three possible directions for future work.

First, since our semantics disallows recursion, a component in Treo is currently restricted to consist of a composition of finitely many subsystems. Consequently, we cannot, for instance, express the construction of a primitive with an unbounded buffer, B_ω, from a set of primitives with buffer capacity of one, B_1. It seems, however, possible to use simulation and recursion to define B_ω in terms of B_1: B_ω is the smallest (with respect to simulation) component that simulates B_1 and is stable under sequential composition with B_1. These assumptions readily imply that B_ω simulates a primitive with buffer of arbitrary large capacity. Semantically, the unbounded buffer would then be defined as a least fixed point of a certain operator on components. An extension of Treo semantics that allows such fixed point definitions would provide a powerful tool to define complex 'dynamic' components.

Second, the current semantics in Section 4.3 does not support components with an identity. If we instantiate a component definition twice with the same parameters, we obtain two instances of the same component. Ideally, component instantiation should return a component instance with a fresh identity. Allowing components with identities in Treo enables programmers to design systems more realistically.

Finally, a semantic sort T from Definition 4.2.1 consists of a single composition operator $\wedge_T$. Generally, a semantic sort consists of multiple composition operators (each with it own arity). For example, we may need both sequential composition as well as parallel composition. Extending Treo with (a variable number of) composition operators would enable users to model virtually all semantic sorts.

Chapter 5

Protocols as Constraints

Given the standardized specification of Reo connections in Treo developed in Chapter 4, we now proceed with the development of a compiler that accepts Treo as input. The first step in the construction of a compiler for Treo is the selection of an appropriate semantics of Treo components. The selected semantics has serious effects on the implementability and scalability of the resulting compiler, and this decision should therefore not be taken lightly. The current chapter proposes a semantics for Reo connectors whose intermediate representation is significantly smaller than the representation of existing semantics, without sacrificing performance[1]. As a result, our approach can compile Reo connectors for which it was previously infeasible to generate efficient code.

Over a decade ago, Baier et al. introduced *constraint automata* for the specification of interaction protocols [BSAR06]. Constraint automata feature a powerful composition operator that *preserves synchrony*: composite constructions not only yield intuitively meaningful asynchronous protocols but also synchronous protocols. Constraint automata have been used as basis for tools, like compilers and model checkers. Jongmans developed Lykos: a compiler that translates constraint automata into reasonably efficient executable Java code [Jon16]. Baier, Blechmann, Klein, and Klüppelholz developed Vereofy, a model checker for constraint automata [BBKK09, Klü12]. Unfortunately, like every automaton model, composition of constraint automata suffers from state space and transition space explosions. These explosions limit the scalability of the tools based on constraint automata.

To improve scalability, Clarke et al. developed a compiler that translates a constraint automaton to a first-order formula [CPLA11]. The transitions of the constraint automaton correspond to the solutions of this formula. At run time, a generic constraint solver finds these solutions and simulates the automaton. Since composition and abstraction for constraint automata respectively correspond to conjunction and existential quantification, the first-order specification does not suffer from state space or transition space explosion. However, the approach proposed by Clarke et al. only delays the complexity until run time: calling a generic constraint solver at run time imposes a significant overhead.

Jongmans realized that the overhead of this constraint solver is not always

[1]The work in this chapter is based on [DA18a]

necessary. He developed a commandification algorithm that accepts constraints without disjunctions (i.e., conjunctions of literals) and translates them into a small imperative program [JA16b]. The resulting program is a light-weight, tailor-made constraint solver with minimal run time overhead. Since commandification accepts only constraints without disjunction, Jongmans applied this technique to data constraints on individual transitions in a constraint automaton. Relying on constraint automata, his approach still suffers from scalability issues [JKA17].

We aim to prevent state space and transition space explosions by combining the ideas of Clarke et al. and Jongmans. To this end, we present the language of *stream constraints*: a generalization of constraint automata based on temporal logic. A stream constraint is an expression that relates streams of observed data at different locations (Section 5.2). We identify a subclass of stream constraints, called *regular (stream) constraints*, which is closed under composition and abstraction (Section 5.3). Regular constraints can be viewed as a constraint automata, and conjunction of reflexive regular constraints is similar to composition of constraint automata (Section 5.4).

A straightforward application of the commandification algorithm of Jongmans to regular stream constraints entails transforming a stream constraint into disjunctive normal form and applying the algorithm to each clause separately. However, the number of clauses in the disjunctive normal form may grow exponentially in the size of the composition. To prevent such exponential blowups of the size of the formula, we recognize and exploit symmetries in the disjunctive normal form. Each clause in the disjunctive normal form can be constructed from a set of basic stream constraints, which we call *rules*. This idea allows us to represent a single large constraint as certain combination of a set of smaller constraints, called the *rule-based form* (Section 5.5). We express the composition of stream constraints in terms of the rule-based normal form (Section 5.6), and show that, for *simple* sets of rules, the number of rules to describe the composition is only linear in the size of the composition (Section 5.7). The class of stream constraints defined by a simple set of rules contains constraints for which the size of the disjunctive normal form explodes, which shows that our approach improves upon existing approaches by Clarke et al. and Jongmans. We express abstraction on stream constraints in terms of the rule-based normal form and provide a sufficient condition under which the number of rules remains constant (Section 5.8). Finally, we conclude and point out future work (Section 5.10).

5.1 Related work

Representation of stream constraints in rule-based form is part of a larger line of research on symbolic approaches, such a symbolic model checking [BCM+92, BCH+97, KNSW07] and symbolic execution [CDE+07]. These approaches not only use logic (cf., SAT solving techniques [Kem12, Ehl10] for verification), but also other implicit representations, like binary decision diagrams [Bry86] and Petri nets [Mur89]. Petri nets offer a small representation of protocols with an exponentially large state space. While our focus is more on compilation, Petri nets have been studied in the context of verification. As inspiration for future work, it is interesting to study the similarities between Petri nets and stream constraints.

Since regular stream constraints correspond to constraint automata, we can view regular stream constrains as a restricted temporal logic for which distributed synthesis is easy. In general, distributed (finite state) synthesis of protocols is undecidable [PR89, PR90]. Pushing the boundary from regular to a larger class of stream constraints can be useful for more effective synthesis methods.

5.2 Syntax and semantics

The semantics of constraint automata is defined as a relation over *timed data streams* [AR02], which are pairs, each consisting of a non-decreasing stream of time stamps and a stream of observed (exchanged) data items. The primary significance of time streams is the proper alignment of their respective data streams, by allowing "temporal gaps" during which no data is observed. For convenience, we drop the time stream and model protocols as relations over streams of data, augmented by a special symbol that designates "no-data" item.

We first define the abstract behavior of a protocol C. Fix an infinite set X of variables, and fix a non-empty set of user-data $Data \supseteq \{0\}$ that contains a datum 0. Consider the *data domain* $D = Data \cup \{*\}$ of data stream items, where we use the "no-data" symbol $* \in D \setminus Data$ to denote the absence of data. We model a single execution of protocol C as a function

$$\theta : X \longrightarrow D^{\mathbb{N}} \tag{5.1}$$

that maps every variable $x \in X$ to a function $\theta(x) : \mathbb{N} \longrightarrow D$ that represents a stream of data at location x. We call θ a *data stream tuple* (over X and D). For all $n \in \mathbb{N}$ and all $x \in X$, the value $\theta(x)(n) \in D$ is the data that we observe at location x and time step n. If $\theta(x)(n) = *$, we say that no data is observed at x in step n (i.e., we may view θ as a partial map $\mathbb{N} \times X \rightharpoonup Data$). The behavior of protocol C consists of the set

$$\mathcal{L}(C) \ \subseteq \ (D^{\mathbb{N}})^X \tag{5.2}$$

of all possible executions of C, called the *accepted language* of C. We can think of accepted language $\mathcal{L}(C)$ as a relation over data streams. In this chapter, we study protocols that are defined as a *stream constraint*:

Definition 5.2.1 (Stream constraints)**.** A stream constraint ϕ is an expression generated by the following grammar

$$\begin{array}{rcl} \phi & ::= & \bot \mid t_0 \doteq t_1 \mid \phi_0 \wedge \phi_1 \mid \neg\phi \mid \exists x\phi \mid \Box\phi \\ t & ::= & x \mid \mathbf{d} \mid t' \end{array}$$

where $x \in X$ is a variable, $d \in D$ is a datum, and t is a stream term.

We use the following standard syntactic sugar: $\top = \neg\bot$, $\phi_0 \vee \phi_1 = \neg(\neg\phi_0 \wedge \neg\phi_1)$, $\Diamond\phi = \neg\Box\neg\phi$, $(t_1 \not\doteq t_2) = \neg(t_1 \doteq t_2)$, $(t_1 \doteq \cdots \doteq t_n) = (t_1 \doteq t_2 \wedge \cdots \wedge t_{n-1} \doteq t_n)$, $t^{(0)} = t$, and $t^{(k+1)} = (t^{(k)})'$, for all $k \geq 0$. Following Rutten [Rut01], we call $t^{(k)}$, $k \geq 0$, the k-th *derivative* of term t.

We interpret a stream constraint as a constraint over streams of data in $D^{\mathbb{N}}$. For a datum $d \in D$, $\mathbf{d}$ is the constant stream defined as $\mathbf{d}(n) = d$, for all $n \in \mathbb{N}$. The

operator $(-)'$, called *stream derivative*, drops the head of the stream and is defined as $\sigma'(n) = \sigma(n+1)$, for all $n \in \mathbb{N}$ and $\sigma \in D^{\mathbb{N}}$. Streams can be related by $\doteq$ that expresses equality of their heads: $x \doteq y$ iff $x(0) = y(0)$, for all $x, y \in D^{\mathbb{N}}$. The modal operator $\Box$ allows us to express that a stream constraint holds after applying any number of derivatives to all variables. For example, $\Box(x \doteq y)$ iff $x^{(k)}(0) = y^{(k)}(0)$, for all $k \in \mathbb{N}$ and $x, y \in D^{\mathbb{N}}$. Stream constraints can be composed via conjunction $\wedge$, or negated via negation $\neg$. Streams can be hidden via existential quantification $\exists$.

Each stream term t evaluates to a data stream in $D^{\mathbb{N}}$. Let $\theta : X \longrightarrow D^{\mathbb{N}}$ be a data stream tuple. We extend the domain of θ from the set of variables X to the set of terms $T \supseteq X$ as follows: we define $\theta : T \longrightarrow D^{\mathbb{N}}$ via $\theta(\mathbf{d}) = \mathbf{d}$ and $\theta(t') = \theta(t)'$, for all $d \in D$ and terms $t \in T$.

Next, we interpret a stream constraint ϕ as a relation over streams.

Definition 5.2.2 (Semantics)**.** The language $\mathcal{L}(\phi) \subseteq (D^{\mathbb{N}})^X$ of a stream constraint ϕ over variables X and data domain D is defined as

1. $\mathcal{L}(\bot) = \emptyset$;
2. $\mathcal{L}(t_0 \doteq t_1) = \{\theta : X \longrightarrow D^{\mathbb{N}} \mid \theta(t_0)(0) = \theta(t_1)(0)\}$;
3. $\mathcal{L}(\phi_0 \wedge \phi_1) = \mathcal{L}(\phi_0) \cap \mathcal{L}(\phi_1)$;
4. $\mathcal{L}(\neg\phi) = (D^{\mathbb{N}})^X \setminus \mathcal{L}(\phi)$;
5. $\mathcal{L}(\exists x\phi) = \{\theta : X \longrightarrow D^{\mathbb{N}} \mid \theta[x \mapsto \sigma] \in \mathcal{L}(\phi)$, for some $\sigma \in D^{\mathbb{N}}\}$;
6. $\mathcal{L}(\Box\phi) = \{\theta : X \longrightarrow D^{\mathbb{N}} \mid \theta^{(k)} \in \mathcal{L}(\phi)$, for all $k \geq 0\}$,

where $\theta[x \mapsto \sigma] : X \longrightarrow D^{\mathbb{N}}$ is defined as $\theta[x \mapsto \sigma](x) = \sigma$ and $\theta[x \mapsto \sigma](y) = \theta(y)$, for all $y \in X \setminus \{x\}$; and $\theta^{(k)} : X \longrightarrow D^{\mathbb{N}}$ is defined as $\theta^{(k)}(x) = \theta(x^{(k)})$, for all $x \in X$.

Let ϕ and ψ be two stream constraints and $\theta : X \longrightarrow D^{\mathbb{N}}$ a data stream tuple. We say that θ *satisfies* ϕ (and write $\theta \models \phi$), whenever $\theta \in \mathcal{L}(\phi)$. We say that ϕ *implies* ψ (and write $\phi \models \psi$), whenever $\mathcal{L}(\phi) \subseteq \mathcal{L}(\psi)$. We call ϕ and ψ *equivalent* (and write $\phi \equiv \psi$), whenever $\mathcal{L}(\phi) = \mathcal{L}(\psi)$.

Example 5.2.1. One of the simplest stream constraints is $\mathsf{Sync}(a, b)$, which is defined as $\Box(a \doteq b)$. Constraint $\mathsf{Sync}(a, b)$ encodes that the data streams at a and b are equal: $\theta(a)(k) = \theta(b)(k)$, for all $k \in \mathbb{N}$ and all $\theta \in (D^{\mathbb{N}})^X$. Therefore, $\mathsf{Sync}(a, b)$ synchronizes the data flow observed at ports a and b.

Conjunction $\wedge$ and existential quantification $\exists$ provide natural operators for composition and abstraction for stream constraints. For example, the composition $\mathsf{Sync}(a, b) \wedge \mathsf{Sync}(b, c)$ synchronizes ports a, b, and c. Hiding port b yields $\exists b(\mathsf{Sync}(a, b) \wedge \mathsf{Sync}(b, c))$, which is equivalent to $\mathsf{Sync}(a, c)$. $\Diamond$

Example 5.2.2. Recall that $x^{(k)}$, for $k \geq 0$, is the k-th derivative of x. We can express that a stream x is periodic via the stream constraint $\Box(x^{(k)} \doteq x)$, for some $k \geq 1$. For $k = 1$, stream x is constant, like $\mathbf{0}$ and $*$. $\Diamond$

Example 5.2.3. The stream constraint $\mathsf{FIFO}(a, b, m)$ defined as $m \doteq * \wedge \Box((a \doteq m' \doteq \mathbf{0} \wedge b \doteq m \doteq *) \vee (a \doteq m' \doteq * \wedge b \doteq m \doteq \mathbf{0}) \vee (a \doteq b \doteq * \wedge m' \doteq m))$ models a 1-place buffer with input location a, output location b, and memory location m that can be full ($m \doteq \mathbf{0}$) or empty ($m \doteq *$). $\Diamond$

Example 5.2.4. Recall that $*$ models absence of data. Stream constraint $\Box\Diamond(a \neq *)$ expresses that always eventually we observe some datum at a. A constraint of such form can be used to define fairness. $\Diamond$

5.3 Regular constraints

We identify a subclass of stream constraints that naturally correspond to constraint automata. We first introduce some notation.

To denote that a string s *occurs* as a substring in a stream constraint ϕ or a stream term t, we write $s \in \phi$ or $s \in t$, respectively.

Every stream constraint ϕ admits a set $\mathrm{free}(\phi) \subseteq X$ of *free variables*, defined inductively via $\mathrm{free}(\bot) = \emptyset$, $\mathrm{free}(t_0 \doteq t_1) = \{x \in X \mid x \in t_0 \text{ or } x \in t_1\}$, $\mathrm{free}(\phi_0 \wedge \phi_1) = \mathrm{free}(\phi_0) \cup \mathrm{free}(\phi_1)$, $\mathrm{free}(\neg\phi) = \mathrm{free}(\Box\phi) = \mathrm{free}(\phi)$, and $\mathrm{free}(\exists x\phi) = \mathrm{free}(\phi) \setminus \{x\}$.

For every variable $x \in X$, we define the *degree of x in ϕ* as

$$\deg_x(\phi) \;=\; \max(\{-1\} \cup \{k \geq 0 \mid x^{(k)} \in \phi\}),$$

and the *degree of* ϕ as $\deg(\phi) = \max_{x \in X} \deg_x(\phi)$. Note that for $x \notin \phi$ we have $\deg_x(\phi) = -1$. For $k \geq 0$, we write $\mathrm{free}^k(\phi) = \{x \in \mathrm{free}(\phi) \mid \deg_x(\phi) = k\}$ for the set of all free variables of ϕ of degree k.

We call a variable x of degree zero in ϕ a *port variable* and write $P(\phi) = \mathrm{free}^0(\phi)$ for the set of port variables of ϕ. We call a variable x of degree one or higher in ϕ a *memory variable* and write $M(\phi) = \bigcup_{k \geq 1} \mathrm{free}^k(\phi)$ for the set of memory variables of ϕ.

Definition 5.3.1 (Regular)**.** A stream constraint ϕ is regular if and only if $\phi = \psi_0 \wedge \Box\psi$, such that $\Box \notin \psi_0 \wedge \psi$ and $\deg_x(\psi_0) < \deg_x(\psi) \leq 1$, for all $x \in X$.

For a regular stream constraint $\phi = \psi_0 \wedge \Box\psi$, we refer to ψ_0 as the *initial condition* of ϕ and we refer to ψ as the *invariant* of ϕ. Stream constraints $\mathsf{Sync}(a, b)$ and $\mathsf{FIFO}(a, b, m)$ in Examples 5.2.1 and 5.2.3 are regular stream constraints.

A regular stream constraint ϕ has an operational interpretation in terms of a labeled transition system $[\![\phi]\!]$. States of the transition system consist of maps $q : M(\phi) \longrightarrow D$ that assign data to memory locations, and its labels consist of maps $\alpha : P(\phi) \longrightarrow D$ that assign data to ports. We write $Q(\phi)$ for the set of states of ϕ and $A(\phi)$ for the set of labels of ϕ.

Definition 5.3.2 (Operational semantics)**.** The operational semantics $[\![\phi]\!]$ of a regular stream constraint $\phi = \psi_0 \wedge \Box\psi$ consists of a labeled transition system $(Q(\phi), A(\phi), \rightarrow, Q_0)$, with set of states $Q(\phi)$, set of labels $A(\phi)$, set of transitions $\rightarrow = \{(q_\phi(\theta), q_\phi(\theta'), \alpha_\phi(\theta)) \mid \theta \in \mathcal{L}(\psi)\}$, and set of initial states $Q_0 = \{q_\phi(\theta) \mid \theta \in \mathcal{L}(\psi_0 \wedge \psi)\}$, where

1. $q_\phi(\theta) : M(\phi) \longrightarrow D$ is defined as $q_\phi(\theta)(x) = \theta(x)(0)$, for $x \in M(\phi)$; and

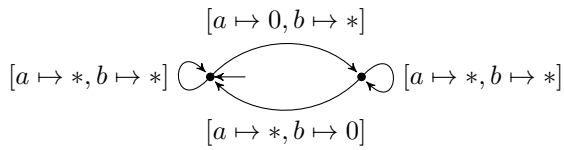


Figure 5.1: Semantics of $\mathsf{FIFO}(a, b, m)$ over the trivial data domain $\{0, *\}$.

2. $\alpha_\phi(\theta) : P(\phi) \longrightarrow D$ is defined as $\alpha_\phi(\theta)(x) = \theta(x)(0)$, for $x \in P(\phi)$.

and θ' is defined as $\theta'(x)(n) = \theta(x)(n+1)$, for all $x \in X$ and $n \in \mathbb{N}$.

Example 5.3.1. Consider the regular stream constraint $\mathsf{FIFO}(a, b, m)$ from Example 5.2.3. Note that in this example, the set of ports equals $\text{free}^0(\mathsf{FIFO}) = \{a, b\}$ and the set of memory locations equals $\text{free}^1(\mathsf{FIFO}) = \{m\}$. The semantics of $\mathsf{FIFO}(a, b, m)$ over the trivial data domain $D = \{0, *\}$ consists of 4 transitions:

1. $([m \mapsto *], [m \mapsto 0], [a \mapsto 0, b \mapsto *])$;
2. $([m \mapsto 0], [m \mapsto *], [a \mapsto *, b \mapsto 0])$; and
3. $([m \mapsto d], [m \mapsto d], [a \mapsto *, b \mapsto *])$, for every $d \in \{*, 0\}$.

Figure 5.1 shows the semantics of FIFO over the trivial data domain. ◊

Equivalent stream constraints do not necessarily have the same operational semantics. We are, therefore, interested in operational equivalence of constraints:

Definition 5.3.3 (Operational equivalence)**.** Stream constraints ϕ and ψ are operationally equivalent ($\phi \simeq \psi$) iff $\phi \equiv \psi$ and $\text{free}^k(\phi) = \text{free}^k(\psi)$, for $k \geq 0$.

Example 5.3.2. Let ϕ be a stream constraint, let t be a term and let $x \notin t$ be a variable that does not occur in t. Then, we have $\exists x(x \doteq t \wedge \phi) \equiv \phi[t/x]$, where $\phi[t/x]$ is obtained from ϕ by substituting t for every free occurrence of x. Observe that $\exists x(x \doteq t \wedge \phi)$ and $\phi[t/x]$ may admit different sets of free variables: if ϕ is just $\top$ and t is a variable y, the equivalence amounts to $\exists x(x \doteq y) \equiv \top$. To ensure that the free variables coincide, we can add the equality $t \doteq t$ and obtain the operational equivalence $\exists x(x \doteq t \wedge \phi) \simeq \phi[t/x] \wedge t \doteq t$. ◊

Operational equivalence of stream constraints ϕ and ψ implies that their operational semantics are identical, i.e., $[\![\phi]\!] = [\![\psi]\!]$. It is possible to introduce weaker equivalences by, for example, demanding that $[\![\phi]\!]$ and $[\![\psi]\!]$ are only weakly bisimilar. Such weaker equivalence offer more room for simplification of stream constraints than operational equivalence does. As our work does not need this generality, we leave the study of such weaker equivalences as future work.

The most important operations on stream constraints are composition ($\wedge$) and hiding ($\exists$). The following result shows that regular stream constraints are closed under conjunction and existential quantification of degree zero variables.

Theorem 5.3.1. *For all stream constraints ϕ and ψ and variables x, we have*

1. $\Box\phi \wedge \Box\psi \equiv \Box(\phi \wedge \psi)$; *and*

2. $\exists x \Box \phi \equiv \Box \exists x \phi$*, whenever* $\deg_x(\phi) \leq 0$ *and* $\Box \notin \phi$*.*

Proof. For assertion 1, $\mathcal{L}(\Box\phi \wedge \Box\psi) = \{\theta \in (D^{\mathbb{N}})^X \mid \forall k \geq 0 : \theta^{(k)} \models \phi \wedge \psi\} = \mathcal{L}(\Box(\phi \wedge \psi))$ shows that $\Box\phi \wedge \Box\psi \equiv \Box(\phi \wedge \psi)$.

For assertion 2, suppose that $\deg_x(\phi) \leq 0$ and $\Box \notin \phi$. We show that $\theta \in \mathcal{L}(\Box\exists x\phi)$ if and only if $\theta \in \mathcal{L}(\exists x \Box \phi)$, for all $\theta \in (D^{\mathbb{N}})^X$. By Definition 5.2.2, this equivalence can be written as

$$\theta^{(k)}[x \mapsto \mu_k] \models \phi \quad \Leftrightarrow \quad (\theta[x \mapsto \sigma])^{(k)} \models \phi, \tag{5.3}$$

for all $k \geq 0$, $\sigma \in D^{\mathbb{N}}$, and $\mu_k \in D^{\mathbb{N}}$ such that $\mu_k(0) = \sigma^{(k)}(0)$.

To prove Equation (5.3), we proceed by induction on the length of ϕ:

Case 1 ($\phi := \bot$): Since $\mathcal{L}(\bot) = \emptyset$, Equation (5.3) holds trivially.

Case 2 ($\phi := t_0 \doteq t_1$): Observe that, since $\deg_x(\phi) \leq 0$, for all terms t, we have $x \in t$ iff $t = x$. We conclude Equation (5.3) from $\mu_k(0) = \sigma^{(k)}(0)$ and

$$\theta^{(k)}[x \mapsto \mu_k](t)(0) = \left\{ \begin{matrix} \mu_k(0) & \text{if } t = x \\ \theta^{(k)}(t)(0) & \text{if } t \neq x \end{matrix} \right\} = (\theta[x \mapsto \sigma])^{(k)}(t)(0).$$

Case 3 ($\phi := \psi_0 \wedge \psi_1$): By the induction hypothesis, Equation (5.3) holds for ψ_0 and ψ_1. By conjunction of Equation (5.3), we conclude Equation (5.3) for ϕ.

Case 4 ($\phi := \neg\psi$): By the induction hypothesis, Equation (5.3) holds for ψ. By contraposition of Equation (5.3), we conclude Equation (5.3) for ϕ.

Case 5 ($\phi := \exists y \psi$): If $y = x$, then $x \notin \text{free}(\phi)$ and both sides in Equation (5.3) are equivalent to $\theta^{(k)} \models \phi$. Hence, Equation (5.3) holds for $y = x$. Suppose $y \neq x$. Then, $\theta^{(k)}[x \mapsto \mu_k] \models \phi$ is equivalent to $(\theta[y \mapsto \tau])^{(k)}[x \mapsto \mu_k] \models \psi$, for some $\tau \in D^{\mathbb{N}}$. Applying the induction hypothesis for θ equal to $\theta[y \mapsto \tau]$, we conclude that $\theta^{(k)}[x \mapsto \mu_k] \models \phi$ is equivalent to $(\theta[y \mapsto \tau][x \mapsto \sigma])^{(k)} \models \psi$, for some $\tau \in D^{\mathbb{N}}$. Since $y \neq x$, we conclude that Equation (5.3) holds.

We conclude that the claim holds for all ϕ with $\deg_x(\phi) \leq 0$ and $\Box \notin \phi$. □

5.4 Reflexive constraints

Conjunction of stream constraints is a simple syntactic composition operator with clear semantics: a data stream tuple θ satisfies a conjunction $\phi_0 \wedge \phi_1$ if and only if θ satisfies both ϕ_0 and ϕ_1. In view of the semantics of regular stream constraints in Definition 5.2.2, it is less obvious how $[\![\phi_0 \wedge \phi_1]\!]$ relates to $[\![\phi_0]\!]$ and $[\![\phi_1]\!]$. The following result characterizes their relation when no memory is shared.

Theorem 5.4.1. *Let* ϕ_0 *and* ϕ_1 *be regular stream constraints such that* $\text{free}(\phi_0) \cap \text{free}(\phi_1) \subseteq P(\phi_0 \wedge \phi_1)$*, and let* $(q_i, q_i', \alpha_i) \in Q(\phi_i)^2 \times A(\phi_i)$*, for* $i \in \{0, 1\}$*. The following are equivalent:*

1. $q_0 \xrightarrow{\alpha_0} q_0'$ *in* $[\![\phi_0]\!]$*,* $q_1 \xrightarrow{\alpha_1} q_1'$ *in* $[\![\phi_1]\!]$*, and* $\alpha_0|_{P(\phi_1)} = \alpha_1|_{P(\phi_0)}$*;*

2. $q_0 \cup q_1 \xrightarrow{\alpha_0 \cup \alpha_1} q_0' \cup q_1'$ *in* $[\![\phi_0 \wedge \phi_1]\!]$*,*

where $|$ *is restriction of maps, and* $\cup$ *is union of maps.*

Proof. Write $\phi_i = \psi_{i0} \wedge \Box\psi_i$, with $\Box \notin \psi_{i0} \wedge \psi_i$ and $\deg_x(\psi_{i0}) < \deg_x(\psi_i) \leq 1$, for all $x \in X$. Then, $\text{free}^k(\phi_i) = \text{free}^k(\psi_i)$, for all $i, k \in \{0, 1\}$.

Suppose that assertion 1 holds. By Definition 5.2.2, we find, for all $i \in \{0, 1\}$, some $\theta_i \in \mathcal{L}(\psi_i)$ such that $q_i = q_{\phi_i}(\theta_i)$, $q'_i = q_{\phi_i}(\theta'_i)$, and $\alpha_i = \alpha_{\phi_i}(\theta_i)$. Define $\theta : X \longrightarrow D^{\mathbb{N}}$ by $\theta(x) = \theta_i(x)$, if $x \in \text{free}(\phi_i)$, and $\theta(x) = *$, otherwise. Since $\text{free}(\phi_0) \cap \text{free}(\phi_1) \subseteq P(\phi_0 \wedge \phi_1)$ and $\alpha_0|_{P(\phi_1)} = \alpha_1|_{P(\phi_0)}$, we have that $\theta_0(x) = \theta_1(x)$, for all $x \in \text{free}(\phi_0) \cap \text{free}(\phi_1)$. Hence, θ is well-defined. By construction, $\theta \models \psi_0$ and $\theta \models \psi_1$. By Definition 5.2.2, we have $\theta \models \psi_0 \wedge \psi_1$. By Theorem 5.3.1, we have $\phi_0 \wedge \phi_1 = \psi_{00} \wedge \psi_{10} \wedge \Box(\psi_0 \wedge \psi_1)$. Since $q_0 \cup q_1 = q_{\phi_0 \wedge \phi_1}(\theta)$, $q'_0 \cup q'_1 = q_{\phi_0 \wedge \phi_1}(\theta')$, and $\alpha_0 \cup \alpha_1 = \alpha_{\phi_0 \wedge \phi_1}(\theta)$, we conclude assertion 2.

Suppose that assertion 2 holds. We find some $\theta \in \mathcal{L}(\psi_0 \wedge \psi_1)$, such that $q_0 \cup q_1 = q_\theta$, $q'_0 \cup q'_1 = q_{\theta'}$, and $\alpha_0 \cup \alpha_1 = \alpha_\theta$. Then, we conclude assertion 1, for $q_i = q_{\phi_i}(\theta)$, $q'_i = q_{\phi_i}(\theta')$, and $\alpha_i = \alpha_{\phi_i}(\theta)$. □

Stream constraints ϕ_0 and ϕ_1 without shared variables ($\text{free}(\phi_0) \cap \text{free}(\phi_1) = \emptyset$) seem completely independent. However, Theorem 5.4.1 shows that their composition $\phi_0 \wedge \phi_1$ admits a transition only if ϕ_0 and ϕ_1 admit respective local transitions (q_0, q'_0, α_0) and (q_1, q'_1, α_1), such that $\alpha_0|_{P(\phi_1)} = \alpha_1|_{P(\phi_0)}$. Since ϕ_0 and ϕ_1 do not share variables, the latter condition on α_0 and α_1 is trivially satisfied. Still, for one protocol ϕ_i, with $i \in \{0, 1\}$, to make progress in the composition $\phi_0 \wedge \phi_1$, constraint ϕ_{1-i} must admit an idling transition.

To allow such independent progress, we assume that ϕ_{1-i} admits an *idling* transition (q, q, τ), where τ is the silent label over $P(\phi_{1-i})$. The *silent label* over a set of ports $P \subseteq X$ is the map $\tau : P \longrightarrow D$ that maps $x \in P$ to $* \in D$. If such idling transitions are available in every state of ϕ_1, we say that ϕ_1 is *reflexive*:

Definition 5.4.1 (Reflexive)**.** A stream constraint ϕ is reflexive if and only if $q \xrightarrow{\tau} q$ in $[\![\phi]\!]$, for all $q \in Q(\phi)$.

For regular constraints, we can define reflexiveness also syntactically, for which we need some notation. For a variable $x \in X$ and an integer $k \in \mathbb{N} \cup \{-1\}$, we define the predicate $x\dagger_k$ (pronounced: "x is blocked at step k") as follows:

$$x\dagger_k \;:=\; (x^{(k)} \doteq x^{(k-1)}), \qquad \text{with } x^{(k)} \doteq *, \text{ for all } k < 0.$$

Predicate $x\dagger_{-1} \equiv \top$ is trivially true. Predicate $x\dagger_0 \equiv (x \doteq *)$ means that we observe no data flow at port x. Predicate $x\dagger_1 \equiv (x' \doteq x)$ means that the data in memory variable x remains the same.

We now provide a syntactic equivalent of Definition 5.4.1 for regular constraints.

Lemma 5.4.2. *A regular stream constraint $\phi = \psi_0 \wedge \Box\psi$ is reflexive if and only if $\bigwedge_{x \in X} x\dagger_{d(x)} \models \psi$, where $d(x) = \deg_x(\phi)$, for all $x \in X$.*

Proof. Since $d(x) = -1$, for all but finitely many $x \in X$, the stream constraint $\bigwedge_{x \in X} x\dagger_{d(x)}$ is well-defined. By definition, $\bigwedge_{x \in X} x\dagger_{d(x)} \models \psi$ if and only if, for all $q \in Q(\phi)$, there exists some $\theta \in \mathcal{L}(\psi)$, such that $q_\theta = q_{\theta'} = q$ and $\alpha_\theta = \tau$. □

Example 5.4.1. The stream constraint $\mathsf{Sync}(a, b) := \Box(a \doteq b)$ from Example 5.2.1 is reflexive, because $\bigwedge_{x \in X} x\dagger_{d(x)} = a \doteq * \wedge b \doteq *$ implies $a \doteq b$. The stream constraint FIFO from Example 5.2.3 is reflexive, because $\bigwedge_{x \in X} x\dagger_{d(x)} = a \doteq * \wedge b \doteq * \wedge m' \doteq m$ is one of the clauses of FIFO. ◊

Theorem 5.4.1 suggests a composition operator $\times$ on labeled transition systems, satisfying $[\![\phi_0]\!] \times [\![\phi_1]\!] = [\![\phi_0 \wedge \phi_1]\!]$. For reflexive constraints ϕ_0 and ϕ_1, composition $\times$ simulates composition of constraint automata [BSAR06]. Constraint automata also feature a hiding operator that naturally corresponds to existential quantification $\exists$ for stream constraints. We leave a full formal comparison between stream constraints and constraint automata as future work.

5.5 Rule-based form

The commandification algorithm developed by Jongmans accepts only constraints without disjunction (i.e., conjunctions of literals) [JA16b]. To apply commandification to the invariant ψ of an arbitrary regular stream constraint $\psi_0 \wedge \Box\psi$, we can first transform ψ into disjunctive normal form (DNF). However, the number of clauses in the disjunctive normal form may be exponential in the length of the constraint. In this section, we introduce an alternative to the disjunctive normal form that prevents such exponential blow up, for a strictly larger class of stream constraints. Our main observation is that the clauses of the disjunctive normal form may contain many symmetries, in the sense that we may generate all clauses from a set of stream constraints R, called a *set of rules*. A *rule* is a stream constraint ρ, such that $\deg(\rho) \leq 1$ and $\Box \notin \rho$.

Definition 5.5.1 (Rule-based form)**.** A reflexive stream constraint ϕ is in rule-based form iff ϕ equals

$$\mathrm{rbf}(R) \;=\; \bigwedge_{x \in \mathrm{free}(R)} \left(x\dagger_{d(x)} \vee \bigvee_{\rho \in R : x \in \mathrm{free}(\rho)} \rho \right) \tag{5.4}$$

with R a finite set of rules, $\mathrm{free}(R) = \bigcup_{\rho \in R} \mathrm{free}(\rho)$, and $d(x) = \max_{\rho \in R} \deg_x(\rho)$. A stream constraint ϕ is defined by R iff $\phi \simeq \mathrm{rbf}(R)$.

We provide some intuition behind Definition 5.5.1. For a variable x, there are two possibilities:

1. Nothing happens at x (i.e., $x\dagger_{d(x)}$). For a port variable ($d(x) = 0$) this means that we do not observe any data ($x \doteq *$). For a memory variable ($d(x) = 1$) this means that the data does not change in ($x' \doteq x$).

2. Something happens at x (i.e., $x\dagger_{d(x)}$ does not hold). Then, the rule-based form states that (at least) one of the rules with x as a free variable must hold (and this rule explains what happens at x).

Both possibilities are captured by Equation (5.4).

We apply the rule-based form to the invariant of regular constraints, via $\psi_0 \wedge \Box\,\mathrm{rbf}(R)$, for some degree zero stream constraint ψ_0 and set of rules R. Intuitively, R remains smaller than the DNF of $\mathrm{rbf}(R)$ under composition.

Example 5.5.1. Let ψ be a reflexive stream constraint, with $\deg(\psi) \leq 1$ and $\Box \notin \psi$. By Definition 5.5.1 and Lemma 5.4.2 and the distributive law, we have

$$\mathrm{rbf}(\{\psi\}) \;=\; \bigwedge_{x \in \mathrm{free}(\{\psi\})} \left(x\dagger_{\deg_x(\psi)} \vee \psi \right) \;\equiv\; \left(\bigwedge_{x \in \mathrm{free}(\{\psi\})} x\dagger_{\deg_x(\psi)} \right) \vee \psi \;\equiv\; \psi$$

Now, for $\psi = (a \doteq b)$, we get from Example 5.4.1 that $\mathsf{Sync}(a,b) = \Box(a \doteq b) \equiv \Box\,\mathrm{rbf}(\{a \doteq b\})$, which shows the Sync from Example 5.2.1 can be expressed in rule-based form. ◊

Example 5.5.2. The stream constraint $\mathsf{LossySync}(a,b) := \Box\,\mathrm{rbf}(\{a \doteq a, a \doteq b\})$ is equivalent to $\Box(b \doteq * \vee a \doteq b)$. Note that $\Box\,\mathrm{rbf}(\{\top, a \doteq b\}) \simeq \Box\,\mathrm{rbf}(\{a \doteq b\}) \simeq \mathsf{Sync}(a,b)$. Hence, rules $a \doteq a$ and $\top$ are different, because they have different sets of free variables. ◊

Example 5.5.3. The set of rules that define a stream constraint is not unique. Consider the stream constraint FIFO from Example 5.2.3. On the one hand, we have $\mathsf{FIFO}(a,b,m) \simeq m \doteq * \wedge \Box\,\mathrm{rbf}(\{\varphi,\psi\})$, where $\varphi \simeq a \doteq m' \doteq \mathbf{0} \wedge m \doteq *$ models the action that puts data in the buffer and $\psi \simeq m' \doteq * \wedge b \doteq m \doteq \mathbf{0}$ models the action that takes data out of the buffer. On the other hand, we have $\mathsf{FIFO}(a,b,m) \simeq m \doteq * \wedge \Box\,\mathrm{rbf}(\{a \doteq m' \doteq \mathbf{0} \wedge b \doteq m \doteq *, a \doteq m' \doteq * \wedge b \doteq m \doteq \mathbf{0}\})$. ◊

Example 5.5.4. Rule-based forms are an alternative to disjunctive normal forms. Consider the reflexive constraint $\phi := \bigvee_{i=1}^{n} \rho_i$ in DNF for which the first conjunctive clause ρ_1 is equivalent to $\bigwedge_{x \in \mathrm{free}(\phi)} x\dagger_{d(x)}$, with $d(x) = \deg_x(\phi)$. By adding equalities of the form $x \doteq x$, we assume without loss of generality that $\mathrm{free}(\rho_i) = \mathrm{free}(\phi)$, for all $2 \leq i \leq n$. For $R = \{\rho_i \mid 2 \leq i \leq n\}$, it follows from

$$\mathrm{rbf}(R) \;\equiv\; \bigwedge_{x \in \mathrm{free}(R)} \left(x\dagger_{d(x)} \vee \bigvee_{\rho \in R} \rho \right) \;\equiv\; \left(\bigwedge_{x \in \mathrm{free}(\phi)} x\dagger_{d(x)} \right) \vee \bigvee_{\rho \in R} \rho \;\equiv\; \phi \quad (5.5)$$

that ϕ is defined by the set R. We therefore conclude that every reflexive constraint can be written in rule-based form. ◊

Definition 5.5.1 presents the rule-based form as a conjunctive normal form. The following result computes the disjunctive normal form of $\mathrm{rbf}(R)$.

Lemma 5.5.1. *For every set of rules R, we have*

$$\mathrm{rbf}(R) \;\simeq\; \mathrm{dnf}(R) \;:=\; \bigvee_{T \subseteq R} \bigwedge_{\rho \in T} \rho \wedge \bigwedge_{x \in \mathrm{free}(R) \setminus \mathrm{free}(T)} x\dagger_{d(x)}.$$

Proof. Let $x \in X$ be arbitrary. By construction, $\deg_x(\mathrm{dnf}(R)) \leq \max_{\rho \in R} \deg_x(\rho)$. Since $d(x) = \max_{\rho \in R} \deg_x(\rho)$, the clause for $T = \emptyset$ shows that $\deg_x(\mathrm{dnf}(R)) \geq d(x)$. By Lemma 5.6.2, $\deg_x(\mathrm{rbf}(R)) = \deg_x(\mathrm{dnf}(R))$, for all $x \in X$. Hence, $\mathrm{free}^k(\mathrm{rbf}(R)) = \mathrm{free}^k(\mathrm{dnf}(R))$, for all $k \geq 0$.

Next, we show that $\mathrm{rbf}(R) \models \mathrm{dnf}(R)$. Let $\theta \in \mathcal{L}(\mathrm{rbf}(R))$. We find, for every $x \in \mathrm{free}(R)$, some rule $\rho_x \in R$, such that $\theta \models \rho$ and $x \in \mathrm{free}(\rho)$. Now, define $T_\theta := \{\rho_x \mid x \in \mathrm{free}(R) \text{ and } \theta \notin \mathcal{L}(x\dagger_{d(x)})\}$. By construction, $\theta \models \rho_x$, for every $\rho_x \in T_\theta$. If $x \in \mathrm{free}(R)$ and $\theta \notin \mathcal{L}(x\dagger_{d(x)})$, then $\rho_x \in T_\theta$ and $x \in \mathrm{free}(\rho_x) \subseteq \mathrm{free}(T_\theta)$. By contraposition, we conclude that $\theta \models x\dagger_{d(x)}$, for all $x \in \mathrm{free}(R) \setminus \mathrm{free}(T_\theta)$. Hence, $\theta \models \mathrm{dnf}(R)$, and $\mathcal{L}(\mathrm{rbf}(R)) \subseteq \mathcal{L}(\mathrm{dnf}(R))$.

Finally, we show that $\mathrm{dnf}(R) \models \mathrm{rbf}(R)$. Let $\theta \in \mathcal{L}(\mathrm{dnf}(R))$. By definition of $\mathrm{dnf}(R)$, we find some $T \subseteq R$ with $\theta \models \rho$, for all $\rho \in T$, and $\theta \models x\dagger_{d(x)}$, for all

$x \in \text{free}(R) \setminus \text{free}(T)$. Suppose that $x \in \text{free}(R)$ and $\theta \not\models x\dagger_{d(x)}$. Since $\theta \models x\dagger_{d(x)}$, for all $x \in \text{free}(R) \setminus \text{free}(T)$, we find by contraposition that $x \in \text{free}(T)$. Hence, we find some $\psi \in T$ with $x \in \text{free}(\psi)$. Since $\theta \models \rho$, for all $\rho \in T$, we find that $\theta \models \psi$. Hence, $\theta \models \text{rbf}(R)$ and we conclude that $\text{rbf}(R) \simeq \text{dnf}(R)$. □

5.6 Composition

We express conjunction of stream constraints in terms of their defining sets of rules. That is, for two sets of rules R_0 and R_1, we define the composition $R_0 \wedge R_1$ of R_0 and R_1, such that $\text{rbf}(R_0 \wedge R_1) \simeq \text{rbf}(R_0) \wedge \text{rbf}(R_1)$. If R_0 and R_1 do not share any variable (i.e., $\text{free}(R_0) \cap \text{free}(R_1) = \emptyset$), composition $R_0 \wedge R_1$ is given by the union $R_0 \cup R_1$. It is not hard to verify that $\text{dnf}(R_0 \cup R_1) \equiv \text{dnf}(R_0) \wedge \text{dnf}(R_1)$, whenever R_0 and R_1 do not share any variable. This result already demonstrates the power of the rule-based form, because the number of rules grows linearly, while the number of clauses is the disjunctive normal form grows exponentially. Recall that we compare the rule-based form with the disjunctive normal form, because Jongmans' commandification algorithm requires conjunctions of literals as input.

Of course, the assumption that R_0 and R_1 do not share any variable is very strong. In this section, we define the composition $R_0 \wedge R_1$ of R_0 and R_1 for $\text{free}(R_0) \cap \text{free}(R_1) \neq \emptyset$. Intuitively, we must find 'small' subsets $S \subseteq R_0 \cup R_1$ of rules that must synchronize (i.e., fire together) as a result of a shared variable. The conjunction of all rules in such a subset S yields a rule in the composition $R_0 \wedge R_1$.

In view of Example 5.5.4, consider the normal form $\text{dnf}(R_0 \wedge R_1)$. Since $\text{dnf}(R_0 \wedge R_1)$ equals $\text{dnf}(R_0) \wedge \text{dnf}(R_1)$, it suffices to characterize the set of clauses of $\text{dnf}(R_0) \wedge \text{dnf}(R_1)$. Every such clause is a conjunction of a clause in $\text{dnf}(R_0)$ and a clause in $\text{dnf}(R_1)$. Lemma 5.5.1 shows that the clauses of $\text{dnf}(R_i)$ correspond to subsets T_i of R_i, for all $i \in \{0, 1\}$. Not every pair of subsets $T_0 \subseteq R_0$ and $T_1 \subseteq R_1$ yields a clause of $\text{dnf}(R_0) \wedge \text{dnf}(R_1)$, but only if $S = T_0 \cup T_1$ is *synchronous*:

Definition 5.6.1 (Synchronous)**.** A synchronous set over sets of rules R_0 and R_1 is a subset $S \subseteq R_0 \cup R_1$, with $\text{free}(S) \cap \text{free}(R_i) \subseteq \text{free}(S \cap R_i)$, for all $i \in \{0, 1\}$.

Example 5.6.1. For any integer $i \geq 1$, let $\varphi_i := a_i \doteq m_i' \doteq \mathbf{0} \wedge m_i \doteq *$ and $\psi_i := m_i' \doteq * \wedge a_{i+1} \doteq m_i \doteq \mathbf{0}$ be the two rules that define $\mathsf{FIFO}(a_i, a_{i+1}, m_i)$, from Example 5.5.3. The synchronous sets consist of exactly those sets $S \subseteq \{\varphi_1, \psi_1\} \cup \{\varphi_2, \psi_2\}$ that satisfy $\psi_1 \in S$ iff $\varphi_2 \in S$. That is, the synchronous sets are given by $\emptyset$, $\{\varphi_1\}$, $\{\psi_2\}$, $\{\psi_1, \varphi_2\}$, $\{\varphi_1, \psi_1, \varphi_2\}$, $\{\psi_1, \varphi_2, \psi_2\}$, $\{\varphi_1, \psi_1, \varphi_2, \psi_2\}$. ◊

Next, we recognize symmetries in the collection of synchronous sets. We can construct every synchronous set as a union of *irreducible* synchronous subsets:

Definition 5.6.2 (Irreducibility)**.** A non-empty synchronous set $\emptyset \neq S \subseteq R_0 \cup R_1$ is irreducible if and only if $S = S_0 \cup S_1$ implies $S = S_0$ or $S = S_1$, for all synchronous subsets $S_0, S_1 \subseteq R_0 \cup R_1$.

Example 5.6.2. Let R_0 and R_1 be sets of rules, and let $\rho \in R_0$ be a rule, such that $\text{free}(\rho) \cap \text{free}(R_1) = \emptyset$. We show that $\{\rho\}$ is irreducible synchronous. Since $\text{free}(\{\rho\}) \cap \text{free}(R_0) = \text{free}(\rho) = \text{free}(\{\rho\} \cap R_0)$ and $\text{free}(\{\rho\}) \cap \text{free}(R_1) = \emptyset \subseteq \text{free}(\{\rho\} \cap R_1)$, we conclude that $\{\rho\}$ is synchronous. Suppose $\{\rho\} = S_0 \cup S_1$.

Then, $\rho \in S_i$, for some $i \in \{0,1\}$. Hence, $\{\rho\} \subseteq S_i \subseteq \{\rho\}$, which shows that $S_i = \{\rho\}$. We conclude that $\{\rho\}$ is irreducible synchronous in $R_0 \cup R_1$. ◊

Example 5.6.3. Consider φ_i and ψ_i, for $i \in \{1,2\}$, from Example 5.6.1. The irreducible synchronous sets of $\{\varphi_1, \psi_1\} \cup \{\varphi_2, \psi_2\}$ are $\{\varphi_1\}$, $\{\psi_2\}$, and $\{\psi_1, \varphi_2\}$. ◊

Definition 5.6.3 (Composition)**.** The composition of sets of rules R_0 and R_1 is $R_0 \wedge R_1 := \{\bigwedge_{\rho \in S} \rho \mid S \subseteq R_0 \cup R_1 \text{ irreducible synchronous}\}$.

Example 5.6.4. Let R_0 and R_1 be sets of rules, with $\text{free}(R_0) \cap \text{free}(R_1) = \emptyset$. By Example 5.6.2, we find that $\{\rho\} \subseteq R_0 \cup R_1$, for all $\rho \in R_0 \cup R_1$, is irreducible synchronous. Hence, every synchronous set $S \subseteq R_0 \cup R_1$, with $|S| \geq 2$, is reducible. Therefore, $S \subseteq R_0 \cup R_1$ is irreducible synchronous if and only if $S = \{\rho\}$, for some $\rho \in R_0 \cup R_1$. We conclude that $R_0 \wedge R_1 = R_0 \cup R_1$. Consequently, $\emptyset$ is a (unique) identity element with respect to composition $\wedge$ of sets of rules. ◊

To show that the composition of sets of rules coincides with conjunction of stream constraints, we need the following result that shows that every non-empty synchronous set can be covered by irreducible synchronous sets.

Lemma 5.6.1. *Let R_0 and R_1 be sets of rules, and let $S \subseteq R_0 \cup R_1$ be a non-empty synchronous set. Then, $S = \bigcup_{i=1}^{n} S_i$, where $S_i \subseteq R_0 \cup R_1$, for $1 \leq i \leq n$, is irreducible synchronous.*

Proof. We prove the lemma by induction on the size $|S|$ of S. For the base case, suppose that $|S| = 1$. We show that S is irreducible synchronous, which provides a trivial covering. Suppose that $S = S_0 \cup S_1$, for some synchronous sets $S_0, S_1 \subseteq R_0 \cup R_1$. Since, $|S| = 1$, we have $S \subseteq S_i \subseteq S$, for some $i \in \{0,1\}$. Hence, $S = S_i$, and S is irreducible. We conclude that the lemma holds, for $|S| = 1$.

For the induction step, suppose that $|S| = k > 1$, and suppose that the lemma holds, for $|S| < k$. If S is irreducible, we find a trivial covering of S. If S is reducible, we find $S = S_0 \cup S_1$, where $S_0 \neq S \neq S_1$ are synchronous sets in $R_0 \cup R_1$. Since $|S_i| < |S|$, for $i \in \{0,1\}$, we find by the hypothesis that $S_i = \bigcup_{j=1}^{n_i} S_{ij}$. Hence, $S = S_0 \cup S_1 = \bigcup_{i=0}^{1} \bigcup_{j=1}^{n_i} S_{ij}$. We conclude that the lemma holds, for $|S| = k$. By induction on $|S|$, we conclude the lemma. □

Lemma 5.6.2. $\deg_x(\text{rbf}(R)) = \max_{\rho \in R} \deg_x(\rho)$*, for all sets of rules R and $x \in X$.*

Proof. For any set of rules R and $y \in X$, we have

$$\deg_y(\text{rbf}(R)) = \max_{x \in \text{free}(R)} \max(\deg_y(x\dagger_{d(x)}), \max_{\rho \in R : x \in \text{free}(\rho)} \deg_y(\rho)).$$

Note that $\deg_y(x\dagger_{d(x)}) = d(y)$, if $y = x$, and $\deg_y(x\dagger_{d(x)}) = -1$, otherwise. Since $d(y) = \max_{\rho \in R} \deg_y(\rho)$, we have $\deg_y(\text{rbf}(R)) = \max_{\rho \in R} \deg_y(\rho)$. □

Theorem 5.6.3. $\text{rbf}(R_0 \wedge R_1) \simeq \text{rbf}(R_0) \wedge \text{rbf}(R_1)$*, for all sets of rules R_0 and R_1.*

Proof. By Lemma 5.6.2 and Definition 5.6.3, $\deg_x(\text{rbf}(R_0 \wedge R_1)) = \deg_x(\text{rbf}(R_0) \wedge \text{rbf}(R_1))$, for all $x \in X$. Hence, $\text{free}^k(\text{rbf}(R_0 \wedge R_1)) = \text{free}^k(\text{rbf}(R_0) \wedge \text{rbf}(R_1))$, for all $k \geq 0$.

Next, we show $\mathrm{rbf}(R_0) \wedge \mathrm{rbf}(R_1)) \models \mathrm{rbf}(R_0 \wedge R_1)$. Let $\theta \in \mathcal{L}(\mathrm{rbf}(R_0) \wedge \mathrm{rbf}(R_1))$. By Definition 5.5.1, we must show that for every $x \in \mathrm{free}(R_0 \wedge R_1)$ there exists some $\rho_x \in R_0 \wedge R_1$ such that $x \in \mathrm{free}(\rho_x)$ and either $\theta \models x\dagger_{d(x)}$ or $\theta \models \rho_x$. Hence, suppose that $\theta \notin \mathcal{L}(x\dagger_{d(x)})$, for some variable $x \in \mathrm{free}(R_0 \wedge R_1)$. Since $\mathrm{free}(R_0 \wedge R_1) = \mathrm{free}(R_0) \cup \mathrm{free}(R_1)$ and $\theta \models \mathrm{free}(R_0) \wedge \mathrm{free}(R_1)$, we find from Definition 5.5.1 some $\psi \in R_0 \cup R_1$, with $\theta \models \psi$ and $x \in \mathrm{free}(\psi)$. We now show that there exists an irreducible synchronous set $S \subseteq R_0 \cup R_1$, such that, for $\rho_x := \bigwedge_{\rho \in S} \rho$, we have $\theta \models \rho_x$ and $x \in \mathrm{free}(\rho_x)$. By repeated application of Definition 5.6.1, we construct a finite sequence

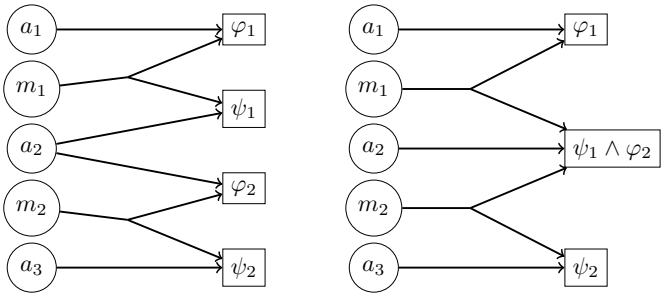
$$\{\psi\} = S_0 \subsetneq \cdots \subsetneq S_n,$$

such that $S_n \subseteq R_0 \cup R_1$ is synchronous, and $\theta \models \bigwedge_{\rho \in S_n} \rho$. Suppose $S_k \subseteq R_0 \cup R_1$, for $k \geq 1$, is not synchronous. By Definition 5.6.1, there exists some $i \in \{0, 1\}$ and a variable $x \in \mathrm{free}(S_k) \cap \mathrm{free}(R_i)$, such that $x \notin \mathrm{free}(S_k \cap R_i)$. Since $x \in \mathrm{free}(R_i)$, we have $R_i^x := \{\rho \in R_i \mid x \in \mathrm{free}(\rho)\} \neq \emptyset$. Since $\theta \models \mathrm{rbf}(R_i)$, there exists some $\psi_k \in R_i^x$ such that $\theta \models \psi_k$. Now define $S_{k+1} := S_k \cup \{\psi_k\}$. Since $x \notin \mathrm{free}(S_k \cap R_i)$ and $x \in \mathrm{free}(S_{k+1} \cap R_i)$, we have a strict inclusion $S_k \subsetneq S_{k+1}$. Due to these strict inclusions, we have, for $k \geq |R_0 \cup R_1|$, that $S_k = R_0 \cup R_1$, which is trivially synchronous in $R_0 \cup R_1$. Therefore, our sequence $S_0 \subsetneq \cdots$ of inclusions terminates, from which we conclude the existence of S_n. By Lemma 5.6.1, we find some irreducible synchronous set $S \subseteq S_n$, such that $\psi \in S$. We conclude that $\rho_x := \bigwedge_{\rho \in S} \rho \in R_0 \wedge R_1$ satisfies $\theta \models \rho_x$ and $x \in \mathrm{free}(\psi) \subseteq \mathrm{free}(S) = \mathrm{free}(\rho_x)$. By Definition 5.5.1, we have $\theta \models \mathrm{rbf}(R_0 \wedge R_1)$, and $\mathrm{rbf}(R_0) \wedge \mathrm{rbf}(R_1) \models \mathrm{rbf}(R_0 \wedge R_1)$.

Finally, we prove that $\mathrm{rbf}(R_0 \wedge R_1) \models \mathrm{rbf}(R_0) \wedge \mathrm{rbf}(R_1)$. Let $\theta \in \mathcal{L}(\mathrm{rbf}(R_0 \wedge R_1))$. We show that $\theta \models \mathrm{rbf}(R_i)$, for all $i \in \{0, 1\}$. By Definition 5.5.1, we must show that for every $i \in \{0, 1\}$ and every $x \in \mathrm{free}(R_i)$ there exists some $\rho \in R_i$ such that $x \in \mathrm{free}(\rho)$ and either $\theta \models x\dagger_{d(x)}$ or $\theta \models \rho$. Hence, let $i \in \{0, 1\}$ and $x \in \mathrm{free}(R_i)$ be arbitrary, and suppose that $\theta \notin \mathcal{L}(x\dagger_{d(x)})$. Since $\mathrm{free}(R_i) \subseteq \mathrm{free}(R_0 \wedge R_1)$, it follows from our assumption $\theta \models \mathrm{rbf}(R_0 \wedge R_1)$ that $\theta \models \bigwedge_{\rho \in S} \rho$, for some irreducible synchronous set $S \subseteq R_0 \cup R_1$ satisfying $x \in \mathrm{free}(S)$. Since $S \subseteq R_0 \cup R_1$ synchronous, we find that $x \in \mathrm{free}(S) \cap \mathrm{free}(R_i) = \mathrm{free}(S \cap R_i)$. Hence, we find some $\rho \in S \cap R_i$, such that $\theta \models \rho$ and $x \in \mathrm{free}(\rho)$. By Definition 5.5.1, we conclude that $\theta \models \mathrm{rbf}(R_i)$, for all $i \in \{0, 1\}$. Therefore, $\mathrm{rbf}(R_0 \wedge R_1) \simeq \mathrm{rbf}(R_0) \wedge \mathrm{rbf}(R_1)$. □

Example 5.6.5. Let φ_i and ψ_i, for $i \geq 1$, be the rules from Example 5.6.1. By Example 5.6.3, the composition $\mathsf{FIFO}_2 := \bigwedge_{i=1}^{2} \mathsf{FIFO}(a_i, a_{i+1}, m_i)$ is defined by the set of rules $\{\varphi_1, \psi_1 \wedge \varphi_2, \psi_2\}$.[2] To compute a set of rules that defines the composition, it is not efficient to enumerate all (exponentially many) synchronous subsets of $R_0 \cup R_1$ and remove all reducible sets. Our tools use an algorithm based on hypergraph transformations to compute the irreducible synchronous sets. Although is would certainly be possible to offer the details of this algorithm here, we postpone the description of such an algorithm until Section 6.3.1. The reason

[2]The rules for the composition of two FIFO stream constraints has striking similarities with *synchronous region* decomposition developed by Proença et al. [PCdVA12]. Indeed, φ_1, $\psi_1 \wedge \varphi_2$, and ψ_2 correspond to the synchronous regions in the composition of two buffers. Therefore, rule-based composition generalizes synchronous region decomposition that has been used as a basis for generation of parallel code [JA18].



(a) $\bigwedge_{i=1}^{2} \mathrm{rbf}(\{\varphi_i, \psi_i\})$ (b) $\mathrm{rbf}(\bigwedge_{i=1}^{2} \{\varphi_i, \psi_i\})$

Figure 5.2: Hypergraph representations of $\bigwedge_{i=1}^{2} \mathsf{FIFO}(a_i, a_{i+1}, m_i)$.

is that the composition operator does not depend on the particular details of the syntax of stream constraints. Indeed, knowing which rules share a variable is the only relevant information for composition. This is precisely the information that is available in multilabeled Petri nets (introduced in Chapter 6), which can be viewed as a data-agnostic abstraction of stream constraints.

Figure 5.2 shows a graphical representation of composition FIFO_2, using hypergraphs. These hypergraphs consist of sets of hyperedges (x, F), where x is a variable and F is a set of rules. Each hyperedge (x, F) in a hypergraph corresponds to a disjunction $x\dagger_{d(x)} \vee \bigvee_{\rho \in F} \rho$ of the rule-based form in Definition 5.5.1. $\Diamond$

5.7 Complexity

In the worst case, composition $R_0 \wedge R_1$ of arbitrary sets of rules R_0 and R_1 may consists of $|R_0| \times |R_1|$ rules. However, if R_0 and R_1 are simple, the size of the composition is bounded by $|R_0| + |R_1|$.

Recall that $P(\phi) = \mathrm{free}^0(\phi)$ is the set of port variables of a stream constraint ϕ.

Definition 5.7.1 (Simple)**.** A set R of rules is simple if and only if $\mathrm{free}(\rho) \cap \mathrm{free}(\rho') \cap P(\mathrm{rbf}(R)) \neq \emptyset$ implies $\rho = \rho'$, for every $\rho, \rho' \in R$.

In other words, a set of rules R is simple if no two (distinct) rules share a port variable. This implies that the dataflow through each port variable is governed by exactly one rule.

Not every stream constraint can be represented by a simple set of rules. For example, a binary exclusive router (Figure 2.5(a) and Example 2.1.7) requires two rules that govern dataflow through its input port A: one rule that routes data from port A to port B and one rule that routes data from port A to port B'. Since both rules share port A, the set of rules is not simple.

Example 5.7.1. By Example 5.5.3, the invariant of $\mathsf{FIFO}(a, b, m)$ is defined by $R := \{a \doteq m' \doteq \mathbf{0} \wedge m \doteq *, m' \doteq * \wedge b \doteq m \doteq \mathbf{0}\}$ as well as $R' := \{a \doteq m' \doteq \mathbf{0} \wedge b \doteq m \doteq *, a \doteq m' \doteq * \wedge b \doteq m \doteq \mathbf{0}\}$. The set R is simple, while R' is not. $\Diamond$

Lemma 5.7.1. *Let R_0 and R_1 be sets of rules, such that* $\mathrm{free}(R_0) \cap \mathrm{free}(R_1) \subseteq P(\mathrm{rbf}(R_0 \cup R_1))$*, and let $S \subseteq R_0 \cup R_1$ be synchronous. Let G_S be a graph with vertices S and edges $E_S = \{(\rho, \rho') \in S^2 \mid \mathrm{free}(\rho) \cap \mathrm{free}(\rho') \cap P(\mathrm{rbf}(R_0 \cup R_1)) \neq \emptyset\}$. If S irreducible, then G_S is connected.*

Proof. Suppose that G_S is disconnected. We find $\emptyset \neq S_0, S_1 \subseteq S$, with $S_0 \cup S_1 = S$, $S_0 \cap S_1 = \emptyset$ and $\mathrm{free}(S_0) \cap \mathrm{free}(S_1) \cap P(\mathrm{rbf}(R_0 \cup R_1)) = \emptyset$. We show that S_0 and S_1 are synchronous. Let $i, j \in \{0, 1\}$ and $x \in \mathrm{free}(S_i) \cap \mathrm{free}(R_j)$. We distinguish two cases:

Case 1 ($x \in \mathrm{free}(R_{1-j})$): Then, $x \in \mathrm{free}(R_0) \cap \mathrm{free}(R_1) \subseteq P(\mathrm{rbf}(R_0 \cup R_1))$. Since $\mathrm{free}(S_0) \cap \mathrm{free}(S_1) \cap P(\mathrm{rbf}(R_0 \cup R_1)) = \emptyset$, we have $x \notin \mathrm{free}(S_{1-i})$. Since S is synchronous, we have $x \in \mathrm{free}(S_i) \cap \mathrm{free}(R_j) \subseteq \mathrm{free}(S) \cap \mathrm{free}(R_j) \subseteq \mathrm{free}(S \cap R_j)$. Hence, we find some $\rho \in S \cap R_j$, with $x \in \mathrm{free}(\rho)$. Since $x \notin \mathrm{free}(S_{1-i})$, we conclude that $\rho \in S_i \cap R_j$. Thus, $x \in \mathrm{free}(S_i \cap R_j)$, if $x \in \mathrm{free}(R_{1-j})$.

Case 2 ($x \notin \mathrm{free}(R_{1-j})$): Since $x \in \mathrm{free}(S_i)$, we find some $\rho \in S_i$, with $x \in \mathrm{free}(\rho)$. Since $x \notin \mathrm{free}(R_{1-j})$, we conclude that $\rho \in R_j$. Hence, $x \in \mathrm{free}(\rho) \subseteq \mathrm{free}(S_i \cap R_j)$, if $x \notin \mathrm{free}(R_{1-j})$.

We conclude in both cases that $x \in \mathrm{free}(\rho) \subseteq \mathrm{free}(S_i \cap R_j)$. Hence, $\mathrm{free}(S_i) \cap \mathrm{free}(R_j) \subseteq \mathrm{free}(S_i \cap R_j)$, for all $i, j \in \{0, 1\}$, and we conclude that S_0 and S_1 are synchronous. Since $S_0 \neq S \neq S_1$, we conclude that S is reducible. By contraposition, we conclude that G_S is connected, whenever S is irreducible. □

Lemma 5.7.2. *Let R_0 and R_1 be simple sets of rules, with* $\mathrm{free}(R_0) \cap \mathrm{free}(R_1) \subseteq P(\mathrm{rbf}(R_0 \cup R_1))$*, and let $S_0, S_1 \subseteq R_0 \cup R_1$ be irreducible synchronous. If $S_0 \cap S_1 \neq \emptyset$, then $S_0 = S_1$.*

Proof. Suppose that $S_0 \cap S_1 \neq \emptyset$. Then, there exists some $\rho_0 \in S_0 \cap S_1$. We show that $S_i \subseteq S_{1-i}$, for all $i \in \{0, 1\}$. Let $i \in \{0, 1\}$, and $\rho \in S_i$. By Lemma 5.7.1, we find an undirected path in G_{S_i} from ρ_0 to ρ. That is, we find a sequence $\rho_0 \rho_1 \cdots \rho_n \in S^*$, such that $\rho_n = \rho$ and $(\rho_i, \rho_{i+1}) \in E_{S_i}$, for all $0 \leq i < n$. We show by induction on $n \geq 0$, that $\rho_n \in S_{1-i}$. For the base case ($n = 0$), observe that $\rho_n = \rho_0 \in S_0 \cap S_1 \subseteq S_{1-i}$. For the induction step, suppose that $\rho_n \in S_{1-i}$. By construction of G_{S_i}, we find that $\mathrm{free}(\rho_n) \cap \mathrm{free}(\rho_{n+1}) \cap P_{01} \neq \emptyset$, where $P_{01} = P(\mathrm{rbf}(R_0 \cup R_1))$. Let $j \in \{0, 1\}$, such that $\rho_{n+1} \in R_j$. Since $\rho_n \in S_{1-i}$ and S_{1-i} is synchronous, we have $\emptyset \neq \mathrm{free}(S_{1-i}) \cap \mathrm{free}(R_j) \cap P_{01} = \mathrm{free}(S_{1-i} \cap R_j) \cap P_{01}$. We find some $\rho' \in S_{1-j} \cap R_j$, with $\mathrm{free}(\rho_{n+1}) \cap \mathrm{free}(\rho') \cap P_{01} \neq \emptyset$. Since R_j is simple, we have $\rho_{n+1} = \rho' \in S_{1-i}$, which concludes the proof by induction. It follows from $\rho_n \in S_{1-i}$ that $S_i \subseteq S_{1-i}$, for all $i \in \{0, 1\}$, that is, $S_0 = S_1$. □

As seen in Lemma 5.5.1, the number of clauses in the disjunctive normal form $\mathrm{dnf}(R_0 \wedge R_1)$ can be exponential in the number of rules $|R_0 \wedge R_1|$ of the composition of R_0 and R_1. However, the following (main) theorem shows the number of rules required to define $\bigwedge_i \phi_i$ is only linear in k.

Theorem 5.7.3. *If R_0 and R_1 are simple sets of rules, and* $\mathrm{free}(R_0) \cap \mathrm{free}(R_1) \subseteq P(\mathrm{rbf}(R_0 \cup R_1))$*, then $R_0 \wedge R_1$ is simple and $|R_0 \wedge R_1| \leq |R_0| + |R_1|$.*

Proof. From Lemmas 5.6.1 and 5.7.2, we find that the irreducible synchronous subsets partition $R_0 \cup R_1$. We conclude that $|R_0 \wedge R_1| \leq |R_0| + |R_1|$. We now show that $R_0 \wedge R_1$ is simple. Let ρ_0 and ρ_1 be rules in $R_0 \wedge R_1$, with $\mathrm{free}(\rho_0) \cap$

$\mathrm{free}(\rho_1) \cap P_{01} \neq \emptyset$, where $P_{01} = P(\mathrm{rbf}(R_0 \cup R_1))$. By Definition 5.6.3, we find, for all $i \in \{0,1\}$, an irreducible synchronous set S_i, such that $\rho_i = \bigwedge_{\psi \in S_i} \psi$. Since $\mathrm{free}(\rho_0) \cap \mathrm{free}(\rho_1) \cap P_{01} \neq \emptyset$ and $\mathrm{free}(\rho_i) = \mathrm{free}(S_i)$, for all $i \in \{0,1\}$, we find some $x \in \mathrm{free}(S_0) \cap \mathrm{free}(S_1) \cap P_{01}$. Suppose that $x \in \mathrm{free}(R_j)$, for some $j \in \{0,1\}$. Since S_0 and S_1 are synchronous sets, we have $x \in \mathrm{free}(S_i) \cap \mathrm{free}(R_j) \subseteq \mathrm{free}(S_i \cap R_j)$, for all $i \in \{0,1\}$. We find, for all $i \in \{0,1\}$, some $\psi_i \in S_i \cap R_j$, such that $x \in \mathrm{free}(\psi_i)$. Hence, $\mathrm{free}(\psi_0) \cap \mathrm{free}(\psi_1) \cap P_{01} \neq \emptyset$, and since R_j is simple, we conclude that $\psi_0 = \psi_1$. Therefore, $S_0 \cap S_1 \neq \emptyset$, and Lemma 5.7.2 shows that $S_0 = S_1$ and $\rho_0 = \rho_1$. We conclude that $R_0 \wedge R_1$ is simple. □

The number of clauses in the disjunctive normal form of direct compositions of k fifo constraints grows exponentially in k. This typical pattern of a sequence of queues manifests itself in many other constructions, which causes serious scalability problems (cf., the benchmarks for 'Alternator$_k$' in [JKA17, Section 7.2]). However, Theorem 5.7.3 shows that rule-based composition of k fifo constraints does not suffer from scalability issues: by Example 5.7.1, the fifo constraint can be defined by a simple set of rules. The result in Theorem 5.7.3, therefore, promises (exponential) improvement over the classical constraint automaton representation.

Unfortunately, it seems impossible to define any arbitrary stream constraint by a simple set of rules. Therefore, the rule-based form may still blow up for certain stream constraints. It seems, however, possible to recognize even more symmetries (cf., the queue-optimization in [JHA14]) to avoid explosion and obtain comparable compilation and execution performance for these stream constraints.

5.8 Abstraction

We now study how existential quantification of stream constraints operates on its defining set of rules.

Definition 5.8.1 (Abstraction)**.** Hiding a variable x in a set of rules R yields $\exists x R := \{\exists x \rho \mid \rho \in R\}$.

Unfortunately, $\exists x R$ does not always define $\exists x \phi$, for a stream constraint ϕ defined by a set of rules R. The following result shows that $\exists x R$ defines $\exists x \phi$ if and only if $\mathrm{rbf}(\exists x R) \models \exists x\, \mathrm{rbf}(R)$. In this case, we call variable x *hidable* in R.

It is non-trivial to find a defining set of rules for $\exists x \phi$, if x is not hidable in R, and we leave this as future work.

Theorem 5.8.1. *Let R be a set of rules, and let $x \in X$ be a variable. Then, $\exists x\, \mathrm{rbf}(R) \simeq \mathrm{rbf}(\exists x R)$ if and only if $\mathrm{rbf}(\exists x R) \models \exists x\, \mathrm{rbf}(R)$.*

Proof. Trivially, $\exists x\, \mathrm{rbf}(R) \simeq \mathrm{rbf}(\exists x R)$ implies $\mathrm{rbf}(\exists x R) \models \exists x\, \mathrm{rbf}(R)$. Conversely, suppose that $\mathrm{rbf}(\exists x R) \models \exists x\, \mathrm{rbf}(R)$. From Lemma 5.5.1, it follows that $\exists x\, \mathrm{rbf}(R) \equiv \exists x\, \mathrm{dnf}(R)$. Since existential quantification distributes over disjunction and $\exists x \phi \wedge \psi \models \exists x \phi \wedge \exists x \psi$, for all stream constraints ϕ and ψ, we find

$$\exists x\, \mathrm{dnf}(R) \models \bigvee_{S \subseteq R} \bigwedge_{\rho \in S} \exists x \rho \wedge \bigwedge_{x \neq y \in \mathrm{free}(R) \setminus \mathrm{free}(S)} y\dagger_{d(y)} \equiv \mathrm{dnf}(\exists x R).$$

By Lemma 5.5.1, we have $\exists x\,\mathrm{rbf}(R) \models \mathrm{rbf}(\exists x R)$, and by assumption $\exists x\,\mathrm{rbf}(R) \equiv \mathrm{rbf}(\exists x R)$. Using Lemma 5.6.2, we have $\deg_y(\exists x\,\mathrm{rbf}(R)) = \max_{\rho \in R} \deg_y(\exists x \rho) = \deg_y(\mathrm{rbf}(\exists x R))$, for every variable y. We conclude $\exists x\,\mathrm{rbf}(R) \simeq \mathrm{rbf}(\exists x R)$. □

Example 5.8.1. Suppose $Data = \{0, 1\}$, which means that the data domain equals $D = \{0, 1, *\}$. Let $\mathbf{1}$ be the constant stream defined as $\mathbf{1}(n) = 1$, for all $n \in \mathbb{N}$. For $i \in \{0, 1\}$, consider the set of rules $R_i = \{x = x, x = y_i = \mathbf{i}\}$. Observe that $\{x = x, x = y_i = \mathbf{i}\} \subseteq R_0 \cup R_1$ is synchronous, for all $i \in \{0, 1\}$. Hence, $x = y_i = \mathbf{i} \in R_0 \wedge R_1$, for all $i \in \{0, 1\}$. However, for $\theta = [y_0 \mapsto \mathbf{0}, y_1 \mapsto \mathbf{1}]$, we have $\theta \models \bigwedge_{i \in \{0,1\}} \exists x (x = y_i = i)$, while $\exists x \bigwedge_{i \in \{0,1\}} x = y_i = i \equiv \bot$. Thus, variable x is not hidable from $R_0 \wedge R_1$. ◊

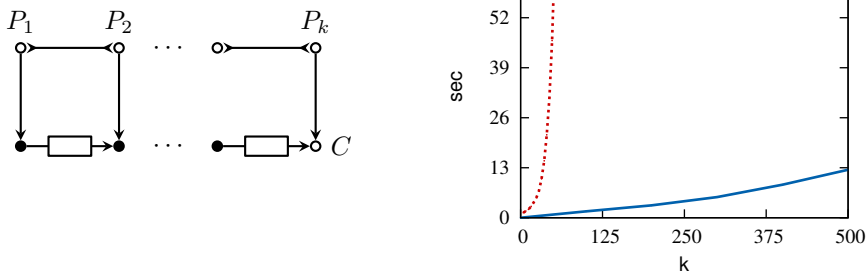
5.9 Application

In on-going work, we applied the rule-based form to compile protocols (in the form of Reo connectors) into executable code. Reo is an exogenous coordination language that models protocols as graph-like structures [Arb04, Arb11]. We recently developed a textual version of Reo, which we use to design non-trivial protocols [DA18b]. An example of such non-trivial protocol is the $\mathsf{Alternator}_k$, where $k \geq 2$ is an integer. Figure 5.3(a) shows a graphical representation of the $\mathsf{Alternator}_k$ protocol.

Intuitively, the behavior of the alternator protocol is as follows: The *nodes* $P_1, \ldots, P_k$ accept data from the environment. Node C offer data to the environment. All other nodes are internal and do not interact with the environment. In the first step of the protocol, the $\mathsf{Alternator}_k$ waits until the environment is ready to offer data at all nodes $P_1, \ldots, P_k$ and is ready to accept data from node C. Only then, the $\mathsf{Alternator}_k$ transfers the data from P_k to C via a synchronous channel, and puts the data from P_i in the i-th fifo channel, for all $i < k$. The behavior of a synchronous channel is defined by the sync stream constraint in Example 5.2.1. Each fifo channel has buffer capacity of one, and its behavior is defined by the fifo stream constraint from Example 5.2.3. In subsequent steps, the environment can one-by-one retrieve the data from the fifo channel buffers, until they are all empty. Then, the protocol cycles back to its initial configuration, and repeats its behavior. For more details on the Reo language and its semantics, we refer to [Arb04, Arb11].

As mentioned in the introduction, Jongmans developed a compiler based on constraint automata [JKA17]. The otherwise stimulating benchmarks presented in [JKA17] show that Jongmans' compiler still suffers from state-space explosion. Figure 5.3(b) shows the compilation time of the $\mathsf{Alternator}_k$ protocol for Jongmans' compiler and ours. Clearly, the compilation time improved drastically and went from exponential in k to almost linear in k.

Every fifo channel in the $\mathsf{Alternator}_k$, except the first, either accepts data from the environment or accepts data from the previous fifo channel. This choice is made by the internal node at the input of each fifo channel. Unfortunately, the behavior of such nodes is not defined in terms of a simple set of rules. Consequently, we cannot readily apply Theorem 5.7.3 to conclude that the number of rules depends only linearly on k. However, it turns out that $\mathsf{Alternator}_k$ can be defined using only k rules: one rule for filling the buffers of all fifo channels, plus $k - 1$ rules, one for



(a) $\mathsf{Alternator}_k$ (b) Compilation times

Figure 5.3: Graphical representation (a) of the $\mathsf{Alternator}_k$ protocol in [JKA17], for $2 \leq k \leq 500$, and its compilation time (b). The dotted red line is produced by the Jongmans' compiler (and corresponds to [JKA17, Fig 11(a)]), and the solid blue line is our compiler.

taking data out of the buffer of each of the $k - 1$ fifo channels. This observation explains why our compiler drastically improves upon Jongmans' compiler.

5.10 Discussion

We introduce (regular) stream constraints as an alternative to constraint automata that does not suffer from state space explosions. We define the rule-based form for stream constraints, and we express composition and abstraction of constraints in terms of their rule-based forms. For simple sets of rules, composition of rule-based forms does not suffer from 'transition space explosions' either.

We have experimented with a new compiler for protocols using our rule-based form, which avoids the scalability problems of state- and transition-space explosions of previous automata-based tools. Our approach still leaves the possibility for transition space explosion for non-simple sets of rules. In the future, we intend to study symmetries in stream constraints that are not defined by simple sets of rules. The queue-optimization of Jongmans serves as a good source of inspiration for exploiting symmetries [JHA14].

The results in this chapter are purely theoretical. In on-going work, we show practical implications of our results by developing a compiler based on stream constraints. Such a compiler requires an extension to the current theory on stream constraints: we did not compute the abstraction $\exists x R$ on sets of rules R wherein variable x is not hidable. Example 5.5.4 indicates the existence of situations where we can compute $\exists x R$ even if x is not hidable, a topic which we leave as future work.

Chapter 6

Protocols as Petri nets

Our constraint-based approach to the specification of interaction protocols is very flexible, because many syntactically different constraints have the same meaning. While such flexibility is useful for development of new compilation techniques, it distracts attention from the fundamental principle that leads to the exponential improvement of compilation in Figure 5.3. The most important concept in Chapter 5 is that rule-based stream constraints expose concurrency in constraints that define sequential behavior.

In the current chapter, we streamline this concept by expressing protocols in terms of an enhanced version Petri nets called multilabeled Petri nets[1]. The explicit concurrency in Petri nets helps us to develop alternative compilers that respect the available concurrency in a protocol.

Although we desire the protocol specification as an orchestration, we want a protocol implementation as a choreography. Indeed, implementing the protocol with a single central protocol component can easily introduce a performance bottleneck. To prevent the bottleneck, we aim for distributed protocols, i.e., protocols implemented as multiple parallel components.

It is the responsibility of the compiler of the coordination language to produce efficient protocol implementations. Jongmans [Jon16] developed a compiler that generates protocol implementations based on constraint automata [BSAR06]. A constraint automaton is a pair (P, A) consisting of a set of variables P and a state machine A, whose transition labels are pairs (N, g) consisting of a set of variables $N \subseteq P$ and a constraint g on their values. The elements P, N, and g are respectively called interface, synchronization constraint, and data constraint.

Constraint automata model protocols in a simple and intuitive manner. However, being a state machine, a constraint automaton is inherently sequential. As we desire distributed protocols, the constraint automaton representation of protocols is not completely adequate. State-space explosions for constraint automata serve as evidence for this mismatch. Alternative algorithms to compute the composite constraint automaton have only partial success [JKA17, Fig. 17(a)].

In contrast to state machines, Petri nets [Rei13] are inherently parallel. Moreover, a state machine can be viewed as a Petri net for which every transition has

[1] The work in this chapter is based on [Dok19]

a single input place and a single output place. It seems natural to generalize constraint automata by replacing the underlying state machine by a Petri net.

In the current chapter, we introduce multilabeled Petri nets as an inherently parallel extension to constraint automata. After stating some basic results on monoids and multisets (Section 6.1), we view a constraint automaton as a *multilabeled Petri net* (Section 6.2), which is an ordinary Petri net whose transitions are labeled by multisets of actions. If multiple (not necessarily distinct) transitions in a multilabeled Petri net fire in parallel, the composite transition is labeled by the union of the labels of its constituent transitions.

We generalize constraint automaton composition to composition of multilabeled Petri nets (Section 6.3). While the composition of arbitrary multilabeled Petri nets seems hard to compute, we develop an efficient algorithm that composes *square-free* nets. Intuitively, a multilabeled Petri net is squarefree iff any action occurs at most once in every (parallel) execution step of the net.

The number of places in the composite Petri net grows linearly, which prevents the state-space explosion. Therefore, multilabeled Petri nets are an adequate intermediate representation of protocols. Since a transition-space explosion is still possible, multilabeled Petri nets are not a silver bullet.

Multilabeled Petri nets can contain silent transitions, which have no observable behavior. Such silent transitions can be the result of hiding irrelevant actions. In protocol implementations based on multilabeled Petri nets, silent transitions do not perform any I/O-operation and delay the throughput of the protocol. We define an abstraction operator for multilabeled nets (Section 6.4) that removes silent transitions. We develop an algorithm that computes the abstraction of a multilabeled net.

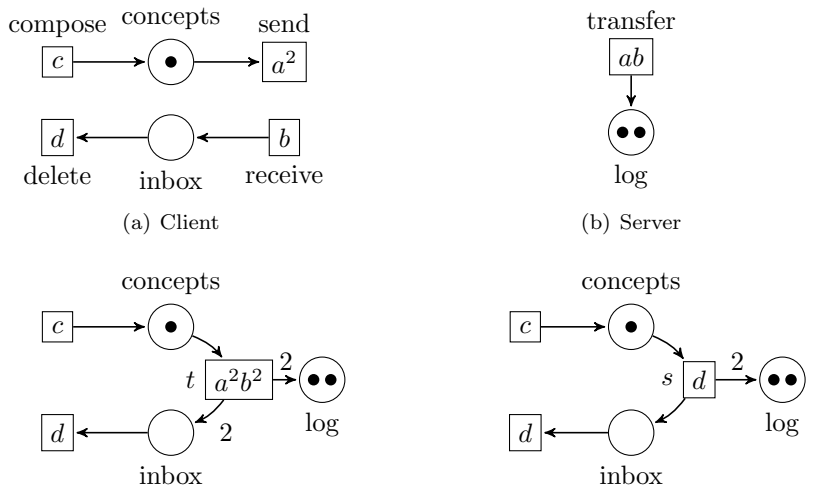
Finally, we summarize the results (Section 6.5) and point out future work.

Running example We illustrate the composition and abstraction of multilabeled Petri nets by an example on a mail server and a client.

Figure 6.1(a) shows the Petri net of a client that can compose, send, receive, and delete messages. Composed messages are stored as concepts, and received messages end up in the inbox. The client can concurrently send and receive messages, as is the case for a large company with internal mail between different departments. We want every message to be transferred to two recipients (e.g., adding a recipient in CC). We represent this intend labeling the send transition with the expression a^2, which denotes a multiset that contains a twice.

Figure 6.1(b) shows the Petri net of a server that can transfer messages. A fingerprint of each transferred message is logged. We assume that message transferal is not buffered: a send message is immediately received. We represent this by labeling the transfer transition with the expression ab that denotes the multiset that contains a and b. Note that the order of a and b is irrelevant, as ab and ba denote the same multiset.

Figure 6.1(c) shows the composition of the client and server, as defined by the composition operator presented in the current chapter. Composition of multilabeled nets synchronize transitions that agree on shared actions. Observe that the transfer transition must fire twice in order to agree on a with the send transition. Consequently, the receive transition must fire twice in order to agree on b with the



(a) Client (b) Server

(c) Composition (d) Abstraction

Figure 6.1: Multilabeled nets of a client (a) and a mail server (b). Their composition (c) synchronizes a single send transition with two transfer and two receive transitions (t equals send | transfer^2 | receive^2). The abstraction (d) hides the a and b actions and eliminates the resulting silent transition (s equals t; delete).

transfer^2 transition. Hence, the send, transfer, and receive transitions synchronize into a single parallel transition send | transfer^2 | receive^2, which we denote as t. The actions c (compose) and d (delete) of the client are not shared with the server. Therefore, the client-server composition allows the client to compose and delete a message, without synchronizing with the mail server.

Figure 6.1(d) shows the abstraction of the composite system, where message transferal (actions a and b) is hidden. After hiding a and b, transition t becomes silent. We eliminate this silent transition by *sequentially* composing t with the (observable) delete transition. In other words, we replace transition t by the sequential composition, s, of t followed by delete (i.e., $s = t;\mathrm{delete}$). Note that s deletes only one of the two messages that are sent. The remaining message can be deleted by the usual delete transition.

Related work The literature offers a wide variety of composition operators on Petri nets, which can be divided in two categories, namely *topological* and *parallel* compositions.

By topological compositions, we mean those operators that 'glue Petri nets along their places and transitions'. For example, Mazurkiewicz [Maz95, Section 8] defines a composition operator that glues transitions with the same name. Bernardinello and De Cindio [BdC92, Part II] define a composition that glues Petri nets on places and transitions. Kotov [Kot78] defines multiple composition operators, including superposition, which glue two Petri nets along shared places and transitions. Hierarchical composition [Feh91] is a topological composition that identifies a complete Petri net with a single transition of another Petri net.

By parallel compositions, we mean those operators that 'synchronize transitions of the constituent Petri nets'. A single transition of one Petri net can synchronize with multiple transitions of the other net. Hence, parallel composition can be seen as duplication plus gluing, which distinguishes it from topological composition.

Parallel composition is called synchronous, if subsystems must 'run at the same speed'. That is, both Petri nets in a composition must simultaneously fire a transition. Examples of these operators are those that are defined as a product in a category of Petri nets [vGV87, Win87, MM90, Gol88]. Such composition operators are not always convenient for specification.

In asynchronous parallel composition, subsystems can run at different speeds. That is, one Petri net can fire a transition, while the other net does nothing. Examples of asynchronous compositions include the composition of Petri Boxes [BDH92], Signal Transition Graphs [VW02], extended safe nets [Tau89, Chapter 4], zero safe nets [BM97], and general Petri nets [Gol88].

The asynchronous composition operator defined by Anisimov, Golenkov, and Kharitonov [AGK01] comes closest to our composition operator. Their parallel composition ${}_\alpha\|_\beta$ of Petri nets is relative to some transition (multi)labelings α and β of its operands. Unlike our composition operator, Anisimov et al. suppose a CCS-like synchronization that synchronizes an action a with its conjugate $\overline{a}$. As such, their composition cannot be used as a formal semantics for Reo circuits.

Silent transitions in safe Petri nets can be removed in at least two different ways. Vogler and Wollowski [VW02] use contraction, which deletes a silent transition and merges all its input and output places. In general, contraction does not yield a safe net.

Wimmel [Wim04] eliminates silent transitions from safe Petri nets in three steps: First, he unfolds a safe Petri net into an occurrence net, whose graph structure is acyclic. Next, he finds the process [Pra86] (i.e., the set of all accepted pomsets) of the occurrence net, while ignoring all silent transitions. Finally, he constructs a suitable finite quotient of the process. By construction, the resulting Petri net has no silent transitions and is pomset-equivalent to the original net.

6.1 Preliminaries

6.1.1 Graded monoids

A **monoid** is a triple $(M, +, 0)$, where $+ : M \times M \to M$ is an associative binary operation, with identity element 0. A **submonoid** of $(M, +, 0)$ is a monoid of the form $(S, +, 0)$, where $S \subseteq M$ is a subset of M that contains $0 \in S$ and is closed under addition.

An element x in a monoid M is **invertible** iff there exists some $y \in M$, such that $x + y = 0$. An element x in a monoid M is **irreducible** iff $x = a + b$ implies that a or b is invertible, for all $a, b \in M$.

A monoid $(M, +, 0)$ is **graded** if there exists a map $g : M \to \mathbb{N}$, such that, for all $x, y \in M$, we have $g(x + y) = g(x) + g(y)$ and if $g(x) = 0$ then x is invertible.

Lemma 6.1.1. *Every element in a graded monoid is a sum of irreducibles.*

Proof. Let $x \in M$ be arbitrary. We show by induction on the grading $g(x) \in \mathbb{N}$ that x is a sum of irreducibles.

If $g(x) = 0$, then x is invertible. Hence, $x + y = 0$, for some $y \in M$. If $x = a + b$, for some $a, b \in M$, then $a + (b + y) = 0$ and a is invertible. Thus, x is irreducibile. In particular, x is a sum of irreducibles.

Suppose that every $y \in M$, with $g(y) < g(x)$, is a sum of irreducibles. If x is irreducible, then x is a sum of irreducibles. Suppose that x is reducible, i.e., $x = a + b$, for some non-invertible $a, b \in M$. By definition of the grading, $g(a), g(b) > 0$. Hence, $g(a), g(b) < g(a) + g(b) = g(x)$. By the induction hypothesis, a and b are sums of irreducibles, and so is x.

By induction, every element in a graded monoid is a sum of irreducibles. □

A **right-ideal** of a monoid $(M, +, 0)$ is a subset $I \subseteq M$, such that $x \in I$ and $y \in M$ implies $x + y \in I$, for all $x, y \in M$. A right-ideal I of a monoid M is **proper** iff $0 \notin I$. An element $x \in I$ is **right-irreducible (in I)** iff $x = a + b$ and $a \in I$ implies that b is invertible, for all $a, b \in M$.

Lemma 6.1.2. *Let I be a proper right ideal of a graded monoid M. Every element in M is a (possibly empty) sum of right-irreducibles in I plus an element $r \in M \setminus I$.*

Proof. Since the lemma holds trivially for right-irreducibles in I, let $x \in M$ be right-reducible. We show by induction on the grading $g(x) \in \mathbb{N}$ that x is a possibly empty sum of right-irreducibles in I plus an element $r \in M \setminus I$.

If $g(x) = 0$, then x is invertible. Since I is proper, we have $x \in M \setminus I$. Hence, x is an empty sum of right-irreducibles plus $r = x$.

Suppose that every y, with $g(y) < g(x)$, is a possibly empty sum of right-irreducibles in I plus an element $r \in M \setminus I$. Since x is right-reducible, we find some $a \in I$ and some non-invertible b, such that $x = a + b$. Non-invertibility of b implies that $g(b) > 0$. Hence,

$$g(a) < g(a) + g(b) = g(a + b) = g(x).$$

The induction hypothesis shows that $a = (\sum_{i=1}^{n} a_i) + r$ is a sum of $n \geq 0$ right-irreducibles $a_1, \ldots, a_n$ in I plus an element $r \in M \setminus I$. Since $a \in I$ and $r \in M \setminus I$, we have $n \geq 1$. Thus, $\sum_{i=1}^{n} a_i$ is an element of the right-ideal I. As I is proper, $\sum_{i=1}^{n} a_i$ is non-invertible, and $g(\sum_{i=1}^{n} a_i) \geq 1$. This implies that

$$g(r + b) < g(\textstyle\sum_{i=1}^{n} a_i) + g(r + b) = g((\sum_{i=1}^{n} a_i) + r + b) = g(x).$$

The induction hypothesis applied to $r + b$ yields right-irreducibles $b_1, \ldots, b_m$ and an element $r' \in M \setminus I$, such that $r + b = (\sum_{j=1}^{m} b_j) + r'$. Hence, $x = a + b = (\sum_{i=1}^{n} a_i) + r + b = (\sum_{i=1}^{n} a_i) + (\sum_{j=1}^{m} b_j) + r'$, which proves the lemma. □

6.1.2 Multisets

A **multiset** over a set X is an unordered collection of elements with duplicates, which is formally represented as a map $m : X \to \mathbb{N}$ that counts the number of occurrences of each $x \in X$ in the multiset. The set of all multisets over X is denoted as $\mathbb{N}^X$. The **cardinality** $|m|$ of a multiset m is defined as the cardinality of the set $\{(x, k) \mid x \in X, 0 \leq k < m(x)\}$. A multiset m is **non-empty** iff $0 < |m|$, and **finite** iff $|m| < \aleph_0$, where $\aleph_0$ is the first infinite cardinal number. The empty multiset is denoted as $\emptyset$. The set of all finite multisets over X is denoted as

$\mathbb{N}^{(X)} = \{m : X \to \mathbb{N} \mid |m| < \aleph_0\}$. The **free commutative monoid** is the triple $(\mathbb{N}^{(X)}, \cup, \emptyset)$.

For $k \in \mathbb{N}$, and multisets $m, m' \in \mathbb{N}^X$, the **union** $m \cup m'$, **intersection** $m \cap m'$, **difference** $m \setminus m'$, and **multiplication** km are defined, for $x \in X$, as

$$\begin{aligned}(m \cup m')(x) &= m(x) + m'(x),\\ (m \cap m')(x) &= \min(m(x), m'(x)),\\ (m \setminus m')(x) &= m(x) \dot{-} m'(x)\\ (k \cdot m)(x) &= k \cdot m(x),\end{aligned}$$

where $\dot{-}$ is monus, defined as $a \dot{-} b = \max(a - b, 0)$, for all $a, b \in \mathbb{N}$. The **subset** relation of multisets is defined as $m \subseteq m'$ iff $m(x) \leq m'(x)$, for all $x \in X$.

Multisets $m_0, m_1, m_2 \in \mathbb{N}^X$ satisfy the following identities:

$$\begin{aligned}m_0 \setminus (m_1 \cup m_2) &= (m_0 \setminus m_1) \setminus m_2\\ m_0 \cup (m_1 \setminus m_0) &= m_1 \cup (m_0 \setminus m_1)\\ (m_0 \cup m_1) \setminus m_2 &= (m_0 \setminus m_2) \cup (m_1 \setminus (m_2 \setminus m_0))\end{aligned}$$

Restriction $m|_Y$ of a multiset m on X to a subset $Y \subseteq X$ is defined, for all $y \in Y$, as $m|_Y(y) = m(y)$.

It is convenient to represent a finite multiset over a set X as (an equivalence class of) a finite sequence of elements from X. Let X^* be the free monoid on X that consists of all finite words of elements from X (including the empty word ϵ). A word $w \in X^*$ induces a multiset $w : X \to \mathbb{N}$, by defining, for all $x \in X$,

$$\epsilon(x) = 0, \quad wx(x) = w(x) + 1, \qquad wy(x) = w(x), \text{ for } y \neq x.$$

Note that different words might define the same multiset. For example, xy and yx both denote the multiset wherein both x and y occur once.

6.2 Multilabeled Petri nets

Multilabeled Petri nets are Petri nets whose transitions are labeled with a multiset of actions.

Definition 6.2.1. A **multilabeled (Petri) net** is a tuple (A, P, T, μ_0) with

1. A a set of actions,
2. P a set of places,
3. $T \subseteq \mathbb{N}^P \times \mathbb{N}^A \times \mathbb{N}^P$ a set of transitions, and
4. $\mu_0 : P \to \mathbb{N}$ an initial marking.

Inspired by Goltz [Gol88], the notation in Definition 6.2.1 slightly differs from the usual notation of Petri nets. The advantage of this presentation is that transitions (including its set of input and output places) can be studied in isolation, which allows for parallel and sequential composition of transitions.

For a transition $t = (P, \alpha, Q) \in T$, we write ${}^\bullet t = P$ for the multiset of input places of t, we write $t^\bullet = Q$ for the multiset of output places of t, and we write $\ell(t) = \alpha$ for the multiset of labels of t.

For the development of the composition operator for multilabeled nets in Section 6.3, we use the standard concurrent semantics of Petri nets, which allows multiple transitions to fire in parallel.

Definition 6.2.2. A **multitransition** of a multilabeled net (A, P, T, μ_0) is a finite multiset $\theta : T \to \mathbb{N}$ of transitions. $\mathbb{N}^{(T)}$ denotes the set of all multitransitions.

A multitransition $\theta \in \mathbb{N}^{(T)}$ of a multilabeled net $N = (A, P, T, \mu_0)$ defines a (concrete) transition $\tau(\theta) = ({}^\bullet\theta, \ell(\theta), \theta^\bullet)$ in $\mathbb{N}^P \times \mathbb{N}^A \times \mathbb{N}^P$, wherein

$$
\begin{aligned}
{}^\bullet\theta &= \textstyle\bigcup_{t \in T} \theta(t) \cdot {}^\bullet t \\
\theta^\bullet &= \textstyle\bigcup_{t \in T} \theta(t) \cdot t^\bullet \\
\ell(\theta) &= \textstyle\bigcup_{t \in T} \theta(t) \cdot \ell(t)
\end{aligned}
$$

A multitransition θ of a N is **enabled** at a marking $\mu \in \mathbb{N}^P$ iff ${}^\bullet\theta \subseteq \mu$. A marking $\mu' \in \mathbb{N}^P$ is **obtained** from a marking $\mu \in \mathbb{N}^P$ via a multitransition θ (denoted $\mu \,[\theta\rangle\, \mu'$) iff θ is enabled at μ, and $\mu' = (\mu \setminus {}^\bullet\theta) \cup \theta^\bullet$.

Definition 6.2.3. The **concurrent semantics** of a multilabeled net (A, P, T, μ_0) is a pointed, directed, labeled graph (V, E, μ_0), consisting of

1. vertices $V = \{\mu : P \to \mathbb{N}\}$, and
2. labeled edges $E = \{(\mu, \ell(\theta), \mu') \in V \times \mathbb{N}^A \times V \mid \mu \,[\theta\rangle\, \mu', |\theta| > 0\}$.

Note that the empty multitransition $\theta = \emptyset$ does not constitute a valid step in the semantics, as $\emptyset$ allows for internal divergent behavior (by always firing $\emptyset$).

For the development of the abstraction operator in Section 6.4, we rely on the interleaving semantics of nets:

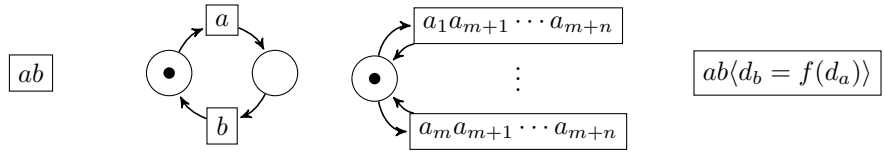
Definition 6.2.4. The **interleaving semantics** of a multilabeled net (A, P, T, μ_0) is a pointed, directed, labeled graph (V, E, μ_0), consisting of

1. vertices $V = \{\mu : P \to \mathbb{N}\}$, and
2. labeled edges $E = \{(\mu, \ell(t), \mu') \in V \times \mathbb{N}^A \times V \mid \mu \,[t\rangle\, \mu'\}$.

The only difference between the concurrent semantics and interleaving semantics of multilabeled nets is the cardinality of the multitransitions.

We introduce some terminology for a multilabeled net $N = (A, P, T, \mu_0)$. N is called **finite** iff A, P, and T are all finite. For $k \geq 1$, N is called **k-bounded** iff every reachable marking μ satisfies $\mu(p) \leq k$, for all $p \in P$. N is called **safe** iff N is 1-bounded.

A marking $\mu' \in \mathbb{N}^P$ is reachable from a marking $\mu \in \mathbb{N}^P$ via a sequence of transitions $t_1 \cdots t_n \in T^*$, with $n \geq 0$, (denoted $\mu \,[t_1 \cdots t_n\rangle\, \mu'$) iff there exists markings $\mu_1, \ldots, \mu_{n-1} \in \mathbb{N}^P$, such that $\mu \,[t_1\rangle\, \mu_1 \cdots \mu_{n-1} \,[t_n\rangle\, \mu'$. A **firing sequence** of N is a sequence of transitions $t_1 \cdots t_n \in T^*$, with $n \geq 0$ and $\mu_0 \,[t_1 \cdots t_n\rangle\, \mu'$, for some marking μ'. A marking μ is called **reachable** iff μ is reachable from the initial marking μ_0.



(a) $\mathsf{sync}(a,b)$ (b) $\mathsf{fifo}_1(a,b)$ (c) $\mathsf{node}_{m,n}(a_1,\ldots,a_{m+n})$ (d) $\mathsf{transformer}_f(a,b)$

Figure 6.2: Multilabeled nets for Reo primitives. The action $\langle d_b = f(d_a)\rangle$ in the transformer is the encoding of a data constraint.

A transition t is **dead** in N iff t does not occur on any firing sequence. A transition t is **potentially fireable** in N iff t is not dead in N. For $k \geq 1$, a transition $t \in T$ in a multilabeled net N is k**-live** iff ${}^\bullet t(p) \leq k$, for all $p \in P$. Every potentially fireable transition in a k-bounded net is k-live.

6.2.1 Constraint automata

As stated earlier, multilabeled nets generalize constraint automata [BSAR06] (without data constraints). If data constraints are ignored, the interpretation of constraint automata as multilabeled nets is rather straightforward. Figure 6.2 shows the multilabeled nets for some frequently used Reo primitives.

The $\mathsf{sync}(a,b)$ protocol in Figure 6.2(a) accepts a datum at input a and immediately offers it at output b. Since the $\mathsf{sync}(a,b)$ protocol is stateless, its multilabeled net does not contain a place. The $\mathsf{fifo}_1(a,b)$ protocol in Figure 6.2(b) accepts a datum at input a and stores it. In the next step, it offers the stored datum at output b. The $\mathsf{node}_{m,n}(a,\ldots,a_{m+n})$ protocol in Figure 6.2(c) accepts a datum at any input a_i, with $1 \leq i \leq m$, and immediately offers a copy of it at every output a_j, with $m < j \leq m+n$. The place in the $\mathsf{node}_{m,n}(a,\ldots,a_{m+n})$ protocol does not serve as memory, but encodes the conflict between the transitions. The $\mathsf{transformer}_f(a,b)$ protocol accepts a datum d_a from its input a, and simultaneously offers the datum $f(d_a)$ at its output b. The transformation of datum d_a into d_b is modeled by the data constraint $d_b = f(d_a)$.

In general, constraint automata can have non-trivial data constraints, as is the case for $\mathsf{transformer}_f(a,b)$. It is certainly possible to extend the definition of multilabeled nets (Definition 6.2.1) to include data constraints as well. However, such extension would not add any expressiveness to multilabeled nets, because data constraints can be encoded as fresh actions. For example, the data constraint $d_b = f(d_a)$ can be encoded as a fresh action $\langle d_b = f(d_a)\rangle$, which we call a **data-constraint action**. Freshness ensures that data-constraint actions are not used for synchronization (as is defined in Section 6.3). After composition, data-constraint actions can be decoded back into data constraints. If multiple data-constraint actions end up in the same transition label, their decoded data constraints are combined via conjunction. Figure 6.2(d) shows the multilabeled net for the transformer channel.

The above trick to encode data constraints as actions can be similarly applied to other semantic models [JA12].

The multilabeled nets that come from constraint automata are square-free:

Definition 6.2.5. A multitransition θ in a multilabeled net N is **square-free** iff $\ell(\theta)(a) \leq 1$, for all $a \in A(N)$. A multilabeled net N is **square-free** iff every potentially-fireable multitransition θ in N is square-free.

It is easy to verify that all multilabeled nets in Figure 6.2 are square-free: every transition is square-free and every multitransition θ of size $|\theta| > 1$ is dead.

The main reason for considering square-free nets is that their composition can be easily computed.

6.3 Composition

For this section, fix two multilabeled nets $N_i = (A_i, P_i, T_i, \mu_{0i})$, for $i \in \{0, 1\}$, which are interpreted according to the concurrent semantics in Definition 6.2.3.

We define the composition $N_0 \times N_1$ of N_0 and N_1 that synchronizes N_0 and N_1 on shared actions $A_0 \cap A_1$. We follow a standard approach to define the (parallel) composition of multilabeled nets. First, we generate all combinations of transitions in N_0 and N_1 that can fire in parallel. Next, we restrict to synchronizing combinations that 'agree on shared actions'. Finally, we restrict to a subset of combinations that generate all synchronizing combinations.

Recall that the disjoint union $X + Y$ of two sets X and Y is defined as $(X \times \{0\}) \cup (Y \times \{1\})$.

Definition 6.3.1. A **global transition** is a finite multiset $\eta : T_0 + T_1 \to \mathbb{N}$.

A global transition $\eta \in \mathbb{N}^{(T_0+T_1)}$ has, for $i \in \{0, 1\}$, a **local component** $\eta|_i \in \mathbb{N}^{(T_i)}$ defined as $\eta|_i(t) = \eta(t, i)$, for all $t \in T_i$.

The set $\mathbb{N}^{(T_0+T_1)}$ of all global transitions, endowed with the union $\cup$ of multiset and the empty multiset $\emptyset$, constitutes a monoid. Moreover, multiset cardinality, $|\cdot|$, defines a grading on $\mathbb{N}^{(T_0+T_1)}$. In particular, any submonoid of $\mathbb{N}^{(T_0+T_1)}$ is a graded monoid.

We now formalize the notion of agreement on shared actions:

Definition 6.3.2. A global transition $\eta \in \mathbb{N}^{(T_0+T_1)}$ is **S-compatible**, $S \subseteq A_0 \cap A_1$, iff $\ell(\eta|_0)(a) = \ell(\eta|_1)(a)$, for all $a \in S$. The set of S-compatible global transitions is denoted as $C(S) \subseteq \mathbb{N}^{(T_0+T_1)}$. We call η **compatible**, if η is $A_0 \cap A_1$-compatible.

Intuitively, the local components $\eta|_0$ and $\eta|_1$ of an S-compatible multitransition $\eta \in \mathbb{N}^{(T_0+T_1)}$ agree only on the shared actions in S, while they may disagree on shared actions $a \in (A_0 \cap A_1) \setminus S$ outside of S.

A compatible global transition $\eta \in C(A_0 \cap A_1)$ defines a (concrete) transition $\lambda(\eta) = ({}^\bullet\eta, \ell(\eta), \eta^\bullet) \in \mathbb{N}^{P_0+P_1} \times \mathbb{N}^{A_0 \cup A_1} \times \mathbb{N}^{P_0+P_1}$, with

$$\begin{aligned} {}^\bullet\eta(p, i) &= {}^\bullet(\eta|_i)(p) \\ \eta^\bullet(p, i) &= (\eta|_i)^\bullet(p) \\ \ell(\eta)(a) &= \ell(\eta|_i)(a) \qquad \text{if } a \in A_i \end{aligned}$$

for all $(t, i) \in T_0 + T_1$ and $a \in A_0 \cup A_1$. Note that $\ell(\eta)(a)$ is well-defined, since $\ell(\eta|_0)(a) = \ell(\eta|_1)(a)$, for $a \in A_0 \cap A_1$.

The empty global transition $\emptyset$ is trivially S-compatible, for all $S \subseteq A_0 \cap A_1$. Furthermore, the union $\alpha \cup \beta$ of two compatible global transitions is again S-compatible, for all $S \subseteq A_0 \cap A_1$. Hence, the set of S-compatibles $C(S)$ is a (graded) submonoid of the graded monoid $\mathbb{N}^{(T_0+T_1)}$. Lemma 6.1.1 shows that every compatible global transition can be decomposed into irreducibles.

Any reducible S-compatible global transition $\eta = \alpha \cup \beta$, for some global transitions α and β, is redundant, as $\lambda(\eta)$ can be simulated by firing $\lambda(\alpha)$ and $\lambda(\beta)$ in parallel. Hence, we consider the set $C_0(S) \subseteq C(S)$ of all irreducible S-compatible global transitions.

The image of $C_0(A_0 \cap A_1)$ under the map $\lambda : C(A_0 \cap A_1) \to \mathbb{N}^{P_0+P_1} \times \mathbb{N}^{A_0 \cup A_1} \times \mathbb{N}^{P_0+P_1}$ defines the set of transitions of the composition:

Definition 6.3.3. The **composition** $N_0 \times N_1$ of the multilabeled nets N_0 and N_1 is a multilabeled net with actions $A_0 \cup A_1$, places $P_0 + P_1$, transitions $\lambda(C_0(A_0 \cap A_1))$, and initial marking μ_0 defined as $\mu_0(p, i) = \mu_{0i}(p)$, for all $(p, i) \in P_0 + P_1$.

It is laborious but straightforward to verify that the composition of multilabeled Petri nets is associative. The composition is commutative only up to renaming of places, due to the index from the disjoint union. The composition is idempotent only up to semantic equivalence, as is duplicates the places.

Example 6.3.1. Consider the mail example in Figure 6.1, and suppose that the set of actions of the client equals $\{a, b, c, d\}$ and the set of actions of the server equals $\{a, b\}$. According to Definition 6.3.3, the transitions in the composition of the nets in Figures 6.1(a) and 6.1(b) consist of irreducible compatible global transitions. The compose transition in Figure 6.1(a) and the (implicit) empty multitransition $\emptyset$ in Figure 6.1(b) trivially agree on shared actions. Therefore, (compose $\mid \emptyset$) is a compatible transition. Being of length 1, the compose transition is necessarily irreducible, which implies that the compose transition is a transition of the composition in Figure 6.1(c).

Similarly, the global transition η, with components $\eta|_0 = \text{send} \mid \text{receive}^2$ and $\eta|_1 = \text{transfer}^2$, is also a compatible transition. Indeed, the label of $\eta|_0$ and $\eta|_1$ both equal a^2b^2. Clearly, η is irreducible, which shows that $\lambda(\eta)$ is a transition of the composition in Figure 6.1(c). ◊

6.3.1 Composition algorithm

Definition 6.3.3 only defines the transitions of the composition: it does not suggest a procedure on how these transitions can be found. It seems difficult to compute the composition of arbitrary multilabeled nets. Since the current work is motivated by constraint automata, we develop an algorithm that computes the composition of square-free multilabeled nets (Definition 6.2.5).

Lemma 6.3.1 shows that square-freeness must be checked only for atomic nets.

Lemma 6.3.1. *If N_0 and N_1 are square-free, then so is $N_0 \times N_1$.*

Proof. If a global transition $\eta \in \mathbb{N}^{(T_0+T_1)}$ is potentially fireable, then so are its local components $\eta|_0$ and $\eta|_1$. For $i \in \{0, 1\}$, square-freeness of N_i implies that $\ell(\eta|_i)(a) \leq 1$ and $a \in A_i$. By construction, $\ell(\eta)(a) \leq 1$, for all $a \in A_0 \cup A_1$. Hence, $N_0 \times N_1$ is square-free. □

By Definition 6.3.3, the composition $N_0 \times N_1$ can contain dead transitions. Since these dead transitions do not contribute to the behavior of the multilabeled net, it is no problem if our composition algorithm does not generate them. As the composition, $N_0 \times N_1$, is square-free, we consider only square-free transitions:

Definition 6.3.4. A global transition $\eta : T_0 + T_1 \to \mathbb{N}$ is **square-free** if $\lambda(\eta)$ is square-free. For $S \subseteq A_0 \cap A_1$, we denote the set of all square-free, irreducible S-compatible global transitions as $\overline{C_0}(S) \subseteq C_0(S)$.

We compute the composition of square-free nets N_0 and N_1 by recursion on the number of shared actions. This procedure is conveniently expressed with the following terminology:

Definition 6.3.5. The **difference** $d_a(\eta)$ of a global transition $\eta \in \mathbb{N}^{(T_0+T_1)}$ at a shared action $a \in A_0 \cap A_1$ is defined as the integer

$$d_a(\eta) = \ell(\eta|_0)(a) - \ell(\eta|_1)(a).$$

The set of all square-free, irreducible, S-compatible global transitions with difference $d \in \mathbb{Z}$ is denoted as $\overline{C_0^d}(S)$.

It is straightforward to verify that $d_a(\alpha \cup \beta) = d_a(\alpha) + d_a(\beta)$, for global transitions $\alpha, \beta \in \mathbb{N}^{(T_0+T_1)}$.

Since every global transition is $\emptyset$-composite, $C(\emptyset) = \mathbb{N}^{(T_0+T_1)}$ and $C_0(\emptyset) = T_0 + T_1$, where each $(t, i) \in T_0 + T_1$ is viewed as a singleton multiset on $T_0 + T_1$.

Lemma 6.3.2 expresses square-free, irreducible S-compatibles in terms of square-free, irreducible S'-compatibles, with $S' \subseteq S$.

Lemma 6.3.2. *If $S \subseteq A_0 \cap A_1$, and $a \in (A_0 \cap A_1) \setminus S$ then*

$$\overline{C_0}(S \cup \{a\}) \;\subseteq\; \overline{C_0^0}(S) \cup \{\alpha_{-1} \cup \alpha_1 \mid \alpha_d \in \overline{C_0^d}(S)\} \;\subseteq\; C(S \cup \{a\}).$$

Proof. For the first inclusion, let $\eta \in C_0(S \cup \{a\})$. Since every $S \cup \{a\}$-compatible is also S-compatible, we have that $\eta \in C(S)$. As $C(S)$ is a graded monoid, Lemma 6.1.1 shows that, for some $n \geq 1$ and $\beta_1, \ldots, \beta_n \in C_0(S)$, we have

$$\eta = \beta_1 \cup \cdots \cup \beta_n$$

Since η is square-free, we have,for every $1 \leq k \leq n$, that

$$\ell(\beta_k)(a) \leq \textstyle\sum_{i=1}^n \ell(\beta_i)(a) = \ell(\bigcup_{i=1}^n \beta_i)(a) = \ell(\eta)(a) \leq 1,$$

which shows that every β_i, $1 \leq i \leq n$, is square-free. In particular, the difference of β_i at a satisfies $d_a(\beta_i) \in \{-1, 0, 1\}$. We distinguish two cases:

Case 1: Suppose that $d_a(\beta_1) = 0$. Then, β_1 and $\beta_2 \cup \ldots \cup \beta_n$ are both S-compatible. Irreducibility of η shows that $n = 1$. Hence, $\eta = \beta_1 \in \overline{C_0^0}(S)$.

Case 2: Suppose that $d_a(\beta_1) = \pm 1$. Since η is S-compatible, we have that

$$\textstyle\sum_{i=1}^n d_a(\beta_i) = d_a(\bigcup_{i=1}^n \beta_i) = d_a(\eta) = 0$$

Since $d_a(\beta_i) \in \{-1, 0, 1\}$, for $1 \leq i \leq n$, we find some $1 < k \leq n$, such that $d_a(\beta_k) = -d_a(\beta_1) = \mp 1$. Without loss of generality, we assume that $k = 2$. From

Algorithm 1: Distribution

Input : Two finite, square-free multilabeled nets N_0 and N_1.
Output: $\overline{C_0}(A_0 \cap A_1) \subseteq C \subseteq C(A_0 \cap A_1)$.

```
C ← T_0 + T_1 ⊆ ℕ^(T_0+T_1);
foreach a ∈ A_0 ∩ A_1 do
    foreach d ∈ {−1, 0, 1} do
        C^d ← {η ∈ C | d_a(η) = d};
    C ← C^0 ∪ {α ∪ β | (α, β) ∈ C^{−1} × C^1, α ∪ β square-free};
```

$d_a(\beta_1 \cup \beta_2) = d_a(\beta_1) + d_a(\beta_2) = 0$, it follows that $\beta_1 \cup \beta_2$ and $\beta_3 \cup \ldots \cup \beta_n$ are both S-compatible. Irreducibility of η and non-emptyness of β_i, for $1 \leq i \leq n$, shows that $n = 2$, which implies that $\eta = \beta_1 \cup \beta_2$, with $\beta_i \in \overline{C_0^{\pm 2i \mp 3}}(S)$, for $i \in \{1, 2\}$. In both cases, $\eta \in \overline{C_0^0}(S) \cup \{\alpha_{-1} \cup \alpha_1 \mid \alpha_d \in \overline{C_0^d}(S)\}$.

For the second inclusion, let $\eta \in \overline{C_0^0}(S) \cup \{\alpha_{-1} \cup \alpha_1 \mid \alpha_d \in \overline{C_0^d}(S)\}$. Suppose that $\eta \in \overline{C_0^0}(S)$. By construction, $d_a(\eta) = 0$, which shows that $\eta \in C(S \cup \{a\})$. Every decomposition of η in $C(S \cup \{a\})$ is also a decomposition in $C(S)$. Hence, irreducibility of η in $C(S)$ implies that $\eta \in \overline{C_0}(S \cup \{a\})$.

Suppose $\eta \in \{\alpha_{-1} \cup \alpha_1 \mid \alpha_d \in \overline{C_0^d}(S)\}$. Then, we have $d_a(\eta) = d_a(\alpha_{-1} \cup \alpha_1) = d_a(\alpha_{-1}) + d_a(\alpha_1) = -1 + 1 = 0$. Hence, $\eta \in C(S \cup \{a\})$. □

Algorithm 1 computes a set $C \subseteq C(A_0 \cap A_1)$ of compatible global transitions of two square-free multilabeled nets N_0 and N_1, including all square-free, irreducible global transitions (i.e., $C \supseteq \overline{C_0}(A_0 \cap A_1)$). We conjecture that Algorithm 1 actually generates $C = \overline{C_0}(A_0 \cap A_1)$, which means that Algorithm 1 produces only irreducible global transitions.

For clarity, we present distribution (Algorithm 1) in its simplest form. Distribution can be optimized by using an appropriate data structure for set C for efficient constructions of the subsets $C^d = \{\eta \in C \mid d_a(\eta) = d\}$, for $d \in \{-1, 0, 1\}$.

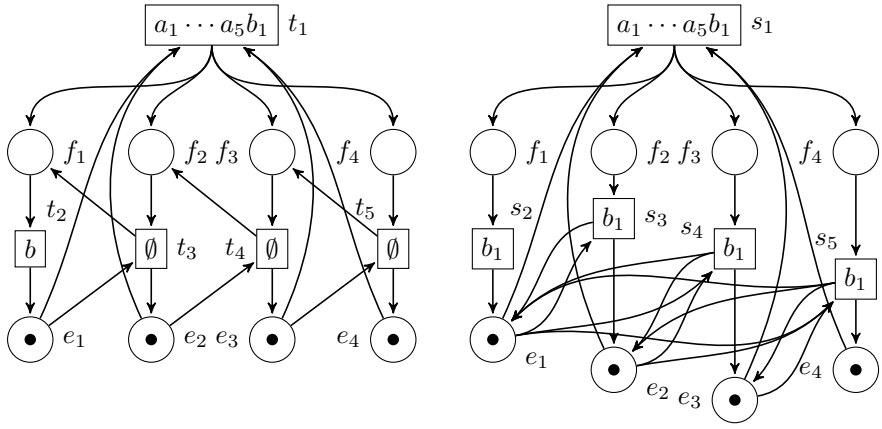
Theorem 6.3.3. *Distribution (Algorithm 1) is totally correct.*

Proof. By Lemma 6.3.2, after $S \subseteq A_0 \cap A_1$ iterations, set C consists of all square-free, irreducible S-compatibles. Finiteness of N_0 and N_1 ensures that $A_0 \cap A_1$ is finite, which implies termination. □

Example 6.3.2 (Alternator)**.** Consider the $\mathsf{alternator}_n$ protocol, $n \geq 2$, defined as the following composition of node, sync, $\mathsf{syncdrain}$, and fifo_1 components:

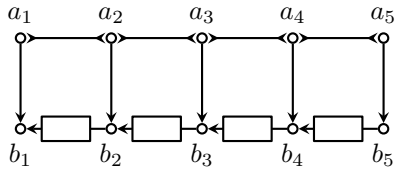
$$\begin{aligned}\mathsf{alternator}_n = {} & \textstyle\prod_{i=1}^{n} \left(\mathsf{node}_{1,3}(a_i; a_i^1, a_i^2, a_i^3) \times \mathsf{node}_{2,1}(b_i^1, b_i^2; b_i) \times \mathsf{sync}(a_i^2, b_i^1)\right) \\ & \times \textstyle\prod_{i=1}^{n-1} \left(\mathsf{syncdrain}(a_i^3, a_{i+1}^1) \times \mathsf{fifo}_1(b_{i+1}, b_i^2)\right),\end{aligned}$$

where the multilabeled net for $\mathsf{syncdrain}$ and sync are identical. For those familiar with the syntax of Reo, $\mathsf{alternator}_5$ has the following diagram:



(a) Composition (b) Abstraction

Figure 6.3: Composition and abstraction of the $\mathsf{alternator}_5$ protocol.



Using Algorithm 1 and Theorem 6.3.3, we can compute the multilabeled net of $\mathsf{alternator}_5$. The composition is shown in Figure 6.3(a). For readability, we do not draw the places induced by the node components. Their contents remains the same throughout the execution of the $\mathsf{alternator}_5$. We also hide all internal actions by removing from the transition labels all actions other than b_1 and $a_1, \ldots, a_5$. We formalize this procedure in Definition 6.4.1.

The multilabeled net in Figure 6.3(a) can be used for the implementation of the $\mathsf{alternator}_5$ protocol. In a naive implementation, a place is implemented as a variable, and a transtion is implemented as a thread. Each thread reads and writes to these variables according to the flow relation, and performs I/O-operations according to the transition label. In particular, transitions with an empty label do not perform any I/O-operation. Of course, care must be taken for variables that are shared amongst different threads. ◊

The number of places in a composition $N_0 \times N_1$ is the sum of the places in N_0 and N_1, which shows that the composition operator in Definition 6.3.3 does not suffer from state-space explosions. The total number of transitions in a composition $N_0 \times N_1$ equals the number of irreducible compatible subsets of N_0 and N_1, which can potentially grow large. The following result shows that, for 'nice compositions', the number of transitions in the composition does not blow up.

Corollary. *The composition $N_0 \times N_1$ has at most $|T_0| + |T_1|$ transitions, if for every $a \in A_0 \cap A_1$ there exists an $i \in \{0, 1\}$ with $|\{t \in T_i \mid \ell(t)(a) > 0\}| \leq 1$.*

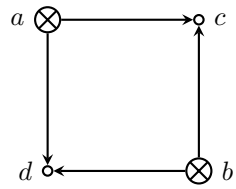


Figure 6.4: Synchronous regions lose concurrency.

Proof. The condition ensures that, in Algorithm 1, $|C^{-1}| = 1$ or $|C^1| = 1$. Hence, the size of C does not increase, which shows that the upper bound holds. □

Example 6.3.3. As shown in [JKA17, Fig. 17(a)], the constraint automaton representation of the $\mathsf{alternator}_n$ protocol does not scale well in $n \geq 1$. However, for multilabeled Petri nets the situation is completely different. Applying Line 5 to the compositions in $\mathsf{alternator}_n$ in Example 6.3.2, it can be shown that, for every $n \geq 1$, the number of transitions of the $\mathsf{alternator}_n$ is equal to n. As such, the multilabeled net representation of the $\mathsf{alternator}_n$ protocol does not suffer from a state-space or a transition-space explosion. ◊

6.3.2 Discussion on concurrency in Reo

Despite the wealth of available semantics for Reo connectors [JA12], most of them focus only on synchronization, but ignore the concurrent behavior of a Reo connector. Since these semantics form the basis of the Reo compilers, these deficiencies in the semantics of Reo leads to problems in the implementation of the Reo language.

For example, Jongmans uses constraint automata (CA) for his compiler for Reo protocols. Since CA are sequential machines, and a sequential implementation of Reo connector is not utilizing all available resources, Jongmans first decomposes the Reo protocol in synchronous regions and runs those regions in parallel with minimal synchronization overhead. In other words, Jongmans partially compensates for the loss of concurrency in CA by considering a decomposition of the CA and synchronizing these components at run time.

Unfortunately, the synchronous regions decomposition does not recover all available concurrency that was present in the original Reo protocol, as demonstrated by the following example:

Example 6.3.4. Consider the connector in Figure 6.4. It is a single synchronous region and it compiled into a single threaded program that executes the constraint automaton in Figure 6.5. The transitions ac and bd are independent and can fire concurrently in the original Reo connector. However, the sequential implementation as a CA loses this independence. ◊

In contrast with the CA semantics, our multilabeled Petri net semantics models both the concurrent and synchronous behavior of a Reo connector. It is captures all available concurrency event within a single synchronous region:

Example 6.3.5. Figure 6.4 shows the multilabeled Petri net for the synchronous regions from Example 6.3.4. The places ensure that no two adjacent transitions

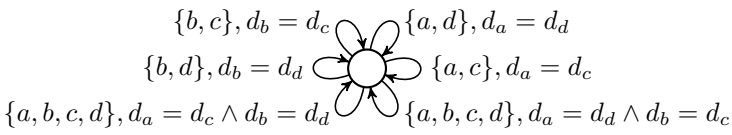


Figure 6.5: Constraint automaton of Figure 6.4

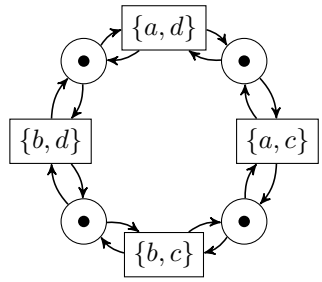


Figure 6.6: Multilabeled Petri net of Figure 6.4

can fire simultaneously. Hence, the $\{a,c\}$ and $\{b,d\}$ can fire at the same time. The same applies to the $\{a,d\}$ and $\{b,c\}$ transitions. $\Diamond$

6.4 Abstraction

The definition of multilabeled Petri nets allows for *silent* transitions, i.e., transitions t with an empty label $\ell(t) = \emptyset$. Such silent transitions can be introduced by hiding internal actions:

Definition 6.4.1 (Hiding)**.** Hiding an action $a \in A$ in a multilabeled net N yields the net $\exists_a N = (B, P, \{({}^\bullet t, \ell(t)|_B, t^\bullet) \mid t \in T\}, \mu)$, where $B = A \setminus \{a\}$ and $\ell(t)|_B$ is the restriction of $\ell(t)$ to B.

The hiding operator $\exists_a$ simply drops all occurrences of a from the label of every transition in the given Petri net.

As indicated in Example 6.3.2, the naive implementation of a silent transition is a thread that does not perform any I/O-operations. As a result, these silent transitions can delay the throughput of the protocol.

In this section, we aim to improve the generated code by transforming a fixed multilabeled net $N = (A, P, T, \mu_0)$ into an equivalent net ∂N without any silent transitions. To define the abstraction operator ∂, we follow the same strategy as for composition of multilabeled nets. First, we consider all possible sequential compositions of transitions. Next, we restrict to sequences of transitions with observable behavior. Finally, we restrict to sequences of transitions that generate all observable traces.

6.4.1 Sequential compositions

Consider the set T^* of all firing sequences of N, i.e., all finite sequences of transitions of N. Following Mazurkiewicz [Maz95], strictly different firing sequences can be considered identical up to permutation of independent transitions.

Definition 6.4.2. The **dependency relation** $D \subseteq T \times T$ is defined as

$$(s,t) \in D \quad \Leftrightarrow \quad s^\bullet \cap {}^\bullet t \neq \emptyset \;\text{ or }\; t^\bullet \cap {}^\bullet s \neq \emptyset \;\text{ or }\; \ell(s) \neq \emptyset \neq \ell(t)$$

Intuitively, transitions s and t are dependent iff one transition takes the output of the other as input, or if both transitions are observable. Note that conflicting transitions are not necessarily dependent.

The dependency relation D induces a **trace equivalence** $\equiv \subseteq T^*$ defined as the smallest congruence on T^*, such that $st \equiv ts$, for all $(s,t) \notin D$. An equivalence class $[x] = \{y \in T^* \mid y \equiv x\}$ of a firing sequence x is called a **trace**.

The **trace monoid** $T^*/\!\equiv$ is the set $\{[x] \mid x \in T^*\}$ of all traces, endowed with composition, defined, for all $x, y \in T^*$, as $[x][y] = [xy]$. Since $\equiv$ is a congruence, composition of traces is well-defined.

Observable behavior of traces is a map $o : T^*/\!\equiv \to (\mathbb{N}^A \setminus \{\emptyset\})^*$ defined, for all $x \in T^*$, as

$$o([\epsilon]) = \epsilon, \qquad o([xt]) = \begin{cases} o([x])\ell(t) & \text{if } \ell(t) \neq \emptyset \\ o([x]) & \text{otherwise} \end{cases}$$

Since observable transitions do not commute (Definition 6.4.2), o is well-defined.

Next, we define map $\sigma : T^*/\!\equiv \to \mathbb{N}^P \times \mathbb{N}^A \times \mathbb{N}^P$ that maps every trace w to a concrete transition $\sigma(w) \in \mathbb{N}^P \times \mathbb{N}^A \times \mathbb{N}^P$. The definition of this map relies on sequential composition of transitions:

Definition 6.4.3. The **sequential composition** $s;t$ of transitions s, t in a multilabeled net N is defined as $({}^\bullet(s;t), \ell(s;t), (s;t)^\bullet)$, where

$${}^\bullet(s;t) = {}^\bullet s \cup ({}^\bullet t \setminus s^\bullet), \qquad (s;t)^\bullet = t^\bullet \cup (s^\bullet \setminus {}^\bullet t), \qquad \ell(s;t) = o([st])$$

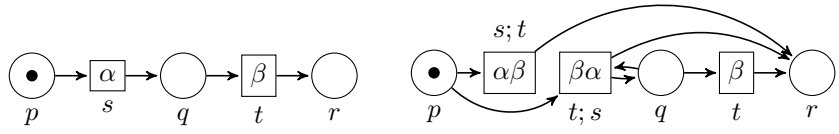
For a sequential composition $s;t$, transition t can use the tokens produced by s. Hence, t consumes only the tokens from the multiset difference ${}^\bullet t \setminus s^\bullet$. Therefore, the sequential composition $s;t$ consumes only the tokens in the multiset union ${}^\bullet s \cup ({}^\bullet t \setminus s^\bullet)$.

Example 6.4.1. Figure 6.7(b) shows some sequential compositions of transitions s and t in Figure 6.7(a). Intuitively, the sequential composition $s;t$ performs t after s. The token generated in place q by s is immediately consumed by t. Therefore, $s;t$ does not have q as input place or output place.

Note, however, that $t;s$ has q both as input place and output place, because the token produced at place q by s comes too late. ◊

Sequential composition of traces is a map $\sigma : T^*/\!\equiv \to \mathbb{N}^P \times \mathbb{N}^A \times \mathbb{N}^P$ defined, for all traces $w \in T^*/\!\equiv$ and transitions $t \in T$, as

$$\sigma([\epsilon]) = (\emptyset, \emptyset, \emptyset), \qquad \sigma(wt) = \sigma(w);t. \tag{6.1}$$



(a) Two dependent transitions (b) Sequential composition

Figure 6.7: Sequential composition of transitions with multilabels α and β.

Lemmas 6.4.1 and 6.4.2 show that σ is a well-defined homomorphism, that is, $\sigma(u)$ does not depend on the representative of $u \in T^*/\equiv$, and $\sigma(uv) = \sigma(u);\sigma(v)$, for all traces $u, v \in T^*/\equiv$.

Lemma 6.4.1. *If $(s,t) \notin D$, then $s;t = t;s$.*

Proof. If $(s,t) \notin D$, then $s^\bullet \cap {}^\bullet t = t^\bullet \cap {}^\bullet s = \emptyset$ and either $\ell(s) \neq \emptyset$ or $\ell(t) \neq \emptyset$. The latter condition implies that $\ell(s;t) = o([st]) = o([ts]) = \ell(t;s)$. The former condition implies that ${}^\bullet t \setminus s^\bullet = {}^\bullet t$ and ${}^\bullet s \setminus t^\bullet = {}^\bullet s$, which implies

$$ {}^\bullet(s;t) = {}^\bullet s \cup ({}^\bullet t \setminus s^\bullet) = ({}^\bullet s \setminus t^\bullet) \cup {}^\bullet t = {}^\bullet(t;s) $$

Similarly, it follows that $(s;t)^\bullet = (t;s)^\bullet$, which proves $s;t = t;s$. □

By construction, sequential composition of traces σ parses each trace as a left-associative composition. Thus, for traces u and v in $T^*/\equiv$, $\sigma(uv)$ and $\sigma(u);\sigma(v)$ could evaluate to different transitions. Lemma 6.4.2 shows that these expressions are equal, and that sequential composition of traces is a homomorphism.

Lemma 6.4.2. *Sequential composition of transitions is associative.*

Proof. The identities in Section 6.1 show, for all transitions $r, s, t \in T$, that

$$ \begin{aligned} {}^\bullet(r;(s;t)) &= {}^\bullet r \cup ({}^\bullet(s;t) \setminus r^\bullet) \\ &= {}^\bullet r \cup (\ (\ {}^\bullet s \cup ({}^\bullet t \setminus s^\bullet)\ ) \setminus r^\bullet\ ) \\ &= {}^\bullet r \cup ({}^\bullet s \setminus r^\bullet) \cup (\ ({}^\bullet t \setminus s^\bullet) \setminus (r^\bullet \setminus {}^\bullet s))\ ) \\ &= {}^\bullet(r;s) \cup (\ {}^\bullet t \setminus (\ s^\bullet \cup (r^\bullet \setminus {}^\bullet s)\ )\ ) \\ &= {}^\bullet(r;s) \cup ({}^\bullet t \setminus (r;s)^\bullet) \\ &= {}^\bullet((r;s);t) \end{aligned} $$

Similarly $(r;(s;t))^\bullet = ((r;s);t)^\bullet$. Since concatenation is associative,

$$ \ell(x;(y;z)) = o(x)o(y)o(z) = \ell((x;y);z), $$

which shows that $x;(y;z)$ and $(x;y);z$. □

Sequential composition of traces induces **observational equivalence** $\approx \subseteq T^*/\equiv$ defined as $v \approx w$ iff $\sigma(v) = \sigma(w)$. Since sequential composition of traces is a homomorphism, observable equivalence $\approx$ is a congruence.

6.4.2 Elimination of silent traces

Consider the set $O = \{w \in T^*/\equiv \mid o(w) \neq \epsilon\}$ of observable traces, which constitutes a proper right-ideal in the trace monoid $T^*/\equiv$. Lemma 6.1.2 shows that every observable trace w, with $o(w) \neq \epsilon$, can be decomposed as a sequence of right-irreducibles in O followed by a silent trace v, with $o(v) = \epsilon$.

Every trace w whose observable behavior $o(w)$ has length greater than one admits a decomposition uv, with $o(u) \neq \epsilon$ and $v \not\equiv \epsilon$. Hence, the observable behavior $o(w)$ of a right-irreducible trace $w \in O$ must have length one.

Nevertheless, the length $|w|$ of the trace w can be arbitrarily large. To shrink the set of traces that generate all observable behavior, we consider two additional conditions.

Definition 6.4.4. A trace w is **acyclic** iff $|w| = \min_{w' \approx w} |w'|$.

Every contiguous subtrace of an acyclic trace is acyclic.

Lemma 6.4.3. *If uvw is acyclic, then v is acyclic.*

Proof. If v is not acyclic, then there exists a trace $v' \approx v$, such that $|v'| < |v|$. Then, $|uv'w| < |uvw|$, and $\sigma(uv'w) = \sigma(u);\sigma(v');\sigma(w) = \sigma(u);\sigma(v);\sigma(w) = \sigma(uvw)$. Hence, $uv'w \approx uvw$, and uvw is not acyclic. □

Definition 6.4.5. A trace w is **k-live** iff $\sigma(w')$ is k-live, for every suffix w' of w.

Every contiguous subtrace of a k-live trace is k-live.

Lemma 6.4.4. *If uvw is k-live, then v is k-live.*

Proof. If v is not k-live, then v can be decomposed as $v = v_0v_1$, such that $\sigma(v_1)$ is not k-live. Then, uvw can be decomposed as $(uv_0)(v_1w)$. From ${}^\bullet\sigma(v_1w) = {}^\bullet(\sigma(v_1);\sigma(w)) = {}^\bullet\sigma(v_1) \cup ({}^\bullet\sigma(w) \setminus \sigma(v_1)^\bullet) \supseteq {}^\bullet\sigma(v_1)$, it follows that $\sigma(v_1w)$ is also not k-live, and that uvw is not k-live. □

The results in Lemmas 6.1.2, 6.4.3 and 6.4.4 show that the trace $w = [t_1 \dots t_n]$ of every firing sequence $t_1 \dots t_n$, $n \geq 1$, can be generated from right-irreducible, acyclic, k-live traces modulo observational equivalence $\approx$. To see this, note that w comes from a firing sequence, which means that w is k-live by construction. Now, select some acyclic $w' \approx w$ (i.e., w' is of minimal length). Lemma 6.1.2 applied for the right-ideal $O = \{u \in T^*/\equiv \mid o(u) \neq \epsilon\}$ yields a decomposition $w' = a_1 \dots a_n b$, where $a_1, \dots, a_n$ are right-irreducible in O, and $o(b) = \epsilon$ is silent. Since w is acylcic and k-live, Lemmas 6.4.3 and 6.4.4 show that the traces $a_1, \dots, a_n$ are right-irreducible, acyclic, and k-live.

Definition 6.4.6. The **abstraction** ∂N of the multilabeled net N is defined as $(A, P, \{\sigma(w) \mid w \text{ right-irreducible, acyclic, and } k\text{-live}\}, \mu)$.

Example 6.4.2. Consider the composite net N in Figure 6.1(c). Hiding actions a and b results in a net $N' = \exists_a \exists_b N$, where send | transfer^2 | receive^2 is a silent transition. From the current marking, the net N' can eventually produce observable behavior. However, to do so, N' must first firing a silent transition.

Application of the abstraction operator yields the $\partial N'$, as shown in Figure 6.1(d). In contrast with N', the abstraction $\partial N'$ can directly fire an observable transition

Algorithm 2: Abstraction

Input : A finite, k-bounded multilabeled net $N = (A, P, T, \mu_0)$.
Output: $H = \{x \in T^* \mid [x]$ right-irreducible, acyclic, and k-live$\}$.

$H \leftarrow \{t \mid t \in T,\ \ell(t) \neq \emptyset\}$;
repeat
  $h \leftarrow |H|$;
  $H \leftarrow H \cup \{sx \mid x \in H,\ s \in T,\ \ell(s) = \emptyset,\ sx \not\equiv xs,\ \sigma([sx]) \notin \sigma(H),\ [sx]\ k\text{-live}\}$;
until $h = |H|$;

in the current marking. As a result, the abstraction $\partial N'$ produces observable behavior faster than N', and the executable code generated from $\partial N'$ potentially optimizes the code generated from N'. ◊

6.4.3 Abstraction algorithm

The transitions of the abstraction ∂N of a multilabeled net N can be computed by recursion on the length of the underlying traces. Algorithm 2 shows a straightforward tree search that starts from the shortest possible acyclic, k-live, right-irreducible traces, namely the observable transitions.

Theorem 6.4.5. *Algorithm 2 is totally correct.*

Proof. It is routine to check that, after $n \geq 0$ iterations, H contains all acyclic, k-live, right-irreducible traces of length $n + 1$. For termination, observe that a finite, k-bounded net has only finitely many k-live transitions. □

Example 6.4.3. We use Algorithm 2 to eliminate the silent transitions $\mathsf{alternater}_5$ protocol as shown in Figure 6.3(a). Table 6.1 shows the intermediate steps

Figure 6.3(b) shows the resulting multilabeled net $\partial\mathsf{alternator}_5$. In $\partial\mathsf{alternator}_5$, a reader component at the output b_1 of $\mathsf{alternator}_5$ does not need to wait for silent transitions to move the data into the fifo_1 buffer between b_2 and b_1. Instead, the reader can immediately take the data from the first non-empty fifo_1 buffer. As such, the naive implementation of the abstraction $\partial\mathsf{alternator}_5$ should have higher throughput than $\mathsf{alternator}_5$. We did not yet verify this claim experimentally. ◊

6.5 Discussion

We introduce multilabeled Petri nets, i.e., Petri nets whose transitions are labeled by a multiset of actions. We define a binary composition operator for multilabeled nets, and construct an algorithm to compute the composition. We define a unary abstraction operator that removes silent transitions from multilabeled nets, and construct an algorithm to compute the abstraction.

The composition algorithm Algorithm 1 assumes that the multilabeled nets are square-free. Although the assumption of square-freeness is justified in the context

Table 6.1: Abstraction of the multilabeled net in Figure 6.3(a) using Algorithm 2.

w	s	${}^\bullet w$	$w^\bullet$	$\ell(w)$	acyclic	1-live	irreduc.
t_1	s_1	$e_1e_2e_3e_4$	$f_1f_2f_3f_4$	$a_1\cdots a_5b_1$	y	y	y
t_2	s_2	f_1	e_1	b	y	y	y
t_3		e_1f_2	e_2f_1	$\emptyset$	y	y	y
t_4		e_2f_3	e_3f_2	$\emptyset$	y	y	y
t_5		e_3f_4	e_4f_3	$\emptyset$	y	y	y
t_3t_1		$e_1^2e_3e_4f_2$	$f_1^2f_2f_3f_4$	$a_1\cdots a_5b_1$	y		y
t_4t_1		$e_1e_2^2e_4f_3$	$f_1f_2^2f_3f_4$	$a_1\cdots a_5b_1$	y		y
t_5t_1		$e_1e_2e_3^2f_4$	$f_1f_2f_3^2f_4$	$a_1\cdots a_5b_1$	y		y
t_3t_2	s_3	e_1f_2	e_1e_2	b	y	y	y
t_4t_2		$e_2f_1f_3$	$e_1e_3f_2$	b	y	y	
t_5t_2		$e_3f_1f_4$	$e_1e_4f_3$	b	y	y	
$t_3t_3t_2$		$e_1^2f_2^2$	$e_1e_2^2f_1$	b	y		y
$t_4t_3t_2$	s_4	$e_1e_2f_3$	$e_1e_2e_3$	b	y	y	y
$t_5t_3t_2$		$e_1e_3f_2f_4$	$e_1e_2e_4f_3$	b	y	y	
$t_3t_4t_3t_2$		$e_1^2f_2f_3$	$e_1e_2e_3f_1$	b	y		y
$t_4t_4t_3t_2$		$e_1e_2^2f_3^2$	$e_1e_2e_3^2f_2$	b	y		y
$t_5t_4t_3t_2$	s_5	$e_1e_2e_3f_4$	$e_1e_2e_3e_4$	b	y	y	y
$t_3t_5t_4t_3t_2$		$e_1^2e_3f_2f_4$	$e_1e_2e_3e_4f_1$	b	y		y
$t_4t_5t_4t_3t_2$		$e_1e_2^2f_3f_4$	$e_1e_2e_3e_4f_2$	b	y		y
$t_5t_5t_4t_3t_2$		$e_1e_2e_3^2f_4^2$	$e_1e_2e_3e_4^2f_3$	b	y		y

of constraint automata, it would be very useful to be able to automatically compose more general multilabeled nets (such as the running example).

While the definition of the composition operator relies on the concurrent semantics of nets, the definition of the abstraction operator relies on interleaving semantics. As a result, the composition and abstraction operator are not interoperable, in the sense that $\partial(N_0 \times N_1) \neq \partial N_0 \times \partial N_1$, for multilabeled nets N_0 and N_1. Such an identity would allow for simplification of intermediate compositions, which potentially speeds up the construction of the composition.

Part III

Scheduling

Chapter 7

Protocols with Workloads

The interaction protocol in a concurrent software imposes dependencies amongst the different processes in this software. For example, the rate at which a producer process can fill a buffer of bounded capacity depends on the rate at which a consumer process drains it. In most applications, the protocol is defined implicitly as a combination of locks and semaphores. For software written in an exogenous coordination language (cf., Part I) the protocol is explicit, which reveals the dependencies that are relevant for scheduling.

We can exploit this relevant scheduling information to optimize the execution of concurrent software. For example, smart scheduling of processes can offer protection against concurrent access of shared resources in a concurrent application, without suffering from drawbacks of the standard mutual exclusion protocols (e.g., locks). Imagine we have a crystal ball that accurately reveals when each process accesses its resources and their proper order of execution. We can then use this information to synthesize a scheduler that executes the processes in the correct order and prevents concurrent access to shared resources by speeding up or slowing down the execution of each process. Locks now become redundant, and their overhead can be avoided.

Unfortunately, some relevant scheduling information is lost if we represent the protocol in one of the existing protocol semantics (such as our stream constraints from Part II), which leaves us with a blurred crystal ball. Existing semantics encode precisely the order of all interactions, but they ignore the amount of work that must be performed in between these interactions. Therefore, existing protocol semantics are inadequate and prevent us from formulating the scheduling problem. In the current chapter, we develop *work automata*, which is a semantics for components and protocols that allows us to formalize the scheduling problem[1].

In Section 7.1 we introduce the syntax and semantics of work automata and define a weak simulation relation that allows us to compare the behavior of two work automata. We define two fundamental operations, namely composition and hiding. Composition allows us to construct a work automaton for a large system by composing the work automata of its smaller subsystems. The composition operation synchronizes the behavior on shared (inter)actions. Although these shared

[1]The work in this chapter is based on [DA17]

interactions are relevant for composition, we might want to ignore them in a larger context. The hiding allows us to do this. By defining composition and hiding of work automata, we can extend our Treo language from Chapter 4 with syntax to express primitive work automata. Our generic (semantics-agnostic) Reo then automatically computes the work automaton of the complete application. We potentially benefit from all semantics-independent compiler optimizations, such as the queue-optimization [JHA14] and protocol splitting [JCP16].

Composition of work automata may suffer from a state space explosion. A large number of states in a work automaton complicates its analysis. In Section 7.2, we introduce state-space minimization techniques to counter this state space explosion. We define in Section 7.2 two procedures, called *translation* and *contraction*, that simplify a given work automaton by minimizing its number of states. We provide conditions (Theorems 7.2.3 and 7.2.5) under which translation and contraction preserve weak simulation.

We show by means of an example that some large work automata can be simplified to their respectively "equivalent" single state work automata. The state-invariant of the single state of such a resulting automaton defines a region in a multidimensional real vector space. Geometric features of this region reveal interesting behavioral properties of the corresponding concurrent application. For example, (explicit or implied) mutual exclusion in an application corresponds to a hole in its respective region, and non-blocking executions correspond to straight lines through this region. Since straight lines are easier to detect than non-blocking executions, the geometric perspective provides additional insight into the behavior of an application. We postulate that such information may be used to develop a smart scheduler that avoids the drawbacks of locks.

In Section 7.3, we discuss related work, and in Section 7.4 we conclude and point out future work.

7.1 Work automata

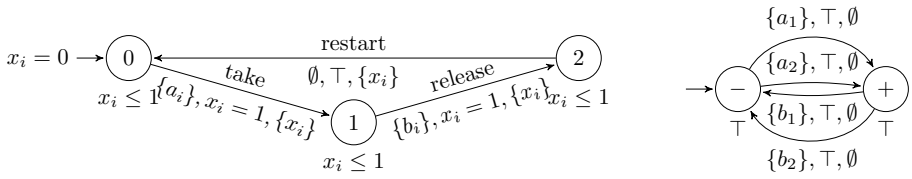
7.1.1 Syntax

Consider an application A that consists of $n \geq 1$ concurrently executing processes $X_1, \ldots, X_n$. We measure the progress of each process X_i in A by a positive real variable $x_i \in \mathbb{R}_+$, called a *job*, and represent the current *progress of application* A by a map $p : J \to \mathbb{R}_+$, where $J = \{x_1, \ldots, x_n\}$ is the set of all jobs in A. We regulate the progress using boolean constraints $\phi \in B(J)$ over jobs:

$$\phi \ ::= \ \top \ \mid \ \bot \ \mid \ x \sim n \ \mid \ \phi_0 \wedge \phi_1 \ \mid \ \phi_0 \vee \phi_1, \tag{7.1}$$

with $\sim \in \{\leq, \geq, =\}$, $x \in J$ a job and $n \in \mathbb{N}_0 \cup \{\infty\}$. We define *satisfaction* $p \models \phi$ of a progress $p : J \to \mathbb{R}_+$ and a constraint $\phi \in B(J)$ by the following rules: $p \models x \sim n$, if $p(x) \sim n$; $p \models \phi_0 \wedge \phi_1$, if $p \models \phi_0$ and $p \models \phi_1$; $p \models \phi_0 \vee \phi_1$, if $p \models \phi_0$ or $p \models \phi_1$. The *interface* of application A consists of a set of ports through which A interacts with its environment via synchronous operations, each one involving a subset $N \subseteq P$ of its ports.

We define the exact behavior of a set of processes as a labeled transition system called a *work automaton*. The progress value $p(x)$ of job x may increase in a state



(a) Process A_i (b) Lock L

Figure 7.1: Mutual exclusion of processes A_1 and A_2 by means of a lock L.

q of a work automaton, as long as the *state-invariant* $I(q) \in B(J)$ is satisfied. A state-invariant $I(q)$ defines the amount of work that each process can do in state q before it blocks. A transition $\tau = (q, N, w, R, q')$ allows the work automaton to reset the progress of each job $x \in R \subseteq J$ to zero and change to state q', provided that the *guard*, defined as *synchronization constraint* $N \subseteq P$ together with the *job constraint* $w \in B(J)$, is satisfied. That is, the transition can be fired, if the environment is able to synchronize on the ports N and the current progress $p : J \to \mathbb{R}_+$ of A satisfies job constraint w.

Definition 7.1.1 (Work automata)**.** A work automaton is a tuple $(Q, P, J, I, \to, \phi_0, q_0)$ that consists of a set of states Q, a set of ports P, a set of jobs J, a state invariant $I : Q \to B(J)$, a transition relation $\to \subseteq Q \times 2^P \times B(J) \times 2^J \times Q$, an initial progress $\phi_0 \in B(J)$, and an initial state $q_0 \in Q$.

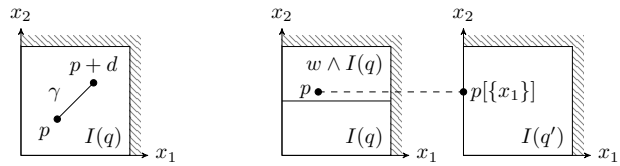
Example 7.1.1 (Mutual exclusion)**.** Figure 7.1 shows the work automata of two identical processes A_1 and A_2 that achieve mutual exclusion by means of a global lock L. The progress of process A_i is recorded by its associated job x_i, and the interface of each process A_i consists of two ports a_i and b_i. Suppose we ignore the overhead of the mutual exclusion protocol. Then, lock L does not need a job and its interface consists of ports a_1, a_2, b_1, and b_2. Each process A_i starts in state 0 with $\phi_0 := x_i = 0$ and is allowed to execute at most one unit of work, as witnessed by the state-invariant $x_i \leq 1$. After finishing one unit of work, A_i starts to compete for the global lock L by synchronizing on port a_i of lock L. When A_i succeeds in taking the lock, then lock L changes its state from $-$ to $+$ and process A_i moves to state 1, its critical section, and resets the progress value of job x_i to zero. Next, process A_i executes one unit of work in its critical section. Finally, A_i releases lock L by synchronizing on port b_i, executes asynchronously its last unit of work in state 2, and resets to state 0. ◊

7.1.2 Semantics

We define the semantics of a work automaton $A = (Q, P, J, I, \to, \phi_0, q_0)$ by means of a finer grained labeled transition system $[\![A]\!]$ whose states are configurations:

Definition 7.1.2 (Configurations)**.** A configuration of a work automaton A is a pair $(p, q) \in \mathbb{R}_+^J \times Q$, where $p : J \to \mathbb{R}_+$ is a state of progress, and $q \in Q$ a state.

The transitions of $[\![A]\!]$ are labeled by two kinds of labels: one for advancing progress of A and one for changing the current state of A. To model advance of



(a) Progress (b) State transition with reset

Figure 7.2: Progress (a) of the application along the path γ in $I(q)$ from p to $p+d$, and (b) transition from state q to q' with reset of job x_1.

progress of A, we use a map $d : J \to \mathbb{R}_+$ representing that $d(x)$ units of work has been done on job x. Such a map induces a transition

$$(p, q) \xrightarrow{d} (p + d, q), \tag{7.2}$$

where $+$ is component-wise addition of maps (i.e., $(p + d)(x) = p(x) + d(x)$, for all $x \in J$). Figure 7.2(a) shows a graphical representation of transition Equation (7.2). A state of progress p of A corresponds to a point in the plane.

In practice, the value of each job $x \in J$ continuously evolves from $p(x)$ to $p(x) + d(x)$. We assume that, during transition Equation (7.2), each job makes progress at a constant speed. This allows us to view the actual execution as a path $\gamma : [0, 1] \to \mathbb{R}_+^J$ defined by $\gamma(c) = p + c \cdot d$, where $\mathbb{R}_+^J$ is the set of maps from J to $\mathbb{R}_+$ and $\cdot$ is component-wise scalar multiplication (i.e., $(p+c\cdot d)(x) = p(x)+c\cdot d(x)$, for all $x \in J$). At any instant $c \in [0, 1]$, the state of progress $p + c \cdot d$ must satisfy the current state-invariant $I(q)$. Figure 7.2(a) shows execution γ as the straight line connecting p and $p + d$. For every $c \in [0, 1]$, state of progress $\gamma(c) = p + c \cdot d$ corresponds to a point on the line from p to $p + d$. Note that, since we have a transition from p to $p+c\cdot d$ in $[\![A]\!]$ for all $c \in [0, 1]$, Figure 7.2(a) provides essentially a finite representation of an infinite semantics, i.e., one with an infinite number of transitions through intermediate configurations between (p, q) and $(p + d, q)$. In Section 7.2.1, we use this perspective to motivate our gluing procedure.

The transition in Equation (7.2) is possible only if the execution does not block between p and $p + d$, i.e., state of progress $p + c \cdot d$ satisfies the state-invariant $I(q)$ of q, for all $c \in [0, 1]$. Since $I(q)$ defines a region $\{p \in \mathbb{R}_+^J \mid p \models I(q)\}$ of a $|J|$-dimensional real vector space, the non-blocking condition just states that the straight line γ between p and $p + d$ is contained in the region defined by $I(q)$ (see Figure 7.2(a)).

A transition $\tau = (q, N, w, R, q')$ changes the state of the current configuration from q to q', if the environment allows interaction via N and the current state of progress p satisfies job constraint w. As a side effect, the progress of each job $x \in R$ resets to zero. Such state changes occur on transitions of the form

$$(p, q) \xrightarrow{N} (p[R], q'), \tag{7.3}$$

where $p[R](x) = 0$, if $x \in R$, and $p[R](x) = p(x)$ otherwise. Figure 7.2(b) shows a graphical representation of transition Equation (7.3). The current state of progress satisfies both the current state-invariant and the guard of the transition, which

allows to change to state q' and reset the value of x_1 to zero. For convenience, we allow at every configuration (p, q) an $\emptyset$-labeled self loop which models idling.

Definition 7.1.3 (Operational semantics)**.** The semantics of a given work automaton $A = (Q, P, J, I, \rightarrow, \phi_0, q_0)$ is the labeled transition system $[\![A]\!]$ with states $(p, q) \in \mathbb{R}_+^J \times Q$, labels $\mathbb{R}_+^J \cup 2^P$, and transitions defined by the rules:

$$\frac{d : J \to \mathbb{R}_+, \quad \forall c \in [0,1] : p + c \cdot d \models I(q)}{(p, q) \xrightarrow{d} (p + d, q)} \tag{S1}$$

$$\frac{\tau = (q, N, w, R, q') \in \rightarrow, \quad p \models w \wedge I(q), \quad p[R] \models I(q')}{(p, q) \xrightarrow{N} (p[R], q')} \tag{S2}$$

$$\frac{}{(p, q) \xrightarrow{\emptyset} (p, q)} \tag{S3}$$

where $p[R](x) = 0$, if $x \in R$, and $p[R](x) = p(x)$ otherwise.

Based on the operational semantics $[\![A]\!]$ of a work automaton A, we define the *trace semantics* of a work automaton. The trace semantics defines all finite sequences of observable behavior that are *accepted* by the work automaton.

Definition 7.1.4 (Actions, words)**.** Let P be a set of ports and J a set of jobs. An action is a pair $[N, d]$ that consist of a set of ports $N \subseteq P$ and a progress $d : J \to \mathbb{R}_+$. We write $\Sigma_{P,J}$ for the set of all actions over ports P and jobs J. We call the action $[\emptyset, \mathbf{0}]$, with $\mathbf{0}(x) = 0$ for all $x \in J$, the silent action. A word over P and J is a finite sequence $u \in \Sigma_{P,J}^*$ of actions over P and J.

Definition 7.1.5 (Trace semantics)**.** Let $A = (Q, P, J, I, \rightarrow, \phi_0, q_0)$ be a work automaton. A run r of A over a word $([N_i, d_i])_{i=1}^n \in \Sigma_{P,J}^*$ is a path

$$r \ : \ (p_0, q_0) \xrightarrow{N_1} \xrightarrow{d_1} s_1 \quad \cdots \quad s_{n-1} \xrightarrow{N_n} \xrightarrow{d_n} s_n$$

in $[\![A]\!]$, with $p_0 \models \phi_0 \wedge I(q_0)$. The language $L(A) \subseteq \Sigma_{P,J}^*$ of A is the set of all words u for which there exists a run of A over u.

Example 7.1.2. The language of the process A_i in Figure 7.1(a) trivially contains the empty word, and the word $u = [\emptyset, \mathbf{1}][\{a\}, \mathbf{1}][\{b\}, \mathbf{1}]$, where $\mathbf{1}(x_i) = 1$. Using Definitions 7.1.3 and 7.1.5, we conclude that $v = [\emptyset, \mathbf{1}][\{a\}, \mathbf{1}][\{b\}, \mathbf{0.5}][\emptyset, \mathbf{0.5}]$, with $\mathbf{0.5}(x_i) = 0.5$, is also accepted by A_i. Note that we can obtain v from u by splitting $[\{b\}, \mathbf{1}]$ into $[\{b\}, \mathbf{0.5}][\emptyset, \mathbf{0.5}]$. ◊

7.1.3 Weak simulation

Different work automata may have similar observable behavior. In this section, we define *weak simulation* as a formal tool to show their similarity. Intuitively, a weak simulation between two work automata A and B can be seen as a map that transforms any run of A into a run of B with identical observable behavior.

Following Milner [Mil89], we define a new transition relation, $\Rightarrow$, on the operational semantics $[\![A]\!]$ of a work automaton A that 'skips' silent steps.

Definition 7.1.6 (Weak transition relation)**.** For any two configurations s and t in $[\![A]\!]$, and any $a \in \mathbb{R}_+^J \cup 2^P$ we define $s \overset{a}{\Rightarrow} t$ if and only if either

1. $a = \emptyset$ and $s \ (\overset{\emptyset}{\rightarrow})^* \ t$; or
2. $a \in 2^P \setminus \{\emptyset\}$ and $s \overset{\emptyset}{\Rightarrow} s' \overset{a}{\rightarrow} s'' \overset{\emptyset}{\Rightarrow} t$; or
3. $a \in \mathbb{R}_+^J$, $s \overset{\emptyset}{\Rightarrow} s_1 \xrightarrow{c_1 \cdot a} t_1 \overset{\emptyset}{\Rightarrow} s_2 \ \cdots \ t_{n-1} \overset{\emptyset}{\Rightarrow} s_n \xrightarrow{c_n \cdot a} t_n \overset{\emptyset}{\Rightarrow} t$, and $\sum_{i=1}^n c_i = 1$,

with $n \geq 1$, s_i, t_i configurations in $[\![A]\!]$, $c_i \in [0,1]$, $(c_i \cdot a)(x) = c_i \cdot a(x)$, for all $x \in J$ and all $1 \leq i \leq n$.

Definition 7.1.7 (Weak simulation)**.** Let $A_i = (Q_i, P, J, I_i, \rightarrow_i, \phi_{0i}, q_{0i})$, for $i \in \{0,1\}$ be two work automata, and let $\preceq \subseteq (\mathbb{R}_+^J \times Q_0) \times (\mathbb{R}_+^J \times Q_1)$ be a binary relation over configurations of A_0 and A_1. Then, $\preceq$ is a weak simulation of A_0 in A_1 (denoted as $A_0 \preceq A_1$) if and only if

1. $p_{00} \models \phi_{00} \wedge I_0(q_{00})$ implies $(p_{00}, q_{00}) \preceq (p_{01}, q_{01})$, with $p_{01} \models \phi_{01} \wedge I_1(q_{01})$;
2. $s \preceq t$ and $s \overset{a}{\rightarrow} s'$, with $a \in \mathbb{R}_+^J \cup 2^P$, implies $t \overset{a}{\Rightarrow} t'$ and $s' \preceq t'$, for some t'.

We call $\preceq$ a weak bisimulation if and only if $\preceq$ and its inverse $\preceq^{-1} = \{(t,s) \mid s \preceq t\}$ are weak simulations. We call A_0 and A_1 weakly bisimilar (denoted as $A_0 \approx A_1$) if and only if there exists a weak bisimulation between them.

7.1.4 Composition

Thus far, our examples used work automata to define the exact behavior of a single job (or just a protocol L in Figure 7.1(b)). We now show that work automata are expressive enough to define the behavior of multiple jobs simultaneously. To this end, we define a product operator $\times$ on the class of all work automata. Before we turn to the definition, we first introduce some notation. For $i \in \{0,1\}$, let $A_i = (Q_i, P_i, J_i, I_i, \rightarrow_i, \phi_{0i}, q_{0i})$ be a work automaton and let $\tau_i = (q_i, N_i, w_i, R_i, q_i') \in \rightarrow_i$ be a transition in A_i. We say that τ_0 and τ_1 are *composable* (denoted as $\tau_0 \frown \tau_1$) if and only if $N_0 \cap P_1 = N_1 \cap P_0$. If $\tau_0 \frown \tau_1$, then we write $\tau_0 \mid \tau_1 = ((q_0, q_1), N_0 \cup N_1, w_0 \wedge w_1, R_0 \cup R_1, (q_0', q_1'))$ for the *composition* of τ_0 and τ_1.

Definition 7.1.8 (Composition)**.** Let $A_i = (Q_i, P_i, J_i, I_i, \rightarrow_i, \phi_{0i}, q_{0i})$, $i \in \{0,1\}$, be two work automata. We define the composition $A_0 \times A_1$ of A_0 and A_1 as the work automaton $(Q_0 \times Q_1, P_0 \cup P_1, J_0 \cup J_1, I_0 \wedge I_1, \rightarrow, \phi_{00} \wedge \phi_{01}, (q_{00}, q_{01}))$, where $\rightarrow$ is the smallest relation that satisfies:

$$\frac{i \in \{0,1\},\ \tau_i \in \rightarrow_i,\ \tau_{1-i} \in \rightarrow_{1-i} \cup \{(q, \emptyset, \top, \emptyset, q) \mid q \in Q_{1-i}\},\ \tau_0 \frown \tau_1}{\tau_0 \mid \tau_1 \in \rightarrow}$$

By means of the composition operator in Definition 7.1.8, we can construct large work automata by composing smaller ones. The following lemma shows that the composite work automaton does not depend on the order of construction.

Lemma 7.1.1. *$(A_0 \times A_1) \times A_2 \approx A_0 \times (A_1 \times A_2)$, $A_0 \times A_1 \approx A_1 \times A_0$, and $A_0 \times A_0 \approx A_0$, for any three work automata A_0, A_1, and A_2.*

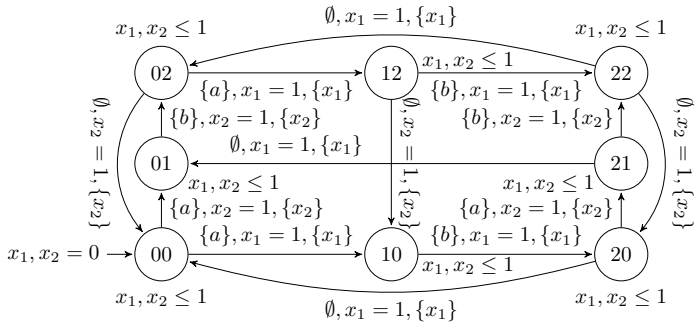


Figure 7.3: The complete application $M = L \times A_1 \times A_2$. In state q_1q_2, lock L is in state $(-1)^{q_1+q_2+1}$ and process A_i is in state q_i.

Example 7.1.3. Consider the work automata from Example 7.1.1. The behavior of the application is the composition M of the two processes A_1 and A_2 and the lock L. Figure 7.3 shows the work automaton $M = L \times A_1 \times A_1$. Each state-invariant equals $\top \wedge x_1 \leq 1 \wedge x_2 \leq 1$. The competition for the lock is visualized by the branching at the initial state 00. ♢

7.1.5 Hiding

Given a work automaton A and a port a in the interface of A, the *hiding* operator $A \setminus \{a\}$ removes port a from the interface of A. As a consequence, the hiding operator removes every occurrence of a from the synchronization constraint N of every transition $(q, N, w, R, q') \in \rightarrow$ by transforming N to $N \setminus \{a\}$. In case N becomes empty, the resulting transition becomes *silent*. If, moreover, the source and the target states of a transition are identical, we call the transition *idling*.

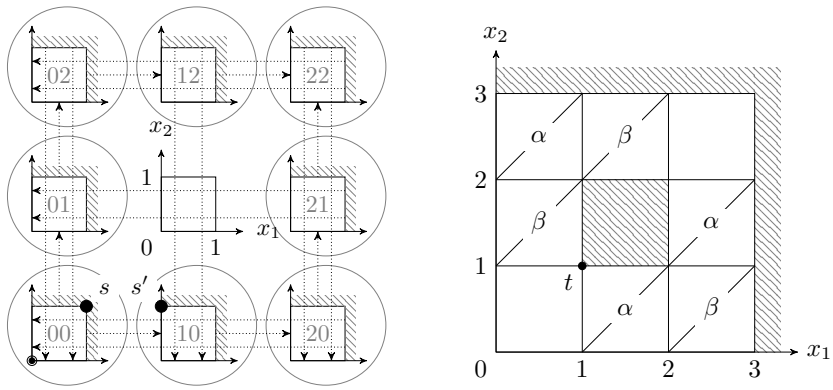
Definition 7.1.9 (Hiding)**.** Let $A = (Q, P, J, I, \rightarrow, \phi_0, q_0)$ be a work automaton, and $M \subseteq P$ a set of ports. We define $A \backslash M$ as the work automaton $(Q, P \backslash M, J, \rightarrow_M, \phi_0, q_0)$, with $\rightarrow_M = \{(q, N \setminus M, w, R, q') \mid (q, N, w, R, q') \in \rightarrow\}$.

Lemma 7.1.2. *Hiding partially distributes over composition: $M \cap P_0 \cap P_1 = \emptyset$ implies $(A_0 \times A_1) \setminus M \approx (A_0 \setminus M) \times (A_1 \setminus M)$, for any two work automata A_0 and A_1 with interfaces P_0 and P_1, respectively.*

Example 7.1.4. Consider the work automaton M in Figure 7.3. Work automaton $M' = M \setminus \{a, b\}$ is M where every occurrence of $\{a\}$ or $\{b\}$ is substituted by $\emptyset$. ♢

7.2 State Space Minimization

The composition operator from Definition 7.1.8 may produce a large complex work automaton with many different states. In this section, we investigate if, and how, a set of states in a work automaton can be merged into a single state, without breaking its semantics. In Section 7.2.1, we present by means of an example the basic idea for our simplification procedures. We define in Section 7.2.2 a *translation* operator that removes unnecessary resets from transitions. We define in Section 7.2.3



(a) $[\![M']\!]$ (b) $[\![K]\!]$

Figure 7.4: Graphical representation (a) of semantics $[\![M']\!]$ of the work automaton M' in Example 7.1.4, where white regions represent state-invariants, and (b) result after gluing the regions in(a). Starting in a configuration below line α and above line β, parallel execution of x_1 and x_2 never blocks on lock L.

a *contraction* operator that identifies different states in a work automaton. We show that translation and contraction are correct by providing weak simulations between their pre- and post-operation automata.

7.2.1 Gluing

The following example illustrates an intuitive *gluing procedure* that relates the product work automaton M in Figure 7.3 to the punctured square in Figure 7.4(b). Formally, we define the gluing procedure as the composition of translation (Section 7.2.2) and contraction (Section 7.2.3).

Example 7.2.1 (Gluing)**.** Consider the work automaton M' in Example 7.1.4 that describes the mutual exclusion protocol for two processes. Our goal is to simplify M' to a work automaton K that simulates M'. To this end, we introduce in Figure 7.4(a) a finite representation of the infinite semantics $[\![M']\!]$ of M', based on the geometric interpretation of progress discussed in Section 7.1.2. For any given state q of M', the state-invariant $I(q) = x_1 \leq 1 \wedge x_2 \leq 1$ is depicted in Figure 7.4(a) as a region in the first quadrant of the plane. Each configuration (p, q) of M' corresponds to a point in one of these regions: q determines its corresponding region wherein point p resides. Each transition of M' is shown in Figure 7.4(a) as a dotted arrow from the border of one region to that of another region. We refer to these dotted arrows as *jumps*. A jump λ from a region R of state q to another region R' of state q' represents infinitely many transitions from configurations (p, q) to configurations (p', q'), for all p and p', as permitted by the semantics $[\![M']\!]$. By the job constraint of the transition corresponding to λ, p and p' must lie on the borders of R and R', respectively, that are connected by λ.

From a topological perspective, a jump from one region to another can be viewed as 'gluing' the source and target configuration of that jump. We can glue any two

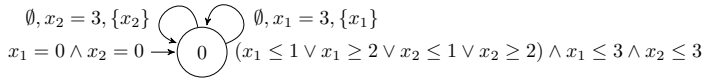


Figure 7.5: Work automaton K that corresponds to Figure 7.4(b).

regions in Figure 7.4(a) together by putting regions (i.e., state-invariants) of the source and the target states side by side to form a single state with a larger region. Each jump in Figure 7.4(a) from a source to a target state corresponds to an idling transition (c.f., rule Equation (S3) in Definition 7.1.3) within a single state. When we apply this gluing procedure in a consistent way to every jump in Figure 7.4(a), we obtain a single state work automaton K that is defined by a single large region, as shown in Figure 7.4(b). Figure 7.5 shows the actual work automaton that corresponds to this region. Note that the restart transition allows the state of progress to jump in Figure 7.4(a) from configuration $((x, 1), i2)$ to $((x, 0), i0)$ and from configuration $((1, y), 2j)$ to $((0, y), 0j)$, for all $x, y \in [0, 1]$ and $i, j \in \{0, 1, 2\}$. Thus, the restart transition identifies opposite boundaries in Figure 7.4(b), turning the punctured square into a torus. $\Diamond$

The next example shows that the geometric view of the semantics of the work automaton in Example 7.2.1 reveals some interesting behavioral properties of M'.

Example 7.2.2. Consider the mutual exclusion protocol in Example 7.1.1. Is it possible to find a configuration such that parallel execution of jobs x_1 and x_2 (at identical speeds) never blocks, even temporarily, on lock L? It is not clear from the work automata in Figure 7.1 (or in their product automaton as, e.g., in Figure 7.3) whether such a non-blocking execution exists. Since only one process can acquire lock L, the execution that starts from the initial configuration blocks after one unit of work. However, using the geometric perspective offered by Figure 7.4(b) and the fact that a parallel execution of jobs x_1 and x_2 at identical speeds correspond to a diagonal line in this representation, it is not hard to see that any execution path below line α and above line β is non-blocking. $\Diamond$

Regions of lock-free execution paths as revealed in Example 7.2.2 are interesting: if some mechanism (e.g., higher-level semantics of the application or tailor-made scheduling) can guarantee that execution paths of an application remains contained within such lock-free regions, then their respective locks can be safely removed from the application code. With or without such locks in an application code, a scheduler cognizant of such lock-free regions can improve resource utilization and performance by regulating the execution of the application such that its execution path remains in a lock-free region.

Example 7.2.3 (Correctness)**.** Let M' be the work automaton in Example 7.1.4, and K the work automaton in Figure 7.5. We denote a configuration of M' as a tuple $(p_1, p_2, q_0, q_1, q_2)$, where $p_i \in \mathbb{R}_+$ is the state of progress of job x_i, for $i \in \{0, 1\}$, and $(q_0, q_1, q_2) \in \{-, +\} \times \{0, 1, 2\}^2$ is the state of M'. We denote a configuration of K as a tuple $(p_1, p_2, 0)$, where $p_i \in \mathbb{R}_+$ is the state of progress of job x_i, for $i \in \{0, 1\}$. The binary relation $\preceq$ over configurations of M' and

(a) A (b) B

Figure 7.6: Shifting state-invariant $x \leq 1$ of state 1 in A by one unit.

K defined by $(p_1, p_2, q_0, q_1, q_2) \preceq (q_1 + p_1, q_2 + p_2, 0)$, for all $0 \leq p_i \leq 1$ and $(q_0, q_1, q_2) \in \{-, +\} \times \{0, 1, 2\}^2$, is a weak simulation of M' in K.

Note that $\preceq^{-1}$ is not a weak simulation of K in M' due to branching. Consider the configurations $s = (1, 1, -, 0, 0)$ and $s' = (0, 1, +, 1, 0)$ of M', and $t = (1, 1, 0)$ of K (cf., Figures 7.4(a) and 7.4(b)). While in configuration t job x_2 can make progress, execution of x_2 is blocked at s' because process A_1 has obtained the lock. Since $s' \preceq t$, we conclude that $\preceq^{-1}$ is not a weak simulation of K in M'.

Fortunately, we can still prove that K is a correct simplification of M by transforming $\preceq^{-1}$ into a weak simulation. Intuitively, such transformation remove pairs like $(t, s') \in \preceq^{-1}$. We make this argument formal in Section 7.2.3. ◊

As illustrated in Example 7.2.2, gluing can reveal interesting and useful properties of an application. To formalize the gluing procedure, we define two operators on work automata. The main idea is to transform a given work automaton A_1 into an equivalent automaton A_2, such that (almost) any step $(p_1, q_1) \xrightarrow{\emptyset} (p_1', q_1')$ in $[\![A_1]\!]$ corresponds with an *idling* step $(p_2, q_2) \xrightarrow{\emptyset} (p_2', q_2')$ in $[\![A_2]\!]$, i.e., a step with $p_2' = p_2$ and $q_2' = q_2$. To achieve this correspondence, we define a translation operator that ensures $p_2' = p_2$, and a contraction operator that ensures $q_2' = q_2$.

7.2.2 Translation

In this section, we define the translation operator that allows us to remove resets of jobs from transitions. The following example shows that removal of job resets can be compensated by shifting the state-invariant of the target state.

Example 7.2.4 (Shifting)**.** Suppose we remove the reset of job x on the transition of work automaton A in Figure 7.6(a). If we fire the transition at $x = a \leq 1$, then the state of progress of x in state 1 equals a instead of 0. We can correct this error by *shifting* the state-invariant of 1 by a, for every $a \leq 1$. We, therefore, transform the state-invariant of 1 into $x \leq 2$ (see Figure 7.6(b)). ◊

The transformation of work automata in Example 7.2.4 suggests a general translation procedure that, intuitively,(1) shifts each state-invariant $I(q)$, $q \in Q$, along the solutions of some job constraint $\theta(q) \in B(J)$, and(2) removes for every transition $\tau = (q, N, w, R, q')$ some resets $\rho(\tau) \subseteq J$ from R.

Definition 7.2.1 (Shifts)**.** A shift on a work automaton $(Q, P, J, I, \longrightarrow, \phi_0, q_0)$ is a tuple (θ, ρ) consisting of a map $\theta : Q \to B(J)$ and a map $\rho : \longrightarrow \to 2^J$.

We define how to shift state-invariants along the solutions of a job constraint.

Definition 7.2.2. Let $\phi, \theta \in B(J)$ be two job constraints with free variables among $\mathbf{x} = (x_1, \ldots, x_n)$, $n \geq 0$. We define the shift $\phi \uparrow \theta$ of ϕ along (the solutions of) θ as any job constraint equivalent to $\exists \mathbf{t}(\phi(\mathbf{x} - \mathbf{t}) \wedge \theta(\mathbf{t}))$.

Lemma 7.2.1. *$\uparrow$ is well-defined: for all $\phi, \theta \in B(J)$ there exists $\psi \in B(J)$ such that $\exists \mathbf{t}(\phi(\mathbf{x} - \mathbf{t}) \wedge \theta(\mathbf{t})) \equiv \psi$.*

We use a shift (θ, ρ) to translate guards and invariants along the solutions of job constraint θ and to remove resets occurring in ρ:

Definition 7.2.3 (Translation)**.** Let $\sigma = (\theta, \rho)$ be a shift on a work automaton $A = (Q, P, J, I, \to, \phi_0, q_0)$. We define the translation $A \uparrow \sigma$ of A along the shift σ as the work automaton $(Q, P, J, I_\sigma, \to_\sigma, \phi_0 \uparrow \theta(q_0), q_0)$, with $I_\sigma(q) = I(q) \uparrow \theta(q)$ and $\to_\sigma = \{(q, N, w \uparrow \theta(q), R \setminus \rho(\tau), q') \mid \tau = (q, N, w, R, q') \in \to\}$.

Lemma 7.2.2. *If $\theta \in B(J)$ has a unique solution $\delta \models \theta$, then $p + \delta \models \phi \uparrow \theta$ implies $p \models \phi$, for all $p \in \mathbb{R}_+^J$ and $\phi \in B(J)$.*

Theorem 7.2.3. *If $p \models w \wedge I(q)$ and $\delta \models \theta(q)$ implies $(p + \delta)[R \setminus \rho(\tau)] - p[R] \models \theta(q')$, for every transition $\tau = (q, N, w, R, q')$ and every $p, d \in \mathbb{R}_+^J$, then $A \preceq A \uparrow \sigma$. If, moreover, $\theta(q)$ has for every $q \in Q$ a unique solution, then $A \approx A \uparrow \sigma$.*

For at transition $\tau = (q, N, w, R, q')$, suppose $\theta(q)$ and $\theta(q')$ define unique solutions δ and δ', respectively. If σ eliminates job $x \in R$ (i.e., $x \in \rho(\tau)$), then $p(x) + \delta(x) = \delta'(x)$, for all $p \models w \wedge I(q)$. Thus, $w \wedge I(q)$ must imply $x = \delta'(x) - \delta(x)$, which seems a strong assumption. For a deterministic application, however, it makes sense to have only equalities in transition guards. In this case, a transition is enabled only when a job finishes some fixed amount of work, which corresponds to having only equalities in transition guards.

Example 7.2.5. Let M' be the work automata in Example 7.1.4, $\sigma = (\delta, \rho)$ the shift defined by $\theta(q) := x_1 = q_1 \wedge x_2 = q_2$, and $\rho(\tau) = R_\tau$. Theorem 7.2.3 shows that $M' \uparrow \sigma$ and M' are weakly bisimilar. $\Diamond$

7.2.3 Contraction

In this section, we define a contraction operator that merges different states into a single state. To determine which states merge and which stay separate, we use an equivalence relation $\sim$ on the set of states Q.

Definition 7.2.4 (Kernel)**.** A kernel of a work automaton A is an equivalence relation $\sim \subseteq Q \times Q$ on the state space Q of A.

Recall that an *equivalence class* of a state $q \in Q$ is defined as the set $[q] = \{q' \in Q \mid q \sim q'\}$ of all $q' \in Q$ related to q. The *quotient set* of Q by $\sim$ is defined as the set $Q/\!\sim = \{[q] \mid q \in Q\}$ of all equivalence classes of Q by $\sim$. By transitivity, distinct equivalence classes are disjoint and $Q/\!\sim$ partitions Q.

Definition 7.2.5 (Contraction)**.** The contraction $A/\!\sim$ of a work automaton $A = (Q, P, J, I, \to, \phi_0, q_0)$ by a kernel $\sim$ is defined as $(Q/\!\sim, P, J, I', \to', \phi_0, [q_0])$, where $\to' = \{([q], N, w, R, [q']) \mid (q, N, w, R, q') \in \to\}$ and $I'([q]) = \bigvee_{\tilde{q} \in [q]} I(\tilde{q})$.

The following results provides sufficient conditions for preservation of weak simulation by contraction. The relation $\preceq$ defined by $(p,[q]) \preceq (p,q)$, for all $(p,q) \in \mathbb{R}_+^J \times Q$, is not a weak simulation of $A/\!\sim$ in A. As indicated in Example 7.2.3, we can restrict $\preceq$ and require only $(p,[q]) \preceq (p,\alpha(p,[q]))$, for some *section* α.

Definition 7.2.6 (Section)**.** A section is a map $\alpha : \mathbb{R}_+^J \times Q/\!\sim \to Q$ such that for all $q, q' \in Q$ and $p, d \in \mathbb{R}_+^J$

1. $p \models I'([q])$ implies $p \models I(\alpha(p,[q]))$;
2. $q \sim \alpha(p,[q])$;
3. $p \models \phi_0 \wedge I(q_0)$ implies $\alpha(p,[q_0]) = q_0$;
4. $(p,[q]) \xrightarrow{N} (p',[q'])$ implies $(p,\alpha(p,[q])) \stackrel{N}{\Rightarrow} (p',\alpha(p',[q']))$;
5. $(p,q) \xrightarrow{d} (p+d,q)$ implies $(p,\alpha(p,[q])) \stackrel{d}{\Rightarrow} (p+d,\alpha(p+d,[q]))$.

In contrast with conditions (1), (2), and (3) in Definition 7.2.6, conditions (4) and (5) impose restrictions on the contraction $A/\!\sim$. These restrictions allow us to prove, with the help of the following lemma, weak simulation of $A/\!\sim$ in A.

Lemma 7.2.4. *If* $(p,[q]) \xrightarrow{d} (p+d,[q])$*, then there exist* $k \geq 1$*,* $0 = c_0 < \cdots < c_k = 1$ *and* $q_1, \ldots, q_k \in [q]$ *such that* $p + c \cdot d \models I(q_i)$*, for all* $c \in [c_{i-1}, c_i]$ *and* $1 \leq i \leq k$*.*

Theorem 7.2.5. $A \preceq A/\!\sim$*; and if there exists a section* α*, then* $A/\!\sim \preceq A$*.*

In our concluding example below, we revisit our intuitive gluing procedure motivated in Section 7.2.1 to show how the theory developed in Sections 7.2.2 and 7.2.3 formally supports our derivation of the geometric representation of $[\![K]\!]$ from $[\![M']\!]$ and implies the existence of mutual weak simulations between K and M'.

Example 7.2.6. Consider the work automaton $M' \uparrow \sigma$ from Example 7.2.5, and let $\sim$ be the kernel that relates all states of $M' \uparrow \sigma$. The contraction $(M' \uparrow \sigma)/\!\sim$ results in K, as defined in Example 7.2.1 (modulo some irrelevant idling transitions). Define $\alpha(p,[(q_1,q_2)]) = \min H$, where $H = \{(q_1,q_2) \in \{0,1,2\}^2 \mid p \models I_\sigma(q_1,q_2)\}$ is ordered by $(q_1,q_2) \leq (q_1',q_2')$ iff $q_1 \leq q_1'$ and $q_2 \leq q_2'$. By Theorem 7.2.5, we have $M' \preceq K$ and $M \preceq M'$. By Example 7.2.3, M' and K are not weakly bisimilar. $\Diamond$

The work automaton in Figure 7.3 and the geometric representation of its infinite semantics in Figure 7.4(a), only indirectly define a mutual exclusion protocol in M'. By Example 7.2.6, we conclude that M' is weakly language equivalent to a much simpler work automaton K that explicitly defines a mutual exclusion protocol by means of its state-invariant. Having such an explicit dependency visible in a state-invariant, reveals interesting behavioral properties of M', such as existence of non-blocking paths. These observations may be used to generate schedulers that force the execution to proceed along these non-blocking paths, which would enable a lock-free implementation and/or execution.

7.3 Related work

Work automata without jobs correspond to *port automata* [KC09], which is a data-agnostic variant of *constraint automata* [BSAR06]. In a constraint automaton, each synchronization constraint $N \subseteq P$ is accompanied with a data constraint that interrelates the observed data d_a, at every port $a \in N$. Although it is straightforward to extend our work automata with data constraints, we refrain from doing so because our work focuses on synchronization rather then data-aware interaction. Hiding on constraint automata defined by Baier et al. in [BSAR06] essentially combines our hiding operator in Definition 7.1.9 with contraction from Theorem 7.2.5.

The syntax of work automata is similar to the syntax of *timed automata* [AD94]. Semantically, however, timed automata are different from work automata because jobs in a work automaton may progress independently (depending on whether or not they are scheduled to run on a processor), while clocks in a timed automaton progress at identical speeds. For the same reason, work automata differ semantically from *timed constraint automata* [ABdBR04], which is introduced by Arbab et al. for the specification of time-dependent connectors.

This semantic difference suggests that we may specify a concurrent application as a *hybrid automaton* [Hen00], which can be seen as a timed automaton wherein the speed of each clock, called a *variable*, is determined by a set of first order differential equations. Instead of fixing the speed of each process beforehand, via differential equations in hybrid automata, our scheduling approach aims to determine the speed of each process only after careful analysis of the application. Therefore, we do not use hybrid automata to specify a concurrent application

Weighted automata [DKV09] constitute another popular quantitative model for concurrent applications. Transitions in a weighted automaton are labeled by a weight from a given semiring. Although weights can define the workload of transitions, weighted automata do not show dependencies among different concurrent transitions, such as mutual exclusion [vGV97]. As a consequence, weighted automata do not reveal dependencies induced by a protocol like work automata do.

A geometric perspective on concurrency has already been studied in the context of *higher dimensional automata*, introduced by Pratt [Pra91] and Van Glabbeek [vG91]. This geometric perspective has been successfully applied in [vGV97] to find and explain an essential counterexample in the study of semantic equivalences [vG06], which shows the importance of their, and indirectly our, geometric perspective. A higher dimensional automaton is a geometrical object that is constructed by gluing hypercubes. Each hypercube represents parallel execution of tasks associated with each dimension. This geometrical view on concurrency allows inheritance of standard mathematical techniques, such as homology and homotopy, which leads to new methods for studying concurrent applications [GJ92, Gun01].

7.4 Discussion

We extended work automata with state-invariants and resets and provided a formal semantics for these work automata. We defined weak simulation of work automata and presented translation and contraction operators that can simplify work automata while preserving their semantics up to weak simulation. Although

translation is defined for any shift (θ, ρ), the conditions in Theorem 7.2.3 prove bisimulation only if θ has a unique solution. In the future, we want to investigate if this condition can be relaxed—and if so, at what cost—to enlarge the class of applications whose work automata can be simplified using our transformations.

Our gluing procedure in Example 7.2.1 associates a work automaton with a geometrical object, and Example 7.2.2 shows that this geometric view reveals interesting behavioral properties of the application, such as mutual exclusion and existence of non-blocking execution paths. This observation suggests our results can lead to smart scheduling that yields lock-free implementation and/or executions.

State-invariants and guards in work automata model the exact amount of work that can be performed until a job blocks. In practice, however, these exact amounts of work are usually not known before-hand. This observation suggests that the 'crisp' subset of the multidimensional real vector space defined by the state-invariant may be replaced by a density function. We leave the formalization of such stochastic work automata as future work.

Chapter 8

Protocol Scheduling

The work automata developed in Section 7.1 encode all relevant scheduling information of a concurrent application and its protocol. For optimal performance of a concurrent software, the scheduler must take these dependencies into account. However, operating systems schedulers are application-independent and remain oblivious to the dependency information inherent in a protocol, even if such information is available. At best, these schedulers detect consequential effects of a protocol, such as blocking on an I/O-operation or waiting for a lock.

In this chapter, we develop a scheduling framework that extracts all scheduling information from a given work automaton and produces an ideal scheduler for the given software[1]. The most straightforward implementation of this ideal scheduler is to replace the operating system scheduler with our ideal scheduler. However, this approach is not possible if the operating system runs multiple different applications, each of which requiring its own ideal scheduler.

We offer an alternative approach that implements the ideal scheduler without changing the operating system scheduler. The main idea is to exploit the duality between schedules and protocols mentioned in Chapter 1 and transform the ideal schedule into a scheduling protocol. Then, we refine the original protocol by composing it with the scheduling protocol. The refined protocol forces the operating system scheduler to closely follows the ideal schedule, which improves the performance of the concurrent application. The scheduling protocol exploits the fact that the operating system scheduler executes only processes that are not blocked or waiting. Hence, it can enforce a custom schedule by blocking all application processes, except for those that should run according to the application's desired schedule. We block a process by prolonging existing blocking operations like I/O operations or waiting for a lock. Therefore, our approach assumes that the application's desired custom schedule is non-preemptive.

We synthesize non-preemptive schedules for concurrent applications using algorithms for games on graphs (Section 8.1). A graph game is a two-player zero-sum game played by moving a token on a directed graph, wherein each vertex is owned by one of the players. Typically, the ownership of the nodes along a path through the graph alternates between the two players. If the token is at a vertex owned

[1]The work in this chapter is based on [DA21, DJA16]

by a player, this player moves the token along one of the outgoing edges to the next vertex, after which the process repeats. The winning condition determines the player that wins based on the resulting path of the token.

We represent the scheduling problem as a graph game between the scheduler and the application (Section 8.2). The scheduler selects the processes that can execute, and the application resolves possible non-determinism within the processes of the concurrent program. Game-theoretic machinery computes a strategy for the scheduler that optimizes the execution of the concurrent program (according to an objective function that embodies some desired performance measure). Here, we consider only the objective of maximizing throughput, but similar techniques can also optimize for other scheduling performance measures, like fairness, context-switches, or energy consumption.

We view the resulting non-preemptive scheduling strategy itself as a scheduling protocol, and we compose the original protocol with this scheduling protocol to obtain a composite, scheduled protocol (Section 8.3). To evaluate the effect of the restricted protocol on a practical situation, we implement a reference version and a scheduled version of a simple cyclo-static dataflow network. This network consists of four processes, called actors, that interact asynchronously via five buffers (Figure 8.1). A buffer can handle overflows by dropping items to match its capacity. We measure the throughput (i.e., the time between consecutive productions) of both versions of the program. The throughput of the reference version varies significantly, and is on average worse than the throughput of the scheduled version, which, moreover, shows only a small variation.

Finally, we conclude and point to future work in Section 8.5.

8.1 Graph games

Our scheduling framework uses algorithms for games on graphs to construct non-preemptive schedules [DJA16]. A graph game is a two-player game of infinite duration. The possible move sequences are characterized by a safety automaton:

Definition 8.1.1. A safety automaton is a tuple $(Q, \Sigma, \delta, q_0, F)$, with Q a finite set of states, Σ a finite set of moves, $\delta : Q \times \Sigma \to Q$ a transition function, $q_0 \in Q$ an initial state, and $F \subseteq Q$ a set of accepting states, such that for every state $q \in Q$, we have $q \in F$ if and only if $\delta(q, \sigma) \in F$, for some move $\sigma \in \Sigma$.

In other words, a safety automaton is a deterministic finite automaton (DFA), wherein accepting states cannot be reached from non-accepting states, and every accepting state has an accepting successor.

As usual, we extend the transition function δ to finite move sequences by defining $\delta(q, \lambda) = q$ and $\delta(q, s\sigma) = \delta(\delta(q, s), \sigma)$, for every state $q \in Q$, finite move sequence $s \in \Sigma^*$, move $\sigma \in \Sigma$, and empty sequence $\lambda \in \Sigma^*$.

Since we are interested in infinite games only, we define the accepted language of a safety automaton as follows:

Definition 8.1.2. The accepted language $L(A)$ of a safety automaton A is the set of infinite words $\sigma_1\sigma_2 \cdots \in \Sigma^\omega$, such that $\delta(q_0, \sigma_1 \cdots \sigma_n) \in F$, for all $n \geq 0$.

Instantiating Definition 8.1.2 for $n = 0$, we see that $\delta(q_0, \sigma_1 \cdots \sigma_n) = \delta(q_0, \lambda) = q_0 \in F$, which means that the accepted language is non-empty if and only if the initial state q_0 is accepting.

Consider a state $q \in Q$ of a safety automaton. While all moves are possible in q (the transition function is defined for all pairs of states and moves), not every move is enabled in the sense that it leads to an accepted word. In view of Definition 8.1.2, we consider the set $\Sigma_q = \{\sigma \mid \delta(q, \sigma) \in F\}$ of enabled moves at q. By Definition 8.1.1, Σ_q is non-empty, for accepting states $q \in F$.

Based on safety automata, we construct a graph game as follows:

Definition 8.1.3. A graph game is a tuple (A, C, W), with $A = (Q, \Sigma, \delta, q_0, F)$ a safety automaton, $C \subseteq Q$ a set of controlled states, and $W \subseteq \Sigma^\omega$ a winning condition.

A graph game (A, C, W) is played by two players (say Player 1 and Player 2) who take turns to move a token from state to state. We consider Player 1 a protagonist and Player 2 an antagonist. To simplify notation, we write $C_1 = C$ for the states controlled by Player 1, and we write $C_2 = Q \setminus C$ for the states controlled by Player 2. Initially, the token is in state $q_0 \in Q$. If the token is in state $q \in C_k$, with $k \in \{1, 2\}$, then Player k selects an enabled move $\sigma \in \Sigma_q$ and moves the token to state $\delta(q, \sigma)$. As a result, the token moves along a path

$$q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} q_2 \xrightarrow{\sigma_3} \cdots$$

through the safety automaton. Player 1 wins if the sequence $\sigma_1\sigma_2\sigma_3 \cdots \in \Sigma^\omega$ of moves is contained in the winning condition W. Otherwise, Player 2 wins.

A joint strategy (for both players) is a function $\zeta : \Sigma^* \to \Sigma$ that selects a move, for every finite move sequence. A strategy for Player $k \in \{1, 2\}$ is a function $\zeta_k : P_k \to \Sigma$, with $P_k = \{\sigma_1 \cdots \sigma_n \in \Sigma^* \mid \delta(q_0, \sigma_1 \cdots \sigma_n) \in C_k\}$, that selects a move, for every finite move sequence that leads to a state controlled by Player k. If Player k follows a strategy ζ_k, then the resulting move sequence is contained in the set

$$L(\zeta_k) = \{\sigma_1\sigma_2\sigma_3 \cdots \in L(A) \mid \zeta_k(\sigma_1 \cdots \sigma_n) = \sigma_{n+1} \text{ whenever defined}\}$$

of outcomes of the game that are ensured by following strategy s_k. A strategy ζ_1 for Player 1 is winning if $L(\zeta_1) \subseteq W$. That is, strategy ζ_1 ensures that the resulting move sequence is contained in W, irrespective of the moves of Player 2. Winning strategies for Player 2 are defined similarly. A strategy ζ_k for Player k is optimal, if it is winning or if no winning strategy for Player k exists.

Of course, for a given game, it is impossible that both players have a winning strategy. However, for general winning conditions, it is possible that neither player has a winning strategy. In this case, the game is not determined. Nevertheless, Martin proved [Mar75] that graph games are determined if the winning condition is a Borel set[2]. Unfortunately, the results by Martin are descriptive and do not suggest a practical algorithm to find a winning strategy.

For simpler winning conditions, such as the ratio objective, we do have algorithms that compute a winning strategy:

[2]A Borel set is a subset of Σ^ω obtained from languages of safety automata by repeated complements and countable intersections.

Definition 8.1.4. A ratio game is a graph game with a winning condition

$$W_{s/t\geq v} = \left\{ \sigma_1\sigma_2\sigma_3\cdots \in \Sigma^\omega \;\middle|\; \liminf_{n\to\infty} \frac{\sum_{i=1}^{n} s(\sigma_i)}{\sum_{i=1}^{n} t(\sigma_i)} \geq v \text{ and } t(\sigma_1) \neq 0 \right\}$$

for some functions $s : \Sigma \to \mathbb{Z}$ and $t : \Sigma \to \mathbb{N}_0$, and value $v \in \mathbb{Q}$.

To the best of our knowledge, ratio games are introduced by Bloem et al. for the synthesis of robust systems [BGHJ09]. The winning condition $W_{s/t\geq v}$ stipulates that, for every $\epsilon \geq 0$ there exists some time $N \geq 0$, such that for all $n \geq N$ after this time, the fraction $(\sum_{i=1}^{n} s(\sigma_i))/(\sum_{i=1}^{n} t(\sigma_i))$ is at most ϵ less than v.

If $t(\sigma) = 1$, for all moves $\sigma \in \Sigma$, then we obtain a mean-payoff game. Much research has been devoted to finding efficient algorithms that solve mean-payoff games. One of the best known algorithms for mean-payoff games is due to Brim et al. [BCD$^+$11]. Their solution for mean-payoff games generalize easily to ratio games. We provide a brief explanation of this algorithm, and refer to [BCD$^+$11] for full details.

A classical result by Ehrenfeucht and Mycielsky [EM79], called memoryless determinacy, states that there exists a positional optimal joint strategy for mean-payoff games (and ratio games).

Definition 8.1.5. A joint strategy $\zeta : \Sigma^* \to \Sigma$ is positional, if $\delta(q_0, s_1) = \delta(q_0, s_2)$ implies $\zeta(s_1) = \zeta(s_2)$, for all move sequences $s_1, s_2 \in \Sigma^*$.

A positional strategy depends only on the current state $\delta(q_0, s)$, instead of the full history $s \in \Sigma^*$. If both players follow some positional strategy, the outcome of the game is a path

$$q_0 \xrightarrow{\sigma_1} \cdots \to q_k \xrightarrow{\sigma_k} \cdots \to q_n \xrightarrow{\sigma_n} q_k \xrightarrow{\sigma_k} \cdots \tag{8.1}$$

in the safety automaton that ends with a cycle in the distinct states $q_k, \ldots, q_n$. The outcome of a ratio game is winning for a value $v = a/b \in \mathbb{Q}$ if and only if $(\sum_{i=k}^{n} s(\sigma_i))/(\sum_{i=k}^{n} t(\sigma_i)) \geq v = a/b$, which is equivalent to $\sum_{i=k}^{n} w(\sigma_i) \geq 0$, with $w(\sigma) = b \cdot s(\sigma) - a \cdot t(\sigma)$, for all moves $\sigma \in \Sigma$.

Brim et al. observed that positional winning strategies for Player 1 in a ratio game correspond with consistent valuations:

Definition 8.1.6. A valuation $f : Q \to \mathbb{N}_0 \cup \{\infty\}$ is consistent in state $q \in Q$ iff

1. $q \in C_1$ implies $f(q) + w(\sigma) \geq f(\delta(q, \sigma))$, for some move $\sigma \in \Sigma_q$, or
2. $q \in C_2$ implies $f(q) + w(\sigma) \geq f(\delta(q, \sigma))$, for every move $\sigma \in \Sigma_q$,

A valuation is consistent if it is consistent in every state.

Suppose that there exists a consistent valuation $f : Q \to \mathbb{N}_0 \cup \{\infty\}$. Repeated application of Definition 8.1.6 to the path in Equation (8.1) yields $f(q_k) + \sum_{i=1k}^{n} w(\sigma_i) \geq f(\delta(q_k, \sigma_1 \cdots \sigma_n)) = f(q_k)$. If $f(q_k) < \infty$ is finite, then the outcome is winning for Player 1.

Brim et al. suggest a value iteration method to find the smallest possible valuation (we compare valuations pointwise: $f \leq f'$ iff $f(q) \leq f'(q)$, for all states

Algorithm 3: Synthesis problem for ratio games.

Input : A ratio game $(A, C, W_{s/t \geq v})$, with $A = (Q, \Sigma, \delta, q_0, F)$, functions $s : \Sigma \to \mathbb{Z}$ and $t : \Sigma \to \mathbb{N}_0$, and a value $v = \frac{a}{b} \in \mathbb{Q}$.

Output: A the largest quasi-strategy $\zeta : Q \to 2^\Sigma$ that is winning.

1 **foreach** $\sigma \in \Sigma$ **do** $w(\sigma) \leftarrow b \cdot s(\sigma) - a \cdot t(\sigma)$;
2 **foreach** $q \in Q$ **do** $f(q) \leftarrow 0$;
3 **foreach** $q \in C$ **do** $c(q) \leftarrow |\{\sigma \in \Sigma_q \mid f(q) + w(\sigma) < f(\delta(q, \sigma))\}|$;
4 $B \leftarrow \sum_{q \in Q} \max(\{0\} \cup \{-w(q, \sigma) \mid \sigma \in \Sigma_q\})$;
5 $L \leftarrow \{q \in Q \mid f \text{ inconsistent in } q\}$;
6 **while** $L \neq \emptyset \neq \{q \in Q \mid f(q) = 0\}$ *and* $f(q_0) < \infty$ **do**
7 Pick $q \in L$, with $f(q)$ minimal;
8 $f_q \leftarrow f(q)$;
9 $f(q) \leftarrow \min\{n \in \{1, \ldots, B, \infty\} \mid f[q \mapsto n] \text{ consistent in } q\}$;
10 $L \leftarrow L \setminus \{q\}$;
11 **if** $q \in C$ **then** $c(q) \leftarrow |\{\sigma \in \Sigma_q \mid f(q) + w(\sigma) < f(\delta(q, \sigma))\}|$;
12 **foreach** (p, σ) *with* $\delta(p, \sigma) = q \neq p$ *and* $f(p) + w(\sigma) < f(q)$ **do**
13 **if** $p \in C$ **then**
14 **if** $f(p) + w(\sigma) \geq f_q$ **then** $c(p) \leftarrow c(p) - 1$;
15 **if** $c(p) = 0$ **then** $L \leftarrow L \cup \{p\}$;
16 **if** $p \in Q_0$ **then** $L \leftarrow L \cup \{p\}$;
17 **foreach** $q \in Q$ **do**
$\zeta(q) \leftarrow \{\sigma \in \Sigma_q \mid f(q) + w(\sigma) \geq f(\delta(q, \sigma)),$ and $f(q) < \infty\}$;

$q \in Q$). Our Algorithm 3 shows a variation of their algorithm with only a few minor, but novel, adjustments.

The first modification is on Lines 6 and 7. Let $a = \min_{q \in Q} f_*(q)$ be the smallest value of the smallest valuation f_*. If $a < \infty$, then the valuation $f' : Q \to \mathbb{N}_0$ defined as $f'(q) = f_*(q) - a$, for all $q \in Q$, is less than or equal to f_*. Minimality of f_* shows that $a = 0$, which means that there exists a state $q \in Q$ with $f_*(q) = 0$. We refer to $q \in Q$ with $f_*(q) = 0$ as a pivot state. If there are no more pivot states, we can terminate the value iteration.

The second (minor) modification is on Line 9, which becomes apparent for states $q \in Q$ with a negative self loop transition (i.e., some $\sigma \in \Sigma$ with $\delta(q, \sigma) = q$ and $w(\sigma) < 0$). While the original algorithm by Brim et al. repeatedly adds $-w(\sigma)$ to the valuation $f(q)$ at state q, Algorithm 3 immediately jumps to the smallest valuation that resolves the inconsistency at q.

Value problem For given functions s and t, we have a family $\{W_{s/t \geq v} \mid v \in \mathbb{Q}\}$ of winning conditions. Since Player 1 wishes to maximize the ratio between the cumulatives of s and t, it is natural to look for the largest value $v \in \mathbb{Q}$ for which there exists a winning strategy. This problem is known as the value-problem. The set of values that are winning for Player 1 is a half-open interval $(-\infty, v_*]$, with v_* the optimal value. Using Algorithm 3, we can test the query $v \geq v_*$ for any value $v \in \mathbb{Q}$. Hence, the value problem can be solved by a binary search. Comin and Rizzi [CR17] improved this idea by reusing results from earlier queries.

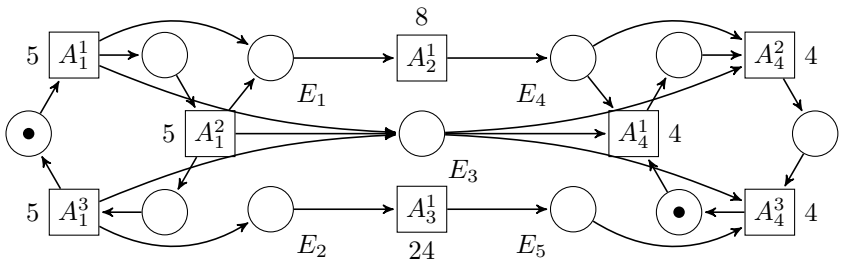


Figure 8.1: Petri net representation of a small cyclo-static dataflow graph.

8.2 Scheduling Game

We use the graph games from Section 8.1 to develop a game-theoretic framework for the synthesis of non-preemptive schedules. We assume that a concurrent program is given in the form of a *work automaton* $(Q, \Sigma, J, T, I, c_0)$ (cf., Section 7.1). Of course, we assume that the work automaton accurately models the real application. It may be nontrivial to verify whether or not the work automaton in fact models the application sufficiently accurately, but this concern is beyond the scope of our work on scheduling. In the worst case, one can ensure the application's compliance with the work automaton model by means of runtime verification.

As a running example, we formalize a simple cyclo-static dataflow network as a work automaton.

Example 8.2.1. Figure 8.1 shows the Petri net representation [Mur89] of a small cyclo-static dataflow (CSDF) graph from [Bam14, p. 29]. Recall that a Petri net consists of places (depicted as circles) that contain zero or more tokens, and transitions (depicted as rectangles). A transition firing consumes a single token from each of its input places and produces a token in each of its output places.

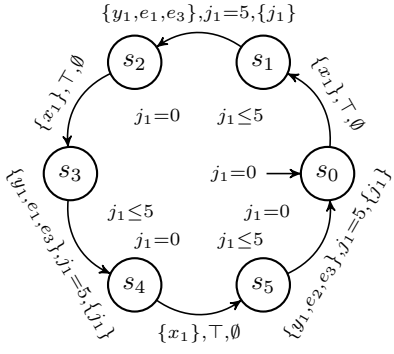
The system in Figure 8.1 consists of 4 actors (A_1 to A_4), which are connected via 5 buffers (E_1 to E_5). The original example in [Bam14] does not make any assumptions on the nature of the buffers: they can be FIFO, LIFO, lossy, or priority buffers. For our purpose, we assume that writing to a full buffer loses the written data. For simplicity, Figure 8.1 represents each buffer as a place in the Petri net. The losing behavior is made precise in the work automaton in Figure 8.2(e).

Each actor A_i, for $1 \leq i \leq 4$, cycles through a number of phases (hence the name cyclo-static). Figure 8.1 represents each phase as a transition A_i^j, for some index j, in the Petri net. Each phase of an actor requires time to execute. Although the actual times may vary, we consider only the worst-case execution times (WCET). We assume that all phases of a given actor have the same WCET. The integer value next to each transition in Figure 8.1 specifies its WCET. ◊

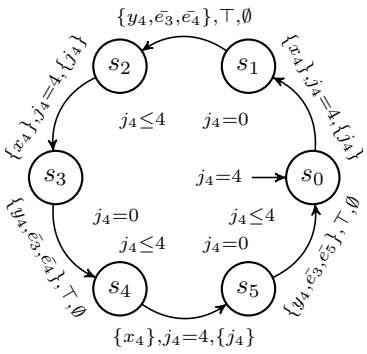
Although the functional behavior (as a Petri net) of the dataflow network in Figure 8.1 is very precise, its non-functional behavior (the timing of transitions) is still unclear. The following example makes this precise by providing a work automaton for each actor and buffer in Figure 8.1:

Example 8.2.2. Figure 8.2 shows the work automata that encode the informal

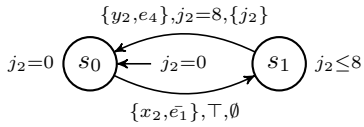
description of the behavior of the CSDF graph in Example 8.2.1. The work automaton for A_i, with $1 \leq i \leq 4$, has a real-valued job variable j_i that measures the progress of its respective actor. The initial condition $j_1 = 0$ in Figure 8.2(a) shows that actor A_1 must first perform 5 units of work on j_1 before it can produce on E_1 and E_3. In contrast, the initial condition $j_4 = 4$ in Figure 8.2(b) shows that actor A_4 can immediately consume tokens from E_3 and E_4. The work automata for A_2 and A_3 in Figures 8.2(c) and 8.2(d) are very similar. Each first consumes a datum from its input buffer, then executes for a given amount of time, and finally produces a datum in its output buffer. Figure 8.2(e) shows the work automaton for a buffer of capacity 4 (buffers of other capacity are defined similarly). The self-loop transition on state s_4 loses the data, if the buffer is full. ◊



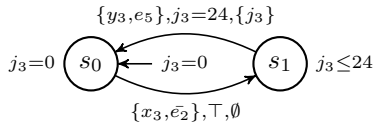
(a) A_1



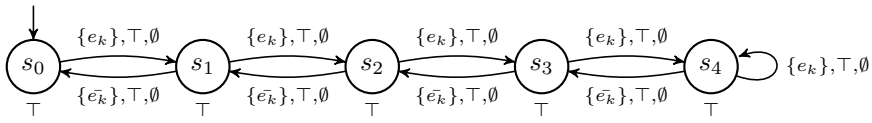
(b) A_4



(c) A_2



(d) A_3



(e) E_k

Figure 8.2: Work automata for the dataflow graph in Figure 8.1, without the idling transitions $(s_i, \emptyset, \top, \emptyset, s_i)$, for all states i. The self-loop transition in (e) loses the data, if the buffer is full.

By definition, non-preemptive scheduling relies on the cooperation of the application for managing its execution. We therefore, assume that the work automaton is cooperative:

Definition 8.2.1. A work automaton A is cooperative if and only if for every job

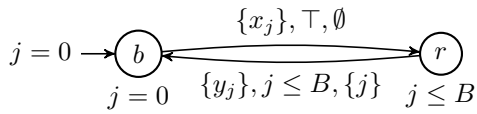


Figure 8.3: Work automaton specifying cooperative behavior of a job j. Signal x_j executes job j, and signal y_j yields job j. The bound $B \in \mathbb{N}_0$ ensures job j eventually yields.

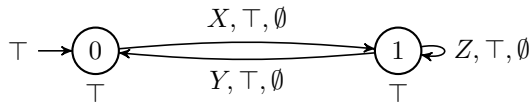


Figure 8.4: Work automaton G_A specifying the rules of the scheduling game for a work automaton A. The scheduler selects any signal $X \subseteq N$ that intersects $X_J = \{x_j \mid j \in J\}$. The application selects a signal $Y \subseteq N \setminus X_J$ that intersects $Y_J = \{y_j \mid j \in J\}$ or a signal $Z \subseteq N \setminus X_J$ that does not intersect Y_J.

$j \in J$ of A, we have that $A \bowtie A_j$ and A are identical up to renaming of states, where A_j is the work automaton in Figure 8.3.

To obtain a scheduling game from a given cooperative work automaton A, we compose A with the auxiliary work automaton G_A in Figure 8.4 that allows us to determine which player is to move and what moves are allowed by each player. In the composition $A \bowtie G_A$, every state is either controlled (if G_A is in state 0) or uncontrolled (if G_A is in state 1).

In the sequel, we assume without loss of generality that the work automaton G_A is already integrated in the specification of the cooperative work automaton:

Definition 8.2.2. A cooperative work automaton is playable iff $A \bowtie G_A$ and A are identical up to state renaming.

The semantics of a playable cooperative work automaton A from Definition 7.1.3 cannot be used directly in a graph game, because there are infinitely many ways to make progress via the d-transitions. Of course, not all progressions are equally likely to happen. In fact, if sufficiently many processors are available, we expect that all running processes make an equal amount of progress. We assume that these processes run uninterruptedly and at equal speeds. That is, we use job assignments $[S] : J \to \mathbb{R}$, for a subset of jobs $S \subseteq J$, such that $[S](j) = 1$ if $j \in X$, and $[S](j) = 0$, otherwise. We denote the expected transition in the semantics with a double arrow:

Definition 8.2.3. The expected semantics $[\![A]\!]_e$ of a work automaton A is the subgraph of $[\![A]\!]$ with the $\Rightarrow$ edges defined by the rules

$$\frac{c \xrightarrow{[S]} c' \qquad \text{if } c \xrightarrow{[S']} c' \text{ and } S' \neq S \text{ then } S \not\subseteq S'}{c \overset{[S]}{\Longrightarrow} c'} \quad \text{and} \quad \frac{c \xrightarrow{\sigma} c'}{c \overset{\sigma}{\Rightarrow} c'}$$

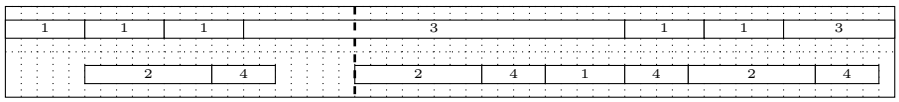


Figure 8.5: Generated schedule. The vertical dashed line indicates the start of the period.

The expected transition relation is just a subrelation of the real transition relation of the semantics of a work automaton. Note that there is no guarantee that the real execution follows the expected one. However, we show in Section 8.3 that deviations of real execution from expected execution do not cause deadlocks.

Lemma 8.2.1. *The expected semantics of a cooperative work automaton is finite.*

Proof. Since A is cooperative, the progress of every job is bounded by $B \in \mathbb{N}_0$ in every state (Figure 8.3). By a simple induction, it follows that the configurations of the expected semantics are contained in the finite set $Q \times \{0, \ldots, B\}^J$. □

The final ingredient for a scheduling game is an objective. The only restriction that we impose on a scheduling objective is that it must be expressible as a ratio objective $W_{s/t \geq v}$, in terms of some functions $s : M \to \mathbb{Z}$ and $t : M \to \mathbb{N}_0$.

Example 8.2.3. The composition of the work automata in Figures 8.2 and 8.4 yields a game graph. We maximize the throughput, which we define as the ratio between the number of productions and the number of time steps (ticks). We can count the productions by counting how often e_5 fires. Hence, we define

$$s(a_1 \cdots a_n) = |\{i \mid e_5 \in a_i \in \Sigma\}| \quad \text{and} \quad t(a_1 \cdots a_n) = |\{i \mid a_i \in \mathbb{R}_+^J\}|.$$

Algorithm 3 finds a subgame for which every play is of optimal throughput.

As the CSDF graph is a deterministic program, all non-determinism is controlled by the scheduler. Since all options ensure optimal throughput, we can resolve the non-determinism arbitrarily. We resolve non-determinism by preferring idling. The deterministic scheduling strategy for this example can be presented as a Gantt chart, shown in Figure 8.5. ♢

8.3 Protocol restriction

Consider a playable cooperative work automaton A from Section 7.1 and a non-preemptive scheduling strategy $\zeta : C \to \Sigma$ from Section 8.2, with C the set of configurations of the expected semantics $[\![A]\!]_e$ of A. Now we intend to implement the schedule ζ as a protocol $A(\zeta)$ that blocks precisely those jobs that are not supposed to make progress, such that the operating system scheduler has to closely follow the desired scheduling strategy.

The solution to ratio games in Algorithm 3 returns a subgraph of the original game graph. For a scheduling game, vertices are configurations of A, and the edge come from the expected transition relation. Hence, the resulting scheduling strategy can be easily transformed back into a work automaton.

Definition 8.3.1. The scheduling work automaton $A(\zeta)$ of a scheduling strategy ζ is defined as the tuple $(C, \Sigma, \emptyset, \to, I, c_0)$ with states $C = Q \times \mathbb{N}_0^J$, trivial invariant $I(c) = \top$ for all states $c \in C$, and transition relation defined by the rule

$$\frac{c \in C \text{ controlled} \qquad c \overset{a}{\Rightarrow} c' \qquad \zeta(c) \text{ defined implies } a = \zeta(c)}{c \xrightarrow{a,\top,\emptyset} c'}$$

We enforce the scheduling strategy by considering the composition $A \bowtie A(\zeta)$. The composition $A \bowtie A(\zeta)$ does not introduce any new, undesired executions, but merely restricts the composition to a subset of desired executions of A. By construction, A and $A(\zeta)$ synchronize only on signals, which agrees with the fact that the schedule ζ is non-preemptive.

For the construction of the scheduling game in Section 8.2, we assume that scheduled jobs run as expected, i.e., they run continuously and at constant speed. However, it is actually very likely that the actual execution deviates from the expected execution. A natural question is whether such deviations may confuse the scheduler enough to introduce deadlocks.

Theorem 8.3.1. *If A is a composition of simple work automata, then $A \bowtie A(\zeta)$ is deadlock free.*

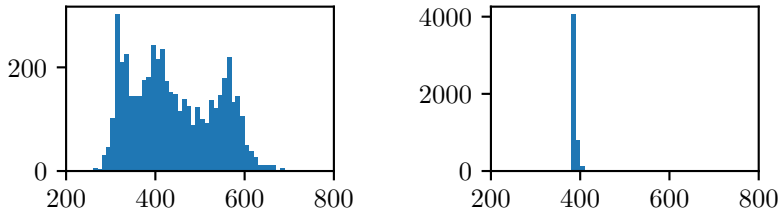
A simple work automaton is a work automaton with at most one job and no silent transitions that in every configuration can either make progress or fire a transition, but cannot do both. The work automata in Figure 8.2 are examples of simple work automata.

of Theorem 8.3.1. The state-space of $A \bowtie A(\zeta)$ is defined by tuples consisting of a state $q \in Q$ and a configuration (q', p') of $A(\zeta)$. Let $c = ((q, (q', p')), p) \in (Q \times (Q \times \mathbb{N}_0^J)) \times \mathbb{R}^J$ be a reachable configuration of $A \bowtie A(\zeta)$. The absence of silent transitions ensures that the scheduler $A(\zeta)$ knows the state of A (i.e, we have $q = q'$). The progress $p : J \to \mathbb{R}$ of the jobs may still be unknown (i.e., $p \neq p'$ is possible). By construction, there exists some expected execution of $A \bowtie A(\zeta)$ that passes through the same state q (although the progress may be different from p) and enables a transition $t = (q, \sigma, g, R, q')$. Since the work automaton for each job is simple, the guard g of transition t states that the progress of a subset of jobs is maximal (while the progress of other jobs is irrelevant). Hence, from c we can make sufficient progress to enable transition t, which implies that c is not a deadlock. □

Example 8.3.1. To evaluate the schedule in Figure 8.5, we implement the CSDF graph in Figure 8.1 in Python. We performed the experiment on a 64-bit Windows 10 Home Edition with a Intel® Core™ i7-7700HQ CPU at 2.80 GHz and 16 GB RAM. We executed the source code with a 64-bit Python 3.9.0 interpreter.

Appendix B shows the source code of the scheduled application. It is obtained manually, but mechanically, from the original source code in Appendix A by adding barrier synchronizations between the actors and a scheduler process. This scheduler process implements the non-preemptive schedule in Figure 8.5.

Figure 8.6 shows the histogram (with a bin-size of 10 ms) of the output of both versions of the program. We measure the throughput (i.e., the time between successive productions) of each version. Both the expected value and the standard



(a) Original (b) Scheduled

Figure 8.6: Throughput of original program (a) and scheduled program (b). The horizontal axis is the time in milliseconds between successive firings of A_4^3 (grouped in bins of 10 milliseconds), and the vertical axis is the frequency of each bin.

deviation of the two versions differ. The original version has an expected throughput of 441 ms with a standard deviation of 95 ms. The scheduled version has an expected throughput of 386 ms and a standard deviation of 6 ms.

The quality of the schedule alone does not explain the improvement of the expected throughput; the characteristics of the original protocol are the most important factor in this example. Recall that we use overflow buffers, which means that an actor loses its datum, if its respective buffer is full. If a datum is lost, all effort invested in its production is also lost. The general-purpose scheduler of the operating system is unaware of these losses. ◊

Although the game-theoretic framework in Section 8.1 aims to optimize the expected throughput, Example 8.3.1 shows that the most significant improvement in the scheduled protocol is in its impact on the standard deviation of the throughput. The time between successive productions is much more predictable for the scheduled version compared to the original version. Since we force the operating system scheduler to closely follow a fixed, deterministic schedule, predictability of the throughput is not surprising.

Predictable timing is a requirement for many systems. For example, a pacemaker must assist a patient's heart to beat at a regular and predictable rate, and a self-driving car must sense and analyze its environment at a predictable rate to avoid collisions. The results of Example 8.3.1 show that the unpredictable behavior of the scheduler of the operating systems can be made tightly predictable by restricting the protocol through composition with a scheduling protocol.

8.4 Related work

There is a wealth of literature on different models for concurrent software, ranging from the well-known Petri nets [Mur89], to the lesser-known higher-dimensional automata [Pra91, vG06].

Our implementation of the scheduler as a composition is very similar to the definition of a scheduler defined by Goubault [Gou95]. In his work, Goubault specifies the application as a higher dimensional automaton and views the scheduler

as a subautomaton. While our scheduling framework builds on graph games, his work depends on the solution of a particular problem on huge sparse matrices.

The work on (variants of) timed automata [AD90] is closely related to our work in Chapters 7 and 8. In fact, syntax of work automata is identical that of timed automata, and the clock variables in timed automata correspond naturally with the jobs in work automata. However, their semantics differs significantly: in a timed automaton all clocks progress at the same speed, while work automata do not make assumptions on the relative speeds of job progress.

Stopwatch automata [CL00] are one step closer to work automata, as in each state a clock is either running or paused. This feature allows stopwatch automata to use the clocks to measure the progress of jobs [AM02].

Uppaal Stratego [DJL+15] is a tool the analysis of stochastic priced timed games. Similar to our work, this tool uses strategies to achieve safety and performance. Uppaal Stratego enforces these strategies via parallel composition with the original automaton. In contrast with our work, Uppaal Stratego can handle stochastic environments by means of simulation and reinforcement learning. Uppaal Stratego's ideas complement our work. Currently, we use the algorithm for ratio games from Section 8.1 to solve the scheduling game in this chapter. However, replacing this algorithm with Uppaal Stratego's algorithms would provide the benefits of both approaches.

8.5 Discussion

Protocols contain valuable information indispensable for construction of optimal (non-preemptive) schedules for allocation of resources to execute a concurrent application. Exogenous languages like Reo express a protocol as an explicit software construct, which makes this scheduling information accessible. We express protocols together with their scheduling information in terms of the work automaton semantics of Reo. We construct a generic scheduling framework based on ratio games to find optimal non-preemptive schedules for an application defined as a work automaton. By composing such a scheduling protocol with the original protocol of an application, we obtain a composite scheduled protocol that forces generic operating system preemptive schedulers to closely follow the desired optimal schedule. The exogenous nature of Reo guarantees that the application code remains oblivious to the substitution of the composite scheduled protocol for its original protocol. An experiment shows that a scheduled version of a cyclo-static dataflow network (with the composite protocol) has higher and more predictable throughput compared to its original version.

Future work The algorithm by Brim et al. to solve ratio games requires the full state-space of an application. In view of the state-space explosion problem, we can use other schemes (like Monte-Carlo tree search) to find good schedules.

Although the work presented in our current chapter focuses on maximizing throughput, the ratio objective can express many other scheduling performance measures, like fairness, context-switches, or energy consumption. We intend to express these performance measures in terms of ratio objectives.

Chapter 9

Conclusion

9.1 Summary

With the advent of multicore processors [ABD+09] and data centers [Kan09] computer hardware has become increasingly parallel, which allows one to run multiple pieces of software at the same time on different machines. Programmers can enjoy the fruits of parallelism by partitioning software into relatively independent components and running these components simultaneously.

Of course, these components are not completely independent: they must cooperate to achieve the common goal of the complete software system. Such cooperation of components requires shared resources and coordination. As explained in Part I, coordination is best expressed in an explicit interaction protocol that clearly defines the interactions amongst all components in the software. An explicit interaction protocol not only improves code structure, but also enables automated analysis of the protocol [Klü12], and improved execution efficiency by compiler optimizations [Jon16]. In Chapter 2, we compare two coordination languages, BIP and Reo, and offer formal translations between them. Our comparison reveals differences between BIP and Reo with respect to composition and priority. The difference in composition leads to the proposal of a composition operator for data-sensitive BIP architectures.

We aim to narrow the 'priority gap' between BIP and Reo in Chapter 3 by expanding the theory of soft constraint automata with memory cells and bipolar preference values. These soft constraint automata offer a semantics for Reo wherein bipolar preference values can express a weak form of priority, such as context-sensitivity. Our approach to context-sensitivity differs significantly from other approaches (that do not model preferences as explicit values), such as connector coloring [CCA07], dual ports [JKA11], intentional automata [CNR11], augmented Büchi automata of records [IBC08], and guarded automata [BCS12]. Although we consider context-sensitivity in the realm of Reo, we stress that context-sensitivity is a fundamental concept that applies to languages other than Reo.

Part II of our thesis focuses on the generation of executable code for a given interaction protocol that merges its coordinated components into a single concurrent software. We temporarily ignore the coordinated components and study the

interaction protocol in isolation. The basis of any compiler is a well-defined input language. Many Reo tools use a plugin in the Eclipse editor as a graphical editor for Reo connectors. The graphical editor not only commits the user to the Eclipse editor, but also lacks important features such as parameter passing, iteration, recursion, or conditional construction of connectors. Some Reo tools therefore develop their own textual language, each having a specific Reo semantics in mind. In Chapter 4, we propose a textual language, called Treo, that does not commit to any specific semantics.

We use the freedom of the semantics in Treo to propose a new semantics that represents Reo connectors as temporal logic formulas called stream constraints. Such constraints can be written in may equivalent forms. Since composition is conjunction, the composite constraint grows linearly in size. However, compilation of such a conjunction is hard and requires a constrain solver (possibly at run time). Chapter 5 identifies the rule-based form as a form that balances the trade-off between composition and compilation of constraints. A rule in a rule-based form corresponds naturally to a loosely coupled thread or process, which makes compilation straightforward. The composition of rule-based constraints grows linearly most many practical cases (such as the $\mathsf{Alternator}_k$ in Figure 5.3), and grows exponentially only in the worst case. We implemented a Reo compiler based on stream constraints, and showed that it outperforms the state-of-the-art Reo compiler ([JKA17]).

Reflecting on our stream constraints, our main observation is the discrepancy with respect to concurrency between Reo connectors and their formal semantics [JA12]. While Reo connectors are considered concurrent, almost all semantics for Reo are expressed in an inherently sequential model (such as streams or automata). The encoding of Reo connectors into zero-safe (Petri) nets is the only exception. This mismatch results in ad hoc extraction of concurrency via techniques like 'synchronous region decomposition' [JCP16], 'local multiplication' [Jon16], or our rule-based form in Chapter 5. In Chapter 6, we propose an inherently concurrent semantics for Reo connectors as Multilabeled Petri nets, which turns the ad hoc extraction of concurrency from our rule-based form into the standard interpretation of concurrency in Petri nets. While the syntax and semantics of Petri nets is standard, our main contribution is a composition operator on Petri nets that mimics the composition in stream constraints. As a result, we obtain a simple expressive Reo semantics that can be effectively compiled into efficient code.

In Part III, we study the effect of the interaction protocol schedulability of the complete software (including the coordinated components). In order to compute high quality schedules for a software system, we must know how much processing time (or work) each component requires. To this end, we develop work automata in Chapter 7, which express, for a fixed number of components, how much work each component can/must do. We develop a gluing technique that allows us to minimize the state space of a work automaton to potentially a single state with a complex invariant. This invariant exposes mutual exclusion as holes in a higher-dimensional space. If a scheduler can avoid such holes, we can potentially drop the (costly) locks that prevent the application from transgressing into these holes.

Next, we use work automata to develop a game theoretic scheduling framework. In general, the scheduler decides which components can run, while the application decides how to resolve non-determinism during execution. In Chapter 8 we for-

malize this game as a two-player zero-sum game played on a graph, and slightly adapt existing solutions to compute a scheduling strategy for a small cyclo-static dataflow application that optimizes throughput.

The most straightforward solution to implement our synthesized strategy is to replace the default operating system scheduler with a custom application-specific scheduler. While this approach is possible, it is a non-trivial, costly task that requires administrative rights on the operating system. We present a novel alternative that implements the synthesized strategy in the software, while keeping the default operating system scheduler in place. First, we transform the resulting strategy into a scheduling protocol, and subsequently compose this scheduling protocol with the original protocol of the application. As a result, a general-purpose operating system scheduler (which schedules all non-blocked components in a round-robin fashion) will closely follow our optimal strategy. As a result, we avoid complex custom schedulers, while still obtaining our scheduling goals.

9.2 Future work

In this thesis, we demonstrated the practical use of the schedule-protocol duality from Chapter 1 by applying it to the special case of cyclo-static dataflow software. However, the applications of the schedule-protocol duality are not limited to this special case, and further research can study the implications of this duality on cyber-physical software and real-time systems. These more general systems often have real-time constraints and alternative objectives (other than throughput), which we do not consider in the current thesis.

Even if a program is represented as a small Petri net, its scheduling game can have a very large number of positions. The scheduling game of the cyclo-static dataflow program from Chapter 8 turned out to be just small enough to be solved on our personal laptop, but solving the scheduling game for larger programs requires superior hardware like a computer cluster. To handle even larger applications, we can improve the scalability of schedule synthesis by using different solvers that avoid the state-space explosion. One direction is to search for classes of programs that can be scheduled by existing powerful solvers (like ILP solvers, SAT solvers, or SMT solvers). Another direction is to develop scheduling heuristics that simplify scheduling synthesis at the cost of suboptimal solutions.

The work automata introduced in Chapter 7 encode all relevant scheduling information and we use them as input to our scheduling framework. In this thesis, we assumed that these work automata are given and put our focus on the resulting scheduling problem. Although the problem of finding the work automata of a given program is non-trivial, the search can be automated, if all necessary details (such as the exact machine instructions and the hardware architecture) are known. Even if not all necessary details are known, we can still approximate the size of the workloads through experimentation. The workload can then be represented as a distribution (rather than a single value), which can be encoded as a work automaton via non-deterministic transitions.

Bibliography

[AA16] Ramón Alonso-Sanz and Andy Adamatzky. Actin automata with memory. *Int. J. Bifurc. Chaos*, 26(1):1650019:1–1650019:14, 2016.

[ABB+14] Paul C. Attie, Eduard Baranov, Simon Bliudze, Mohamad Jaber, and Joseph Sifakis. A general framework for architecture composability. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, volume 8702 of *Lecture Notes in Computer Science*, pages 128–143. Springer, 2014.

[ABC+08] Farhad Arbab, Roberto Bruni, Dave Clarke, Ivan Lanese, and Ugo Montanari. Tiles for reo. In Andrea Corradini and Ugo Montanari, editors, *Recent Trends in Algebraic Development Techniques, 19th International Workshop, WADT 2008, Pisa, Italy, June 13-16, 2008, Revised Selected Papers*, volume 5486 of *Lecture Notes in Computer Science*, pages 37–55. Springer, 2008.

[ABD+09] Krste Asanovic, Rastislav Bodík, James Demmel, Tony M. Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David A. Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine A. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[ABdBR04] Farhad Arbab, Christel Baier, Frank S. de Boer, and Jan J. M. M. Rutten. Models and temporal logics for timed component connectors. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China*, pages 198–207. IEEE Computer Society, 2004.

[ABdBR07] Farhad Arbab, Christel Baier, Frank S. de Boer, and Jan J. M. M. Rutten. Models and temporal logical specifications for timed component connectors. *Softw. Syst. Model.*, 6(1):59–82, 2007.

[ÁBJ16] Erika Ábrahám, Marcello M. Bonsangue, and Einar Broch Johnsen, editors. *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, volume 9660 of *LNCS*. Springer, 2016.

[ACMM07] Farhad Arbab, Tom Chothia, Sun Meng, and Young-Joo Moon. Component connectors with qos guarantees. In Amy L. Murphy and Jan Vitek, editors, *Coordination Models and Languages, 9th International Conference, COORDINATION 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*, volume 4467 of *Lecture Notes in Computer Science*, pages 286–304. Springer, 2007.

[AD90] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In Mike Paterson, editor, *Proc. of ICALP*, volume 443 of *LNCS*, pages 322–335. Springer, 1990.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[AGK01] Nikolay A. Anisimov, Evgeniy A. Golenkov, and Dmitriy I. Kharitonov. Compositional petri net approach to the development of concurrent and distributed systems. *Programming and Computer Software*, 27(6):309–319, 2001.

[AM02] Yasmina Abdeddaïm and Oded Maler. Preemptive job-shop scheduling using stopwatch automata. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 113–126. Springer, 2002.

[AR02] Farhad Arbab and Jan J. M. M. Rutten. A coinductive calculus of component connectors. In Martin Wirsing, Dirk Pattinson, and Rolf Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, September 24-27, 2002, Revised Selected Papers*, volume 2755 of *Lecture Notes in Computer Science*, pages 34–55. Springer, 2002.

[Arb98] Farhad Arbab. What do you mean, coordination? In *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI*, pages 11–22, 1998.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Math. Struct. Comput. Sci.*, 14(3):329–366, 2004.

[Arb11] Farhad Arbab. Puff, the magic protocol. In Gul Agha, Olivier Danvy, and José Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, volume 7000 of *LNCS*, pages 169–206. Springer, 2011.

[Arb16] Farhad Arbab. Proper protocol. In Ábrahám et al. [ÁBJ16], pages 65–87.

[AS12] Farhad Arbab and Francesco Santini. Preference and similarity-based behavioral discovery of services. In Maurice H. ter Beek and Niels Lohmann, editors, *Web Services and Formal Methods - 9th International Workshop, WS-FM 2012, Tallinn, Estonia, September 6-7, 2012, Revised Selected Papers*, volume 7843 of *Lecture Notes in Computer Science*, pages 118–133. Springer, 2012.

[BAC+09] Tomás Barros, Rabéa Ameur-Boulifa, Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Behavioural models for distributed fractal components. *Ann. des Télécommunications*, 64(1-2):25–43, 2009.

[Bai05] Christel Baier. Probabilistic models for reo connector circuits. *J. Univers. Comput. Sci.*, 11(10):1718–1748, 2005.

[Bam14] Mohamed Ahmed Mohamed Bamakhrama. *On hard real-time scheduling of cyclo-static dataflow and its application in system-level design.* PhD thesis, Leiden University, 2014.

[BBKK09] Christel Baier, Tobias Blechmann, Joachim Klein, and Sascha Klüppelholz. A uniform framework for modeling and verifying components and connectors. In John Field and Vasco Thudichum Vasconcelos, editors, *Coordination Models and Languages, 11th International Conference, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5521 of *Lecture Notes in Computer Science*, pages 247–267. Springer, 2009.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India*, pages 3–12. IEEE Computer Society, 2006.

[BCD+11] Lubos Brim, Jakub Chaloupka, Laurent Doyen, Raffaella Gentilini, and Jean-François Raskin. Faster algorithms for mean-payoff games. *Formal Methods Syst. Des.*, 38(2):97–118, 2011.

[BCH+97] Christel Baier, Edmund M. Clarke, Vasiliki Hartonas-Garmhausen, Marta Z. Kwiatkowska, and Mark Ryan. Symbolic model checking for probabilistic processes. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings*, volume 1256 of *Lecture Notes in Computer Science*, pages 430–440. Springer, 1997.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in java. *Softw., Pract. Exper.*, 36(11-12):1257–1284, 2006.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[BCS12] Marcello M. Bonsangue, Dave Clarke, and Alexandra Silva. A model of context-dependent component connectors. *Sci. Comput. Program.*, 77(6):685–706, 2012.

[BdC92] Luca Bernardinello and Fiorella de Cindio. A survey of basic net models and modular net classes. In Rozenberg [Roz92], pages 304–351.

[BDH92] Eike Best, Raymond R. Devillers, and Jon G. Hall. The box calculus: a new causal algebra with multi-label communication. In Rozenberg [Roz92], pages 21–69.

[BG06] Stefano Bistarelli and Fabio Gadducci. Enhancing constraints manipulation in semiring-based formalisms. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 63–67. IOS Press, 2006.

[BGHJ09] Roderick Bloem, Karin Greimel, Thomas A. Henzinger, and Barbara Jobstmann. Synthesizing robust systems. In *Proc. of FMCAD*, pages 85–92. IEEE, 2009.

[BHM19] Simon Bliudze, Ludovic Henrio, and Eric Madelaine. Verification of concurrent design patterns with data. In Hanne Riis Nielson and Emilio Tuosto, editors, *Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*, volume 11533 of *Lecture Notes in Computer Science*, pages 161–181. Springer, 2019.

[bip16] BIP toolset, apr 2016.

[BK85] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.

[BKK14] Christel Baier, Joachim Klein, and Sascha Klüppelholz. Synthesis of reo connectors for strategies and controllers. *Fundam. Informaticae*, 130(1):1–20, 2014.

[BM97] Roberto Bruni and Ugo Montanari. Zero-safe nets, or transition synchronization made simple. *Electr. Notes Theor. Comput. Sci.*, 7:55–74, 1997.

[BMM11] Roberto Bruni, Hernán C. Melgratti, and Ugo Montanari. Connector algebras, petri nets, and BIP. In Edmund M. Clarke, Irina B. Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics - 8th International Andrei Ershov Memorial Conference,*

PSI 2011, Novosibirsk, Russia, June 27-July 1, 2011, Revised Selected Papers, volume 7162 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 2011.

[BMMS16] Stefano Bistarelli, Fabio Martinelli, Ilaria Matteucci, and Francesco Santini. A formal and run-time framework for the adaptation of local behaviours to match a global property. In Olga Kouchnarenko and Ramtin Khosravi, editors, *Formal Aspects of Component Software - 13th International Conference, FACS 2016, Besançon, France, October 19-21, 2016, Revised Selected Papers*, volume 10231 of *Lecture Notes in Computer Science*, pages 134–152, 2016.

[BMR97] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.

[BMR06] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Soft concurrent constraint programming. *ACM Trans. Comput. Log.*, 7(3):563–589, 2006.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[BS07] Simon Bliudze and Joseph Sifakis. The algebra of connectors: structuring interaction in BIP. In Christoph M. Kirsch and Reinhard Wilhelm, editors, *Proceedings of the 7th ACM & IEEE International conference on Embedded software, EMSOFT 2007, September 30 - October 3, 2007, Salzburg, Austria*, pages 11–20. ACM, 2007.

[BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan J. M. M. Rutten. Modeling component connectors in reo by constraint automata. *Sci. Comput. Program.*, 61(2):75–113, 2006.

[BSBJ14] Simon Bliudze, Joseph Sifakis, Marius Bozga, and Mohamad Jaber. Architecture internalisation in BIP. In Lionel Seinturier, Eduardo Santana de Almeida, and Jan Carlson, editors, *CBSE'14, Proceedings of the 17th International ACM SIGSOFT Symposium on Component-Based Software Engineering (part of CompArch 2014), Marcq-en-Baroeul, Lille, France, June 30 - July 4, 2014*, pages 169–178. ACM, 2014.

[CCA07] Dave Clarke, David Costa, and Farhad Arbab. Connector colouring I: synchronisation and context dependency. *Sci. Comput. Program.*, 66(3):205–225, 2007.

[CDB+16] Philipp Chrszon, Clemens Dubslaff, Christel Baier, Joachim Klein, and Sascha Klüppelholz. Modeling role-based systems with exogenous coordination. In Ábrahám et al. [ÁBJ16], pages 122–139.

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[CGK+13] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. An overview of the mcrl2 toolset and its recent advances. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2013.

[CL00] Franck Cassez and Kim Guldstrand Larsen. The impressive power of stopwatches. In Catuscia Palamidessi, editor, *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, volume 1877 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2000.

[CNR11] David Costa, Milad Niqui, and Jan J. M. M. Rutten. Intentional automata: A context-dependent model for component connectors - (extended abstract). In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering - 4th IPM International Conference, FSEN 2011, Tehran, Iran, April 20-22, 2011, Revised Selected Papers*, volume 7141 of *Lecture Notes in Computer Science*, pages 335–342. Springer, 2011.

[CPLA11] Dave Clarke, José Proença, Alexander Lazovik, and Farhad Arbab. Channel-based coordination via constraint satisfaction. *Sci. Comput. Program.*, 76(8):681–710, 2011.

[CR17] Carlo Comin and Romeo Rizzi. Improved pseudo-polynomial bound for the value problem and optimal strategy synthesis in mean payoff games. *Algorithmica*, 77(4):995–1021, 2017.

[CRBS08] Mohamed Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis. Translating AADL into BIP - application to the verification of real-time systems. In Michel R. V. Chaudron, editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers*, volume 5421 of *Lecture Notes in Computer Science*, pages 5–19. Springer, 2008.

[DA17] Kasper Dokter and Farhad Arbab. Exposing latent mutual exclusion by work automata. In Mohammad Reza Mousavi and Jirí Sgall, editors, *Topics in Theoretical Computer Science - Second IFIP WG 1.8*

International Conference, TTCS 2017, Tehran, Iran, September 12-14, 2017, Proceedings, volume 10608 of *Lecture Notes in Computer Science*, pages 59–73. Springer, 2017.

[DA18a] Kasper Dokter and Farhad Arbab. Rule-based form for stream constraints. In Giovanna Di Marzo Serugendo and Michele Loreti, editors, *Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018. Proceedings*, volume 10852 of *Lecture Notes in Computer Science*, pages 142–161. Springer, 2018.

[DA18b] Kasper Dokter and Farhad Arbab. Treo: Textual syntax for reo connectors. In Simon Bliudze and Saddek Bensalem, editors, *Proceedings of the 1st International Workshop on Methods and Tools for Rigorous System Design, MeTRiD@ETAPS 2018, Thessaloniki, Greece, 15th April 2018*, volume 272 of *EPTCS*, pages 121–135, 2018.

[DA21] Kasper Dokter and Farhad Arbab. Protocol scheduling. In Hossein Hojjat and Mieke Massink, editors, *Fundamentals of Software Engineering - 9th International Conference, FSEN 2021, Virtual Event, May 19-21, 2021, Revised Selected Papers*, volume 12818 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2021.

[dBBR18] Frank S. de Boer, Marcello M. Bonsangue, and Jan Rutten, editors. *It's All About Coordination - Essays to Celebrate the Lifelong Scientific Achievements of Farhad Arbab*, volume 10865 of *Lecture Notes in Computer Science*. Springer, 2018.

[DGLS21] Kasper Dokter, Fabio Gadducci, Benjamin Lion, and Francesco Santini. Soft constraint automata with memory. *J. Log. Algebraic Methods Program.*, 118:100615, 2021.

[DGS18] Kasper Dokter, Fabio Gadducci, and Francesco Santini. Soft constraint automata with memory. In de Boer et al. [dBBR18], pages 70–85.

[DJA16] Kasper Dokter, Sung-Shik Jongmans, and Farhad Arbab. Scheduling games for concurrent systems. In Alberto Lluch-Lafuente and José Proença, editors, *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9686 of *Lecture Notes in Computer Science*, pages 84–100. Springer, 2016.

[DJAB15] Kasper Dokter, Sung-Shik Jongmans, Farhad Arbab, and Simon Bliudze. Relating BIP and reo. In Sophia Knight, Ivan Lanese, Alberto

Lluch-Lafuente, and Hugo Torres Vieira, editors, *Proceedings 8th Interaction and Concurrency Experience, ICE 2015, Grenoble, France, 4-5th June 2015*, volume 189 of *EPTCS*, pages 3–20, 2015.

[DJAB17] Kasper Dokter, Sung-Shik Jongmans, Farhad Arbab, and Simon Bliudze. Combine and conquer: Relating BIP and reo. *J. Log. Algebraic Methods Program.*, 86(1):134–156, 2017.

[DJL+15] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikucionis, and Jakob Haahr Taankvist. Uppaal stratego. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 206–211. Springer, 2015.

[DKV09] Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of weighted automata*. Springer Science & Business Media, 2009.

[Dok19] Kasper Dokter. Multilabeled petri nets. In Farhad Arbab and Sung-Shik Jongmans, editors, *Formal Aspects of Component Software - 16th International Conference, FACS 2019, Amsterdam, The Netherlands, October 23-25, 2019, Proceedings*, volume 12018 of *Lecture Notes in Computer Science*, pages 106–126. Springer, 2019.

[ECT] Extensible coordination tools ECT. `http://reo.project.cwi.nl`. Accessed: 2018-03-23.

[Ehl10] Rüdiger Ehlers. Minimising deterministic büchi automata precisely using SAT solving. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 326–332. Springer, 2010.

[EM79] A. Ehrenfeucht and J. Mycielski. Positional strategies for mean payoff games. *Int. J. Game Theory*, 8(2):109–113, 1979.

[Feh91] Rainer Fehling. A concept of hierarchical petri nets with building blocks. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1993, Papers from the 12th International Conference on Applications and Theory of Petri Nets, Gjern, Denmark, June 1991*, volume 674 of *Lecture Notes in Computer Science*, pages 148–168. Springer, 1991.

[FMS14] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[GHMW13] Fabio Gadducci, Matthias M. Hölzl, Giacoma Valentina Monreale, and Martin Wirsing. Soft constraints for lexicographic orders. In

Félix Castro-Espinoza, Alexander F. Gelbukh, and Miguel González-Mendoza, editors, *Advances in Artificial Intelligence and Its Applications - 12th Mexican International Conference on Artificial Intelligence, MICAI 2013, Mexico City, Mexico, November 24-30, 2013, Proceedings, Part I*, volume 8265 of *Lecture Notes in Computer Science*, pages 68–79. Springer, 2013.

[GJ92] Eric Goubault and Thomas P. Jensen. Homology of higher dimensional automata. In Rance Cleaveland, editor, *CONCUR '92, Third International Conference on Concurrency Theory, Stony Brook, NY, USA, August 24-27, 1992, Proceedings*, volume 630 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 1992.

[GM00] Fabio Gadducci and Ugo Montanari. The tile model. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 133–166. The MIT Press, 2000.

[Gol88] Ursula Goltz. On representing CCS programs by finite petri nets. In Michal Chytil, Ladislav Janiga, and Václav Koubek, editors, *Mathematical Foundations of Computer Science 1988, MFCS'88, Carlsbad, Czechoslovakia, August 29 - September 2, 1988, Proceedings*, volume 324 of *Lecture Notes in Computer Science*, pages 339–350. Springer, 1988.

[Gol03] J. Golan. *Semirings and Affine Equations over Them: Theory and Applications*. Kluwer, 2003.

[Gou95] Eric Goubault. Schedulers as abstract interpretations of higher-dimensional automata. In *Proc. of PEPM*, pages 134–145. ACM, 1995.

[Gra66] R.L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 1966.

[GS17] Fabio Gadducci and Francesco Santini. Residuation for bipolar preferences in soft constraints. *Inf. Process. Lett.*, 118:69–74, 2017.

[GSPV15] Fabio Gadducci, Francesco Santini, Luis Fernando Pino, and Frank D. Valencia. A labelled semantics for soft concurrent constraint programming. In Holvoet and Viroli [HV15], pages 133–149.

[GSPV17] Fabio Gadducci, Francesco Santini, Luis Fernando Pino, and Frank D. Valencia. Observational and behavioural equivalences for soft concurrent constraint programming. *Logical and Algebraic Methods in Programmming*, 92:45–63, 2017.

[Gun01] Jeremy Gunawardena. Homotopy and concurrency. In Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Current Trends in Theoretical Computer Science, Entering the 21th Century*, pages 447–459. World Scientific, 2001.

[Hen00] Thomas A Henzinger. The theory of hybrid automata. In M. Kemal Inan and Robert P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, pages 265–292. Springer, 2000.

[HMT81] Leon Henkin, J Donald Monk, and Alfred Tarski. Cylindric set algebras and related structures. In *Cylindric Set Algebras*, pages 1–129. Springer, 1981.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Hol04] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[HU67] John E. Hopcroft and Jeffrey D. Ullman. Nonerasing stack automata. *J. Comput. Syst. Sci.*, 1(2):166–186, 1967.

[HV15] Tom Holvoet and Mirko Viroli, editors. *Coordination Models and Languages - 17th IFIP WG 6.1 International Conference, COORDINATION 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, volume 9037 of *Lecture Notes in Computer Science*. Springer, 2015.

[IBC08] Mohammad Izadi, Marcello M. Bonsangue, and Dave Clarke. Modeling component connectors: Synchronisation and context-dependency. In Antonio Cerone and Stefan Gruner, editors, *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*, pages 303–312. IEEE Computer Society, 2008.

[JA12] Sung-Shik T. Q. Jongmans and Farhad Arbab. Overview of thirty semantic formalisms for reo. *Sci. Ann. Comp. Sci.*, 22(1):201–251, 2012.

[JA15] Sung-Shik T. Q. Jongmans and Farhad Arbab. Take command of your constraints! In Holvoet and Viroli [HV15], pages 117–132.

[JA16a] Sung-Shik T. Q. Jongmans and Farhad Arbab. Data optimizations for constraint automata. *Logical Methods in Computer Science*, 12(3), 2016.

[JA16b] Sung-Shik T. Q. Jongmans and Farhad Arbab. Prdk: Protocol programming with automata. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 547–552. Springer, 2016.

[JA18] Sung-Shik T. Q. Jongmans and Farhad Arbab. Centralized coordination vs. partially-distributed coordination with reo and constraint automata. *Sci. Comput. Program.*, 160:48–77, 2018.

[JCP16] Sung-Shik T. Q. Jongmans, Dave Clarke, and José Proença. A procedure for splitting data-aware processes and its application to coordination. *Sci. Comput. Program.*, 115-116:47–78, 2016.

[JHA14] Sung-Shik T. Q. Jongmans, Sean Halle, and Farhad Arbab. Automata-based optimization of interaction protocols for scalable multicore platforms. In eva Kühn and Rosario Pugliese, editors, *Coordination Models and Languages - 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, volume 8459 of *Lecture Notes in Computer Science*, pages 65–82. Springer, 2014.

[JKA11] Sung-Shik T. Q. Jongmans, Christian Krause, and Farhad Arbab. Encoding context-sensitivity in reo into non-context-sensitive semantic models. In Wolfgang De Meuter and Gruia-Catalin Roman, editors, *Coordination Models and Languages - 13th International Conference, COORDINATION 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, volume 6721 of *Lecture Notes in Computer Science*, pages 31–48. Springer, 2011.

[JKA17] Sung-Shik T. Q. Jongmans, Tobias Kappé, and Farhad Arbab. Constraint automata with memory cells and their composition. *Sci. Comput. Program.*, 146:50–86, 2017.

[Jon16] S.-S. T. Q. Jongmans. *Automata-theoretic protocol programming*. PhD thesis, Leiden University, 2016.

[Kan09] Krishna Kant. Data center evolution: A tutorial on state of the art, issues, and challenges. *Comput. Networks*, 53(17):2939–2965, 2009.

[KAT16] Tobias Kappé, Farhad Arbab, and Carolyn L. Talcott. A compositional framework for preference-aware agents. In Mehdi Kargahi and Ashutosh Trivedi, editors, *Proceedings of the The First Workshop on Verification and Validation of Cyber-Physical Systems, V2CPS@IFM 2016, Reykjavík, Iceland, June 4-5, 2016*, volume 232 of *EPTCS*, pages 21–35, 2016.

[KAT17] Tobias Kappé, Farhad Arbab, and Carolyn L. Talcott. A component-oriented framework for autonomous agents. In José Proença and Markus Lumpe, editors, *Formal Aspects of Component Software - 14th International Conference, FACS 2017, Braga, Portugal, October 10-13, 2017, Proceedings*, volume 10487 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2017.

[KC09] Christian Koehler and Dave Clarke. Decomposing port automata. In Sung Y. Shin and Sascha Ossowski, editors, *Proceedings of the 2009*

ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009, pages 1369–1373. ACM, 2009.

[Kem12] S. Kemper. Sat-based verification for timed component connectors. *Sci. Comput. Program.*, 77(7-8):779–798, 2012.

[KKdV12] Natallia Kokash, Christian Krause, and Erik P. de Vink. Reo + mcrl2: A framework for model-checking dataflow in service compositions. *Formal Aspects Comput.*, 24(2):187–216, 2012.

[Klü12] Sascha Klüppelholz. *Verification of Branching-Time and Alternating-Time Properties for Exogenous Coordination Models.* PhD thesis, Dresden University of Technology, 2012.

[KNSW07] Marta Z. Kwiatkowska, Gethin Norman, Jeremy Sproston, and Fuzhi Wang. Symbolic model checking for probabilistic timed automata. *Inf. Comput.*, 205(7):1027–1077, 2007.

[Kot78] Vadim E. Kotov. An algebra for parallelism based on petri nets. In Józef Winkowski, editor, *Mathematical Foundations of Computer Science 1978, Proceedings, 7th Symposium, Zakopane, Poland, September 4-8, 1978*, volume 64 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 1978.

[Kra09] Christian Krause. Integrated structure and semantics for reo connectors and petri nets. In Filippo Bonchi, Davide Grohmann, Paola Spoletini, and Emilio Tuosto, editors, *Proceedings 2nd Interaction and Concurrency Experience: Structured Interactions, ICE 2009, Bologna, Italy, 31st August 2009*, volume 12 of *EPTCS*, pages 57–69, 2009.

[Kru95] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[MA09] Sun Meng and Farhad Arbab. Qos-driven service selection and composition using quantitative constraint automata. *Fundam. Informaticae*, 95(1):103–128, 2009.

[Mar75] Donald A Martin. Borel determinacy. *Annals of Mathematics*, pages 363–371, 1975.

[Maz95] Antoni W. Mazurkiewicz. Introduction to trace theory. In Volker Diekert and Grzegorz Rozenberg, editors, *The Book of Traces*, pages 3–41. World Scientific, 1995.

[Mil89] Robin Milner. *Communication and Concurrency*, volume 84. Prentice-Hall, Inc., 1989.

[MM90] José Meseguer and Ugo Montanari. Petri nets are monoids. *Inf. Comput.*, 88(2):105–155, 1990.

[MSKA14] Young-Joo Moon, Alexandra Silva, Christian Krause, and Farhad Arbab. A compositional model to reason about end-to-end qos in stochastic reo connectors. *Sci. Comput. Program.*, 80:3–24, 2014.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[OMG06] OMG. CORBA component model, v4.0, OMG document formal/06-04-01, 2006. visited 11-13-2017.

[PA98] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. *Adv. Comput.*, 46:329–400, 1998.

[PA01] George A. Papadopoulos and Farhad Arbab. Configuration and dynamic reconfiguration of components using the coordination paradigm. *Future Gener. Comput. Syst.*, 17(8):1023–1038, 2001.

[Par13] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[PC08] José Proença and Dave Clarke. Coordination models orc and reo compared. *Electron. Notes Theor. Comput. Sci.*, 194(4):57–76, 2008.

[PCdVA12] José Proença, Dave Clarke, Erik P. de Vink, and Farhad Arbab. Dreams: a framework for distributed synchronous coordination. In Sascha Ossowski and Paola Lecca, editors, *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, pages 1510–1515. ACM, 2012.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 179–190. ACM Press, 1989.

[PR90] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*, pages 746–757. IEEE Computer Society, 1990.

[Pra86] Vaughan R. Pratt. Modeling concurrency with partial orders. *Int. J. Parallel Program.*, 15(1):33–71, 1986.

[Pra91] Vaughan R. Pratt. Modeling concurrency with geometry. In Wise [Wis91], pages 311–322.

[Rei13] Wolfgang Reisig. *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013.

[Reo] Reolanguage github repository. `https://github.com/ReoLanguage/Reo`. Accessed: 2018-03-23.

[reo16] Reo toolset, apr 2016.

[Roz92] Grzegorz Rozenberg, editor. *Advances in Petri Nets 1992, The DEMON Project*, volume 609 of *Lecture Notes in Computer Science*. Springer, 1992.

[Rut01] Jan J. M. M. Rutten. Elements of stream calculus (an extensive exercise in coinduction). *Electr. Notes Theor. Comput. Sci.*, 45:358–423, 2001.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[SRP91] Vijay A. Saraswat, Martin C. Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In Wise [Wis91], pages 333–352.

[SSAA13] Mahdi Sargolzaei, Francesco Santini, Farhad Arbab, and Hamideh Afsarmanesh. A tool for behaviour-based discovery of approximately matching web services. In Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti, editors, *Software Engineering and Formal Methods - 11th International Conference, SEFM 2013, Madrid, Spain, September 25-27, 2013. Proceedings*, volume 8137 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2013.

[Tal18] Carolyn L. Talcott. From soft agents to soft component automata and back. In de Boer et al. [dBBR18], pages 189–207.

[Tau89] Dirk Taubner. *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets*, volume 369 of *Lecture Notes in Computer Science*. Springer, 1989.

[TNAK16] Carolyn L. Talcott, Vivek Nigam, Farhad Arbab, and Tobias Kappé. Formal specification and analysis of robust adaptive distributed cyber-physical systems. In Marco Bernardo, Rocco De Nicola, and Jane Hillston, editors, *Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems - 16th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2016, Bertinoro, Italy, June 20-24, 2016, Advanced Lectures*, volume 9700 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2016.

[TSR11] Carolyn L. Talcott, Marjan Sirjani, and Shangping Ren. Comparing three coordination models: Reo, arc, and PBRD. *Sci. Comput. Program.*, 76(1):3–22, 2011.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[vG91] Rob J. van Glabbeek. Bisimulation semantics for higher dimensional automata. Email message, July 1991.

[vG06] Rob J. van Glabbeek. On the expressiveness of higher dimensional automata. *Theor. Comput. Sci.*, 356(3):265–290, 2006.

[vGV87] Rob J. van Glabbeek and Frits W. Vaandrager. Petri net models for algebraic theories of concurrency. In J. W. de Bakker, A. J. Nijman, and Philip C. Treleaven, editors, *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, June 15-19, 1987, Proceedings*, volume 259 of *Lecture Notes in Computer Science*, pages 224–242. Springer, 1987.

[vGV97] Rob J. van Glabbeek and Frits W. Vaandrager. The difference between splitting in n and n+1. *Inf. Comput.*, 136(2):109–142, 1997.

[VW02] Walter Vogler and Ralf Wollowski. Decomposition in asynchronous circuit design. In Jordi Cortadella, Alexandre Yakovlev, and Grzegorz Rozenberg, editors, *Concurrency and Hardware Design, Advances in Petri Nets*, volume 2549 of *Lecture Notes in Computer Science*, pages 152–190. Springer, 2002.

[Wim04] Harro Wimmel. Eliminating internal behaviour in petri nets. In Jordi Cortadella and Wolfgang Reisig, editors, *Applications and Theory of Petri Nets 2004, 25th International Conference, ICATPN 2004, Bologna, Italy, June 21-25, 2004, Proceedings*, volume 3099 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2004.

[Win87] Glynn Winskel. Petri nets, algebras, morphisms, and compositionality. *Inf. Comput.*, 72(3):197–238, 1987.

[Wis91] David S. Wise, editor. *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991.* ACM Press, 1991.

[WN95] Glynn Winskel and Mogens Nielsen. Models for concurrency. In *Handbook of Logic in Computer Science*, volume 4, pages 1–148. Oxford University Press, 1995.

Appendix A

Original CSDF program

```
import multiprocessing as mp
import queue
import psutil
import time

def dummy_work(duration):
    a = 1
    for i in range(duration * 100000):
        a = (a * i) % 10000

def put_or_lose(e):
    try:
        e.put_nowait(True)
    except queue.Full:
        pass

def A1(e1, e2, e3):
    wcet = 5
    psutil.Process().cpu_affinity([0,1])
    while True:
        dummy_work(wcet)
        put_or_lose(e1)
        put_or_lose(e3)
        dummy_work(wcet)
        put_or_lose(e1)
        put_or_lose(e3)
        dummy_work(wcet)
        put_or_lose(e2)
        put_or_lose(e3)

def A2(e1, e4):
    wcet = 8
    psutil.Process().cpu_affinity([0,1])
    while True:
        e1.get()
```

```
        dummy_work(wcet)
        put_or_lose(e4)

def A3(e2, e5):
    wcet = 24
    psutil.Process().cpu_affinity([0,1])
    while True:
        e2.get()
        dummy_work(wcet)
        put_or_lose(e5)

def A4(e3, e4, e5):
    wcet = 4
    psutil.Process().cpu_affinity([0,1])
    throughputs = []
    while len(throughputs) < 500:
        t = time.time()
        dummy_work(wcet)
        e3.get()
        e4.get()
        dummy_work(wcet)
        e3.get()
        e4.get()
        dummy_work(wcet)
        e3.get()
        e5.get()
        throughputs.append(round(1000*(time.time()-t)))
    print(throughputs)

if __name__ == '__main__':
    e1 = mp.Queue(50)
    e2 = mp.Queue(50)
    e3 = mp.Queue(50)
    e4 = mp.Queue(50)
    e5 = mp.Queue(50)
    A = [
        mp.Process(target=A1,args=(e1, e2, e3), daemon=True),
        mp.Process(target=A2,args=(e1, e4), daemon=True),
        mp.Process(target=A3,args=(e2, e5), daemon=True),
        mp.Process(target=A4,args=(e3, e4, e5), daemon=True)
    ]
    for p in A:
        p.start()
    A[3].join()
```

Appendix B

Scheduled CSDF program

```
import multiprocessing as mp
import queue
import psutil
import time

def dummy_work(duration):
    a = 1
    for i in range(duration * 100000):
        a = (a * i) % 10000

def put_or_lose(e):
    try:
        e.put_nowait(True)
    except queue.Full:
        pass

def A1(e1, e2, e3, b):
    wcet = 5
    psutil.Process().cpu_affinity([0,1])
    while True:
        b.wait()
        dummy_work(wcet)
        put_or_lose(e1)
        put_or_lose(e3)
        b.wait()
        b.wait()
        dummy_work(wcet)
        put_or_lose(e1)
        put_or_lose(e3)
        b.wait()
        b.wait()
        dummy_work(wcet)
        put_or_lose(e2)
        put_or_lose(e3)
        b.wait()
```

```
def A2(e1, e4, b):
    wcet = 8
    psutil.Process().cpu_affinity([0,1])
    while True:
        b.wait()
        e1.get()
        dummy_work(wcet)
        put_or_lose(e4)
        b.wait()

def A3(e2, e5, b):
    wcet = 24
    psutil.Process().cpu_affinity([0,1])
    while True:
        b.wait()
        e2.get()
        dummy_work(wcet)
        put_or_lose(e5)
        b.wait()

def A4(e3, e4, e5, b):
    wcet = 4
    psutil.Process().cpu_affinity([0,1])
    throughputs = []
    while len(throughputs) < 500:
        t = time.time()
        b.wait()
        dummy_work(wcet)
        e3.get()
        e4.get()
        b.wait()
        b.wait()
        dummy_work(wcet)
        e3.get()
        e4.get()
        b.wait()
        b.wait()
        dummy_work(wcet)
        e3.get()
        e5.get()
        b.wait()
        throughputs.append(round(1000*(time.time()-t)))
    print(throughputs)

def Scheduler(b1, b2, b3, b4):
    b1.wait()
    b1.wait()
    b1.wait()
    b2.wait()
    b1.wait()
    b1.wait()
```

```
    b2.wait()
    b4.wait()
    b1.wait()
    b3.wait()
    b4.wait()
    while True:
        b2.wait()
        b2.wait()
        b4.wait()
        b4.wait()
        b1.wait()
        b1.wait()
        b3.wait()
        b1.wait()
        b4.wait()
        b4.wait()
        b2.wait()
        b1.wait()
        b1.wait()
        b1.wait()
        b3.wait()
        b2.wait()
        b4.wait()
        b4.wait()

if __name__ == '__main__':
    e1 = mp.Queue(50)
    e2 = mp.Queue(50)
    e3 = mp.Queue(50)
    e4 = mp.Queue(50)
    e5 = mp.Queue(50)
    b1 = mp.Barrier(2)
    b2 = mp.Barrier(2)
    b3 = mp.Barrier(2)
    b4 = mp.Barrier(2)
    A = [
        mp.Process(target=A1,args=(e1,e2,e3,b1),daemon=True),
        mp.Process(target=A2,args=(e1,e4,b2),daemon=True),
        mp.Process(target=A3,args=(e2,e5,b3),daemon=True),
        mp.Process(target=A4,args=(e3,e4,e5,b4),daemon=True),
        mp.Process(
          target=Scheduler,
          args=(b1,b2,b3,b4),
          daemon=True
        )
    ]
    for p in A:
        p.start()
    A[3].join()
```

Curriculum Vitae

Born in 1990 in Putten.

Education

- 1994-2002: Basisschool met de Bijbel Huinen, Putten
- 2002-2007: Havo, Johannes Fontanus College, Barneveld
- 2007-2008: Propedeuse mechanical engineering, Windesheim, Zwolle
- 2008-2011: BSc Wiskunde (cum laude), Utrecht University
- 2011-2013: MSc Mathematical sciences (cum laude), Utrecht University

Employment

- 2014-2017: Doctoral candidate, Centrum Wiskunde & Informatica
- 2017-2019: Doctoral candidate, Leiden University
- 2018-2022: Financial data analyst, SoliTrust, Apeldoorn
- 2019-2021: Member of ECiDA project, Centrum Wiskunde & Informatica
- 2022-Now: Data scientist and software engineer, self-employed

List of Publications

The results of this dissertation are based on the following publications:

Journal Articles

- Kasper Dokter, Fabio Gadducci, Benjamin Lion, Francesco Santini: Soft constraint automata with memory. *J. Log. Algebraic Methods Program.* 118: 100615 (2021)
- Kasper Dokter, Sung-Shik Jongmans, Farhad Arbab, Simon Bliudze: Combine and conquer: Relating BIP and Reo. *J. Log. Algebraic Methods Program.* 86(1): 134-156 (2017)

Conference Papers

- Kasper Dokter, Farhad Arbab: Protocol Scheduling. *FSEN 2021*: 3-17
- Kasper Dokter: Multilabeled Petri Nets. *FACS 2019*: 106-126
- Kasper Dokter, Fabio Gadducci, Francesco Santini: Soft Constraint Automata with Memory. *It's All About Coordination 2018*: 70-85
- Kasper Dokter, Farhad Arbab: Rule-Based Form for Stream Constraints. COORDINATION 2018: 142-161
- Kasper Dokter, Farhad Arbab: Treo: Textual Syntax for Reo Connectors. *MeTRiD@ETAPS 2018*: 121-135
- Kasper Dokter, Farhad Arbab: Exposing Latent Mutual Exclusion by Work Automata. *TTCS 2017*: 59-73
- Kasper Dokter, Sung-Shik Jongmans, Farhad Arbab: Scheduling Games for Concurrent Systems. *COORDINATION 2016*: 84-100
- Kasper Dokter, Sung-Shik Jongmans, Farhad Arbab, Simon Bliudze: Relating BIP and Reo. *ICE 2015*: 3-20

Samenvatting

Met de opkomst van multicore processoren en datacenters is computer hardware in toenemende mate parallel geworden, waardoor het mogelijk is om verschillende softwareonderdelen tegelijkertijd op verschillende machines uit te voeren. Coördinatie van deze softwareonderdelen wordt het beste uitgedrukt in een coördinatietaal als een expliciet interactieprotocol dat de interacties tussen alle onderdelen van de software duidelijk definieert.

Een expliciet interactieprotocol verbetert niet alleen de structuur van de code, maar maakt ook geautomatiseerde analyse van het protocol mogelijk om de uitvoeringsefficiëntie van de software te verbeteren. In het bijzonder bevatten interactieprotocollen belangrijke informatie die essentieel is voor efficiënte roostering, een activiteit die betrekking heeft op de toewijzing van (reken)middelen aan softwaretaken. In dit proefschrift richten we ons specifiek op het verbeteren van de uitvoeringsefficiëntie door middel van roosteren. Roosteren is bijna altijd de verantwoordelijkheid van een generiek besturingssysteem dat geen aannames maakt over de software en daardoor alle relevante roosterinformatie in die software negeert. Als gevolg hiervan kan het besturingssysteem alleen niet zorgen voor optimaal geroosterde uitvoering van de software.

In dit proefschrift stellen we een oplossing voor die het protocol in de software verandert, zodat het efficiënt wordt geroosterd door het generieke besturingssysteem. Het belangrijkste idee is om gebruik te maken van de dualiteit tussen roostering en coördinatie. Om precies te zijn, analyseren we het protocol van de software om een optimale roosterstrategie voor deze software te bepalen. Vervolgens dwingen we dit optimale rooster af door de strategie op te nemen in het oorspronkelijke protocol. Als gevolg hiervan dwingen we de onwetende roostermodule van het besturingssysteem om ons vooraf bepaalde optimale rooster te volgen.

Om dit grotere doel te bereiken, presenteren we drie kleinere bijdragen. Ten eerste verkrijgen we een uitgangspunt voor de planningsinformatie die beschikbaar is in een coördinatietaal door twee coördinatiestalen, BIP en Reo, met elkaar te vergelijken. Onze vergelijking leidt tot het voorstel van een compositieoperator voor datasensitieve BIP-architecturen en de uitbreiding van de theorie van zachte beperkingsautomaten (soft constraint automata) met geheugencellen en bipolaire voorkeurswaarden.

Vervolgens bepalen we alle onafhankelijke delen van de software (inclusief die in het protocol) die kunnen worden gepland. Hier introduceren we twee werkelijk gelijktijdige semantieken, namelijk multigelabelde Petri-netten en stroombeperkingen (stream constraints) in regelgebaseerde vorm. Door deze semantieken als interne representatie te gebruiken, verbeteren we aanzienlijk de state-of-the-art

Reo-compiler. Als bijproduct stellen we een tekstuele taal voor genaamd Treo voor, waarmee we grote protocollen kunnen opbouwen door primitieve protocollen samen te stellen.

Ten slotte representeren we alle relevante roosterinformatie van elk deel van de software door de software uit te drukken als een werkautomaat die uitdrukt hoeveel werk elk deel van de software kan/moet doen. We gebruiken deze werkautomaten om een speltheoretisch roosterraamwerk te ontwikkelen dat roosteren formaliseert als een tweespeler-nulspel dat wordt gespeeld op een graaf. We maken een kleine aanpassing op bestaande oplossingen om een roosterstrategie te berekenen voor een kleine cyclo-statistische datastroomtoepassing die de doorvoer optimaliseert. Zoals beloofd dwingen we de optimale roosterstrategie af door deze te integreren met het oorspronkelijke protocol, waardoor aangepaste wijzigingen in de standaard roostermodule van het besturingssysteem worden vermeden.

Hoofdstuk 9 geeft een uitgebreidere samenvatting van dit proefschrift.

Summary

With the advent of multicore processors and data centers, computer hardware has become increasingly parallel, allowing one to run multiple pieces of software at the same time on different machines. Coordination of these pieces is best expressed in a coordination language as an explicit interaction protocol that clearly defines the interactions among all components in the software.

An explicit interaction protocol not only improves code structure but also enables automated analysis of the protocol to improve execution efficiency of the software. Specifically, interaction protocols contain significant information that is essential for efficient scheduling, an activity that concerns the allocation of (computing) resources to software tasks. In this thesis, we focus in particular on improving execution efficiency through scheduling. Almost always, scheduling is the responsibility of a general-purpose operating system that makes no assumptions on the software and thereby ignores all relevant scheduling information in that software. As a result, the operating system alone cannot ensure optimally scheduled execution of the software.

In this thesis, we propose a solution that changes the protocol in the software such that it will be efficiently scheduled by the general-purpose operating system. The main idea is to take advantage of the duality between scheduling and coordination. To be precise, we analyze the protocol of the software to determine an optimal scheduling strategy for this software. Then, we enforce this optimal schedule by incorporating the strategy in the original protocol. As a result, we force the ignorant operating scheduler to follow our precomputed optimal schedule.

To achieve this larger goal, we present three smaller contributions. First, we obtain a baseline for the scheduling information that is available in a coordination language by comparing two coordination languages, BIP and Reo. Our comparison leads to the proposal of a composition operator for data-sensitive BIP architectures and the expansion of the theory of soft constraint automata with memory cells and bipolar preference values.

Next, we concretely establish all independent parts in the software (including those in the protocol) that can be scheduled. Here, we introduce two truly concurrent semantics namely multilabeled Petri nets and stream constraints in rule-based form. Using these semantics as an intermediate representation, we significantly improve state-of-the-art Reo compiler. As a byproduct, we propose a textual language called Treo that allows us to construct large protocols by composing primitive ones.

Finally, we represent all relevant scheduling information of each part of the software by expressing the software as a work automaton that expresses how much work each part of the software can/must do. We use these work automata to

develop a game-theoretic scheduling framework that formalizes scheduling as a two-player zero-sum game played on a graph. We slightly adapt existing solutions to compute a scheduling strategy for a small cyclo-static dataflow application that optimizes throughput. As promised, we enforce the optimal scheduling strategy by integrating it with the original protocol, which avoids custom changes to the default operating system scheduler.

A more extensive summary of this thesis appears in Chapter 9.

Acknowledgments

First and foremost, I would like to express my deep gratitude to my parents, Aart and Nanny Dokter, for their unwavering support throughout my academic journey. Their love, encouragement, and belief in me have been my constant source of strength and inspiration.

I would also like to thank my sister Hermien Bokhorst for her outstanding artistic work on the cover of this thesis. Her creativity and talent have added a special touch to this project and made it truly unique.

My sincere appreciation goes to Farhad Arbab for the way he fulfilled his duties as a guide and mentor. His commitment to excellence has been instrumental in shaping my research and professional development.

I am deeply grateful to Sung-Shik Jongmans and Benjamin Lion for their valuable contributions to my research. Their insights, feedback, and expertise have greatly enriched my work and broadened my perspectives.

I would also like to acknowledge Jan Rutten for his supervision during Farhad's short absence. His support and guidance were crucial in ensuring the continuity and quality of my research.

Finally, I would like to express my heartfelt thanks to Kim Larssen for inviting me to Denmark and offering me the opportunity to discuss my work. His generosity, kindness, and enthusiasm have been a source of encouragement and motivation.

Last but not least, I would like to extend my appreciation to Frank de Boer, Krzysztof Apt, Keyvan Azadbakht, Vlad Serbanescu, Julian Salamanca, Hans-Dieter Hiep, Klaas van Harn, and all the other colleagues, friends, and family members who have supported me along the way. Your encouragement, feedback, and friendship have been invaluable to me, and I am deeply grateful for your presence in my life.

Titles in the IPA Dissertation Series since 2020

M.A. Cano Grijalba. *Session-Based Concurrency: Between Operational and Declarative Views.* Faculty of Science and Engineering, RUG. 2020-01

T.C. Nägele. *CoHLA: Rapid Co-simulation Construction.* Faculty of Science, Mathematics and Computer Science, RU. 2020-02

R.A. van Rozen. *Languages of Games and Play: Automating Game Design & Enabling Live Programming.* Faculty of Science, UvA. 2020-03

B. Changizi. *Constraint-Based Analysis of Business Process Models.* Faculty of Mathematics and Natural Sciences, UL. 2020-04

N. Naus. *Assisting End Users in Workflow Systems.* Faculty of Science, UU. 2020-05

J.J.H.M. Wulms. *Stability of Geometric Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2020-06

T.S. Neele. *Reductions for Parity Games and Model Checking.* Faculty of Mathematics and Computer Science, TU/e. 2020-07

P. van den Bos. *Coverage and Games in Model-Based Testing.* Faculty of Science, RU. 2020-08

M.F.M. Sondag. *Algorithms for Coherent Rectangular Visualizations.* Faculty of Mathematics and Computer Science, TU/e. 2020-09

D. Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security.* Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic.* Faculty of Sciences, Department of Computer Science, VU. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components.* Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management.* Faculty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS.* Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning.* Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs.* Faculty of Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical.* Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions.* Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition.* Faculty of

Sciences, Department of Computer Science, VU. 2022-07

J. Wagemaker. *Concurrent Separation Logics for Safety, Refinement, and Security.* Faculty of Science, Mathematics and Computer Science, RU. 2022-08

R. Janssen. *Refinement and Partiality for Model-Based Testing.* Faculty of Science, Mathematics and Computer Science, RU. 2022-09

M. Laveaux. *Accelerated Verification of Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2022-10

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving.* Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution.* Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems.* Faculty of Mathematics and Computer Science, TU/e. 2023-03

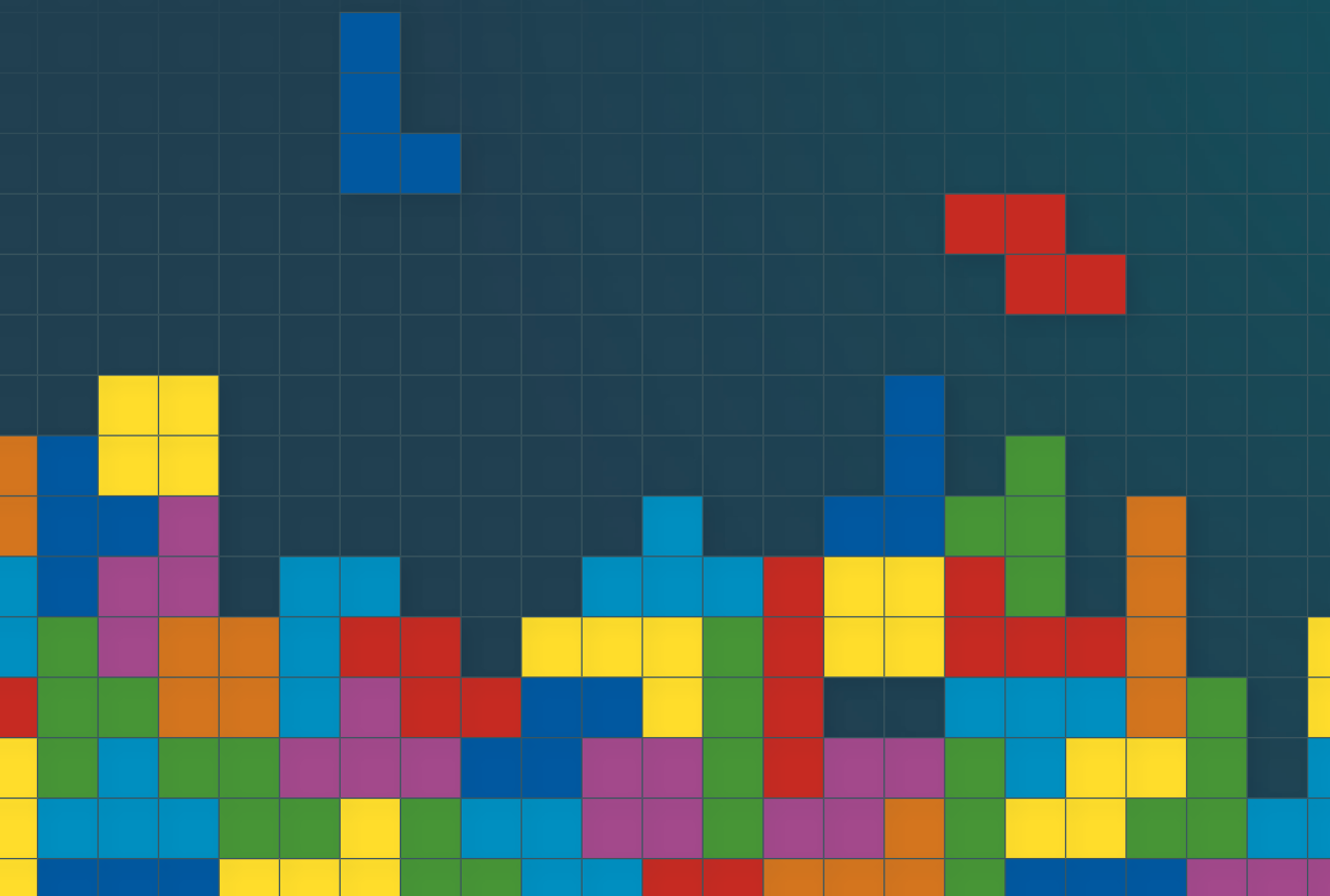
J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN.* Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming.* Faculty of Mathematics and Natural Sciences, UL. 2023-05

CWI

Research institute for mathematics & computer science in the Netherlands

Universiteit Leiden

The Netherlands

ipa