Cover Page

Universiteit Leiden

The handle http://hdl.handle.net/1887/38223 holds various files of this Leiden University dissertation

Author: Jongmans, Sung-Shik T.Q.
Title: Automata-theoretic protocol programming : parallel computation, threads and their interaction, optimized compilation, [at a] high level of abstraction
Issue Date: 2016-03-03

Chapter 2

DSL for Interaction I: Semantics

As defined in Chapter 1, a true intention-expressing DSL for interaction provides constructs for implementing protocol specifications as first-class entities that preserve as much intention information from software engineers as possible. To achieve this, such constructs should very precisely capture the intention that people have when they use the word "protocol". In this thesis, I therefore commit myself to the Oxford Advanced American Dictionary, which informally defines "protocol" as follows:

> "**3** [countable] (*computing*) a set of rules that control the way data is sent between computers"—[Lea11, page 1179]

In this chapter, I present the semantics of a true intention-expressing DSL for interaction, whose programs satisfy this dictionary definition, replacing "computers" with "workers" according to terminology established in Chapter 1. In this DSL, significantly, *protocols* (i.e., admissible interaction, i.e., constraints on interaction as explained in Chapter 1) constitute the set of first-class entities (i.e., mathematical relations, concisely represented through automata, as explained in this chapter) and primary units of composition, built out of atomic protocols—interactions—which explicitly constrain the timing, ordering, and data-flows between actions. This strongly contrasts, for instance, process calculi, where *processes* constitute the set of first-class entities and primary units of composition, built of out atomic processes—actions—which only implicitly may (or may not!) induce interaction.

In Section 2.1, I present both elementary machinery for modeling "[...] the way data is sent between workers" and complementary machinery for modeling "a set of rules that control [...]". In Section 2.2, I briefly discuss a practical incarnation of the theoretical work in Section 2.1.

2.1 Theory

(I have not yet submitted the material in this section for publication.)

Interaction Languages

In this thesis, as explained in Chapter 1, interaction among workers occurs through blocking I/O operations with value-passing semantics on their ports. Workers may have access to multiple ports, each of which they may use for a different purpose or even in different protocols. Consequently, modeling protocols at the finer level of "interaction among ports" instead of at the coarser level of "interaction among workers" improves the accuracy of the resulting models. Indeed, in contrast to the latter approach, the former approach enables me to state precisely through which of a worker's ports certain interaction occurs. The compiler that I present in Chapter 4 actually requires this level of precision. Therefore, in what follows, I develop machinery for modeling "the way data is exchanged through ports" instead of "sent between workers".

As a first step, I formally define two essential ingredients: ports and data.

Definition 1 (ports). *A port is an unstructured object.* $\mathbb{P}$ *denotes the set of all ports, ranged over by* p. $2^{\mathbb{P}}$ *denotes the set of all sets of ports, ranged over by* P, V. $2^{2^{\mathbb{P}}}$ *denotes the set of all sets of sets of ports, ranged over by* E. $2^{2^{2^{\mathbb{P}}}}$ *denotes the set of all sets of sets of sets of ports, ranged over by* G.

Definition 2 (data). *A datum is an unstructured object.* $\mathbb{D}$ *denotes the set of all data, ranged over by* d*, such that* $\mathbb{P} \cap \mathbb{D} = \emptyset$.

Definition 3 (empty datum). `nil` *is an unstructured object such that* `nil` $\notin \mathbb{D}$.

The extra condition in Definition 2 means that workers do not communicate their ports to other workers; the theory presented in this chapter does not support *mobility* as in π-calculus. In fact, in practice, the value-passing semantics of I/O operations inhibits this. Nevertheless, as explained in Chapter 1, nothing prevents a worker from sending a reference to a port *as a value* to another worker; and symmetrically, nothing prevents this other worker from interpreting and using this value as a reference to a port. Whenever software engineers set this up, however, they also take full responsibility for the consequences. Definition 3 asserts the existence of a distinguished empty datum. The exact content of $\mathbb{P}$ and $\mathbb{D}$ depends on the context of their use and formally does not matter much. For instance, if I use Java for writing worker subprograms, $\mathbb{D}$ contains all Java objects. Henceforth, I write elements of $\mathbb{P}$ in capitalized lower case sans-serif (e.g., A, B, C, In_1, Out_2), while I write elements of $\mathbb{D}$ in lower case monospace (e.g., `1`, `3.14`, `true`, `"foo"`). At this point, I do not yet distinguish input ports from output ports; this comes later, in the next subsection.

Out of ports and data, I construct *interaction letters*, so called for a reason that becomes clear shortly. Every interaction letter models one instance of interaction, in which particular data pass through particular ports. Formally, I

define an interaction letter λ as a partial function that associates a datum d with every port p in its domain, where $\lambda(p) = d$ means that d passes through p in the instance of interaction modeled by λ.

> **Definition 4** (interaction letters)**.** *An interaction letter is a partial function from ports to data.* $\mathbb{L}\text{ETT} = (\mathbb{P} \rightharpoonup \mathbb{D}) \setminus \emptyset$ *denotes the set of all interaction letters, ranged over by* λ*.* $2^{\mathbb{L}\text{ETT}}$ *denotes the set of all sets of interaction letters, ranged over by* Λ*.*

Suppose that one producer has access to port A, the other producer to port B, and the consumer to port C in the producers/consumer example in Chapter 1. In that case, interaction letter $\{\mathsf{A} \mapsto \texttt{"foo"}\}$ models an instance of interaction in which one producer exchanges datum `"foo"`—a string—through port A with its environment. Similarly, interaction letter $\{\mathsf{C} \mapsto \texttt{"foo"}\}$ models an instance of interaction in which the consumer exchanges `"foo"` through port C with its environment. Together, these two instances of interaction can model the asynchronous send/receive of `"foo"` from a producer (on port A) to the consumer (on port C). Similarly, interaction letter $\{\mathsf{A} \mapsto \texttt{"foo"}, \mathsf{C} \mapsto \texttt{"foo"}\}$ can model an instance of interaction in which a producer and the consumer synchronously exchange `"foo"`. The LateAsyncMerger2 protocol in Chapter 1 forbids such synchronous communication, though, so this third instance of interaction should never occur.

Out of interaction letters, I construct *interaction words*, so called for a reason that becomes clear shortly. Every interaction word models one chain of interaction, in which infinitely many instances of interaction follow each other. Formally, I define an interaction word w as an infinite sequence of nonempty interaction letters $\lambda_1\lambda_2\cdots$, where λ_i models the i-th instance of interaction in the chain of interaction modeled by w.

> **Definition 5** (interaction words)**.** *An interaction word is an infinite sequence of interaction letters.* $\mathbb{W}\text{ORD} = \mathbb{L}\text{ETT}^{\omega}$ *denotes the set of all interaction words, ranged over by* w*.*

Continuing the previous example, the following four interaction words model chains of interaction in which, from left to right, (i) only one producer communicates with the consumer, (ii) both producers communicate with the consumer (iii) both producers synchronously communicate with the consumer, and (iv) both producers nontransactionally communicate with the consumer:

(i)	(ii)	(iii)	(iv)
$\{\mathsf{A} \mapsto \texttt{"foo"}\}$	$\{\mathsf{A} \mapsto \texttt{"foo"}\}$	$\{\mathsf{A} \mapsto \texttt{"foo"}, \mathsf{C} \mapsto \texttt{"foo"}\}$	$\{\mathsf{A} \mapsto \texttt{"foo"}\}$
$\{\mathsf{C} \mapsto \texttt{"foo"}\}$	$\{\mathsf{C} \mapsto \texttt{"foo"}\}$	$\{\mathsf{B} \mapsto \texttt{"bar"}, \mathsf{C} \mapsto \texttt{"bar"}\}$	$\{\mathsf{B} \mapsto \texttt{"bar"}\}$
$\{\mathsf{A} \mapsto \texttt{"bar"}\}$	$\{\mathsf{B} \mapsto \texttt{"bar"}\}$	$\{\mathsf{A} \mapsto \texttt{"baz"}, \mathsf{C} \mapsto \texttt{"baz"}\}$	$\{\mathsf{C} \mapsto \texttt{"foo"}\}$
$\{\mathsf{C} \mapsto \texttt{"bar"}\}$	$\{\mathsf{C} \mapsto \texttt{"bar"}\}$	$\{\mathsf{B} \mapsto \texttt{"qux"}, \mathsf{C} \mapsto \texttt{"qux"}\}$	$\{\mathsf{C} \mapsto \texttt{"bar"}\}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

LateAsyncMerger2 forbids the chains of interaction modeled by (iii) and (iv).

Out of interaction words, I construct *interaction languages*, so called for a reason that becomes clear shortly. Every interaction language models a collection of chains of interaction. Formally, I define an interaction language $\mathcal{L}$ as a set of interaction words.

Definition 6 (interaction languages)**.** *An interaction language is a set of interaction words.* $\mathbb{L}\text{ANG} = 2^{\mathbb{W}\text{ORD}}$ *denotes the set of all interaction languages, ranged over by* $\mathcal{L}$*.*

Continuing the previous example, the following interaction language models all chains of interaction admitted by LateAsyncMerger2.

$$\left\{ w \,\middle|\, \begin{array}{l} w \in \mathbb{W}\text{ORD} \\ \textbf{and } \left[\left[i \bmod 2 = 0 \textbf{ implies } \mathrm{Dom}(w(i)) \subset \{\mathsf{A}, \mathsf{B}\}\right] \textbf{ for all } i\right] \\ \textbf{and } \left[\left[i \bmod 2 = 1 \textbf{ implies } \mathrm{Dom}(w(i)) = \{\mathsf{C}\}\right] \textbf{ for all } i\right] \end{array} \right\}$$

The machinery presented so far models the second component of the dictionary definition on page 27 as follows. First, an interaction word (i.e., a chain of interaction) models one infinite "way data is exchanged through ports" in one particular infinite run of a program. Then, an interaction language (i.e., a collection of chains of interaction) models a collection of infinite ways "data is exchanged through ports". One can model finite ways "data is exchanged through ports" by extending finite sequences to infinite sequences as usual.

Interaction letters, interaction words, and interaction languages go by different names in the literature. For instance, Izadi et al. call interaction letters *records*, interaction words *streams of records*, and interaction languages *languages of records* [IBC11, Iza11]. Alternatively, both Baier et al., Klein, and Klüppelholz et al. call interaction letters *concurrent* I/O *operations* and interaction words I/O *streams* [BBK+10, BBKK09a, BBKK09b, BKK11, KB09, KB10, Kle12, Klü12], while Arbab et al. call interaction words *scheduled data streams* [ABdBR07]. Each of those names refers to the same kind of mathematical object, though. Tuples of *timed data streams* (TDS) comprise a different but still related kind of mathematical object, originally introduced by Rutten and Arbab and later further developed by Arbab into *abstract behavior types* [AR03, Arb05]. Every tuple of TDSs contains one TDS for every port of interest. Every TDS, in turn, consists of two infinite sequences: a *time stream* of monotonically increasing real numbers and a *data stream* of data. A TDS for a port p subsequently models that the i-th datum in the data stream flows through p at the time represented by the i-th real number in the time stream. Consequently, tuples of TDSs contain not only information about the order in which instances of interaction take place but also more precise timing information. For instance, if I need to extract the full instance of interaction that occurs at time 3.14, I check for every port (i) which index 3.14 has in that port's time stream, and if such an index indeed exists, (ii) which datum occurs in that port's data stream at that index.

Constraint Automata

If an interaction language contains exactly those interaction words that model the chains of interaction admitted by some protocol, this interaction language indeed models that protocol. To model protocols in terms of interaction languages, thus, I need a method of concisely specifying the content of interaction languages. This brings me to the first component of the dictionary definition on page 27: I need machinery for modeling "a set of rules that control [the second component]". I intend to capture such "a set of rules" that a protocol consists of with an *automaton* of some kind, and in particular, with that automaton's *transition relation*. By constructing this automaton such that it *accepts* interaction words, each of its transitions effectively models one of the *state*ful "rules that control the way data is exchanged through ports".

Naively, I may adopt the set of all interaction letters as my alphabet and use Büchi automata as interaction language acceptors. In this approach, every infinite sequence of transitions straightforwardly corresponds to exactly one interaction word. However, such an *explicit representation* of interaction letters on transitions has a problem: if the set of all data $\mathbb{D}$ contains infinitely many elements, so does the set of all interaction letters $\mathbb{L}\text{ETT}$. In that case, the resulting Büchi automata have infinite transition relations, which I cannot account for.

Instead of labeling a transition $\boldsymbol{t}$ with an interaction letter, I label $\boldsymbol{t}$ with a *symbolic representation* of a possibly infinite set of interaction letters Λ. Every such a representation consists of two elements: a *synchronization constraint* and a *data constraint*. A synchronization constraint specifies the domain of every interaction letter in Λ. This models which ports participate in every instance of interaction modeled by Λ. Formally, I define a synchronization constraint as a set of ports. A data constraint specifies two things. First, it specifies to which data every interaction letter in Λ maps the ports in its domain. This models which data pass through which ports in every instance of interaction modeled by Λ. Second, it specifies the content of *memory cells* before and after firing $\boldsymbol{t}$. This models how internal buffers in a protocol evolve. Shortly, I define a data constraint as a formula in a first-order calculus with variables, constants, functions, and relations [Rau10a]. Before I can do so, however, I first need to introduce other machinery.

I start by formally defining memory cells.

Definition 7 (memory cells)**.** *A memory cell is an unstructured object.* $\mathbb{M}$ *denotes the set of all memory cells, ranged over by* m. $2^{\mathbb{M}}$ *denotes the set of all sets of memory cells, ranged over by* M.

The exact content of $\mathbb{M}$ depends on the context of its use and does not matter much. Henceforth, I write elements of $\mathbb{M}$ in lower case sans-serif (e.g., x, buff_1).

Out of memory cells and data, I construct *memory snapshots*. Every memory snapshot models the content of memory cells in some time instant. Formally, I define a memory snapshot μ as a partial function that associates a datum d to every memory cell m in its domain, where $\mu(m) = d$ means that m contains d in the time instant modeled by μ.

Definition 8 (memory snapshots)**.** *A memory snapshot is a partial function from memory cells to data.* $\mathbb{S}\text{NAPSH} = \mathbb{M} \rightharpoonup \mathbb{D}$ *denotes the set of all memory snapshots, ranged over by* μ*.*

Suppose that the producers and the consumer use buffer x for temporary storage of their data in the producers/consumer example in Chapter 1. In that case, $\{\texttt{x} \mapsto \texttt{0}\}$ models the initial content of the buffer (i.e., an arbitrarily selected datum), while $\{\texttt{x} \mapsto \texttt{"foo"}\}$ models the content of the buffer after a producer has sent `"foo"` to the consumer.

I proceed by defining variables in the calculus, called *data variables*. Every data variable models a container for data. For instance, ports can hold data, so every port serves as a data variable in the calculus. Similarly, memory cells can hold data, but the meaning of "to hold" differs in this case. Ports hold data only *during* an instance of interaction (i.e., transiently, in passing). In contrast, memory cells hold data also *before* and *after* an instance of interaction. Consequently, in the context of data variables, a memory cell before an instance of interaction and the same memory cell after that instance have a different identity. After all, the content of the memory cell may have changed in between. Therefore—inspired by notation from Petri nets [Rei85]—for every memory cell m, both $^\bullet m$ and $m^\bullet$ serve as data variables: $^\bullet m$ refers to the datum in m before an instance of interaction, while $m^\bullet$ refers to the datum in m after that instance. I abbreviate sets $\{^\bullet m \mid m \in M\}$ and $\{m^\bullet \mid m \in M\}$ as $^\bullet M$ and $M^\bullet$.

Definition 9 (data variables)**.** *A data variable is an object* x *generated by the following grammar:*

$$x ::= p \mid {^\bullet m} \mid m^\bullet \qquad \textit{(data variables)}$$

$\mathbb{X}$ *denotes the set of all data variables.* $2^{\mathbb{X}}$ *denotes the set of all sets of data variables, ranged over by* X*.*

I assign meaning to data variables with *data assignments*.

Definition 10 (data assignments)**.** *A data assignment is a partial function from data variables to data.* $\mathbb{A}\text{SSIGNM} = \mathbb{X} \rightharpoonup \mathbb{D}$ *denotes the set of all data assignments, ranged over by* σ*.* $2^{\mathbb{A}\text{SSIGNM}}$ *denotes the set of all sets of data assignments, ranged over by* Σ*.*

I proceed by defining constants, functions, and predicates in the calculus. To avoid excessive machinery—but at the cost of formal precision—I do not distinguish constant, function, and predicate symbols from their *interpretation* as data, functions on data, and relations on data [Rau10a]. Instead, I directly refer to data, *data functions*, and *data relations*.

Definition 11 (data functions)**.** *A data function is a function from tuples of data to data.* $\mathbb{F} = \bigcup\{\mathbb{D}^k \to \mathbb{D} \mid \mathbf{true}\}$ *denotes the set of all data functions, ranged over by* f*.*

Definition 12 (data relations). *A data relation is a relation on tuples of data.* $\mathbb{R} = \bigcup\{2^{\mathbb{D}^k} \mid \mathbf{true}\}$ *denotes the set of all data relations, ranged over by* R.

Henceforth, I write elements of $\mathbb{F}$ in camel case monospace (e.g., `divByThree`, `inc`), while I write elements of $\mathbb{R}$ in capitalized camel case monospace (e.g., `Odd`, `SmallerThan`).

Out of data variables, data, and data functions, I construct *data terms*. Every data term represents a datum. This models an operation on (some of) the data involved in an instance of interaction.

Definition 13 (data terms). *A data term is an object* t *generated by the following grammar:*

$$t ::= x \mid d \mid f(t_1, \ldots, t_{k \geq 1}) \qquad \text{(data terms)}$$

$\mathbb{T}\text{ERM}$ *denotes the set of all data terms.* $2^{\mathbb{T}\text{ERM}}$ *denotes the set of all sets of data terms, ranged over by* T.

Henceforth, let $<_{\mathbb{T}\text{ERM}}$ denote some strict total order on $\mathbb{T}\text{ERM}$.

Given a data assignment whose domain includes at least the data variables in a data term t, one can *evaluate* t to a datum. (To evaluate t, additionally, every data function application in t must have the right number of inputs: the *arity* of a data function and its number of inputs must match. Henceforth, I tacitly assume that this always holds true.)

Definition 14 (evaluation). $\mathsf{eval} : \mathbb{A}\text{SSIGNM} \times \mathbb{T}\text{ERM} \to \mathbb{D} \cup \{\texttt{nil}\}$ *denotes the function defined by the following equations:*

$$\mathsf{eval}_\sigma(x) = \begin{cases} \sigma(x) & \textbf{if } x \in \mathrm{Dom}(\sigma) \\ \texttt{nil} & \textbf{otherwise} \end{cases}$$

$$\mathsf{eval}_\sigma(d) = d$$

$$\mathsf{eval}_\sigma(f(t_1, \ldots, t_k)) = \begin{cases} f(\mathsf{eval}_\sigma(t_1), \ldots, \mathsf{eval}_\sigma(t_k)) & \textbf{if } \begin{bmatrix} \mathsf{eval}_\sigma(t_1) \neq \texttt{nil} \\ \textbf{and } \cdots \textbf{ and} \\ \mathsf{eval}_\sigma(t_k) \neq \texttt{nil} \end{bmatrix} \\ \texttt{nil} & \textbf{otherwise} \end{cases}$$

Out of data terms, data relations, and data variables, I finally construct a first-order calculus of data constraints. Although this calculus supports existential quantification, it does not support universal quantification for two reasons. First, universal quantification seems only marginally useful in this thesis (i.e., I do not miss the extra expressiveness that it would provide). More importantly, however, inclusion of universal quantification would complicate computing a particular normal form of data constraints in Chapter 6.

$$\frac{}{\sigma \models \top} \quad (2.1) \qquad \frac{\begin{bmatrix} m \in M \ \textbf{implies} \\ \sigma \models {}^\bullet m = m^\bullet \end{bmatrix} \ \textbf{for all} \ m}{\sigma \models \mathtt{K}(M)} \quad (2.2)$$

$$\frac{\mathsf{eval}_\sigma(t_1) = \mathsf{eval}_\sigma(t_2) \neq \mathtt{nil}}{\sigma \models t_1 = t_2} \quad (2.3) \qquad \frac{(\mathsf{eval}_\sigma(t_1), \ldots, \mathsf{eval}_\sigma(t_k)) \in R}{\sigma \models R(t_1, \ldots, t_k)} \quad (2.4)$$

$$\frac{\sigma \models \chi_1 \ \textbf{and} \ \sigma \models \chi_2}{\sigma \models \chi_1 \wedge \chi_2} \quad (2.5) \qquad \frac{\sigma \models \chi_1 \ \textbf{or} \ \sigma \models \chi_2}{\sigma \models \chi_1 \vee \chi_2} \quad (2.6)$$

$$\frac{\begin{matrix}\mathsf{Free}(a) \subseteq \mathrm{Dom}(\sigma) \\ \textbf{and} \ \sigma \not\models a\end{matrix}}{\sigma \models \neg a} \quad (2.7) \qquad \frac{\sigma \models \phi[d/x] \ \textbf{for some} \ d}{\sigma \models \exists x.\phi} \quad (2.8)$$

$$\frac{\sigma \models \phi_1 \ \textbf{and} \ \cdots \ \textbf{and} \ \sigma \models \phi_k}{\sigma \models \phi_1 \wedge \cdots \wedge \phi_k} \quad (2.9) \qquad \frac{\sigma \models \phi_1 \ \textbf{or} \ \cdots \ \textbf{or} \ \sigma \models \phi_k}{\sigma \models \phi_1 \vee \cdots \vee \phi_k} \quad (2.10)$$

Figure 2.1: Addendum to Definition 16

Definition 15 (data constraints)**.** *A data constraint is an object ϕ generated by the following grammar:*

M	$::=$	*any subset of* $\mathbb{M}$	
a	$::=$	$\bot \mid \top \mid \mathtt{K}(M) \mid t = t \mid R(t_1, \ldots, t_{k \geq 1})$	*(data atoms)*
ℓ	$::=$	$a \mid \neg a$	*(data literals)*
χ	$::=$	$\ell \mid \chi \wedge \chi \mid \chi \vee \chi$	*(data formulas)*
ϕ	$::=$	$\chi \mid \exists x.\phi \mid \phi_1 \wedge \cdots \wedge \phi_{k \geq 2} \mid \phi_1 \vee \cdots \vee \phi_{k \geq 2}$	*(data constraints)*

$\mathbb{DC}$ *denotes the set of all data constraints.* $2^{\mathbb{DC}}$ *denotes the set of all sets of data constraints, ranged over by* Φ*.*

To simplify some of the proofs later in this thesis, the grammar features multiary conjunction and disjunction in addition to their binary versions. Also, negation cannot occur freely but only in data literals, because free occurrences of negation seriously complicate data constraint normalization.

Henceforth, let $<_{\mathbb{DC}}$ denote a strict total order on $\mathbb{DC}$, let $\bigwedge\Phi$ denote the *unique* multiary conjunction of the data constraints in Φ under $<_{\mathbb{DC}}$, and let $\bigvee\Phi$ similarly denote a unique multiary disjunction.

Every data constraint characterizes a set of interaction letters—its semantics—through an *entailment relation*. Let $\phi[t/x]$ denote data constraint ϕ with data term t substituted for every occurrence of data variable x (in a capture-free way).

Definition 16 (entailment). $\models \subseteq \mathbb{A}\text{SSIGNM} \times \mathbb{DC}$ *denotes the smallest relation induced by the rules in Figure 2.1.*

Contradiction, tautology, (multiary) conjunction, and (multiary) disjunction have standard semantics [Rau10a]. Negation $\neg a$ means that, despite all free variables in a having a value, a does not hold true; the extra condition on the free variables in a ensures the *monotonicity* of entailment. Data atom $\mathtt{K}(M)$ means that every memory cell in M keeps the same value before and after an instance of interaction. Data atom $t_1 = t_2$ means that t_1 and t_2 evaluate to the same datum. Typical examples include $p_1 = p_2$ (i.e., the same datum passes through ports p_1 and p_2), $p = m^\bullet$ (i.e., the datum that passes through port p enters the buffer modeled by memory cell m), and $p = {}^\bullet m$ (i.e., the datum in the buffer modeled by memory cell m exits that buffer and passes through port p). Tautology $\top$ means that it does not matter which data flow through which ports.

Henceforth, let $\Rightarrow$ and $\equiv$ denote the implication relation and the equivalence relation on data constraints, derived from $\models$ in the usual way [Rau10a]. Furthermore, let $\mathsf{Variabl}(\phi)$ denote the set of data variables in ϕ, let $\mathsf{Free}(\phi)$ denote its set of *free* data variables, and let $\mathsf{Bound}(\phi)$ denote its multiset of *bound* data variables (i.e., $\mathsf{Bound}(\phi)$ contains as many occurrences of x as the number of quantifiers that bind x in ϕ).

Let X denote a set of data variables. I call a data constraint ϕ *good* under X if (i) ϕ has no free data variables outside X and (ii) the structure of ϕ either allows for moving all existential quantifiers outward, in a semantics-preserving way, or already satisfies such a *prenex normal form*. More precisely, to move existential quantifiers outward, (ii.a) ϕ must have no bound data variables in X and (ii.b) every existential quantifier in ϕ must bind a unique data variable. Typically, X contains all data variables for ports and memory cells in (a transition in) a constraint automaton $\boldsymbol{a}$. By subsequently requiring that all data constraints in $\boldsymbol{a}$ come from $\mathsf{Good}(X)$, condition (i) ensures that the protocol modeled by $\boldsymbol{a}$ cannot affect interaction on ports and memory cells outside its own scope. Condition (ii) plays a role in data constraint normalization, which I discuss in more detail in Chapter 6.

Definition 17 (goodness). $\mathsf{Good} : 2^{\mathbb{X}} \to 2^{\mathbb{DC}}$ *denotes the function defined by the following equation:*

$$\mathsf{Good}(X) = \left\{ \phi \,\middle|\, \begin{array}{l} \mathsf{Free}(\phi) \subseteq X \ \textbf{and} \ \mathsf{Bound}(\phi) \cap X = \emptyset \\ \textbf{and} \ |\mathsf{Bound}(\phi)| = |\mathsf{Bound}(\phi) \cap \mathbb{X}| \end{array} \right\}$$
$$\cup \{\phi \mid \mathsf{Free}(\phi) \subseteq X \ \textbf{and} \ \phi = \exists x_1. \ldots \exists x_l.(\chi_1 \wedge \ldots \wedge \chi_k)\}$$

To understand the previous definition, recall that $\mathsf{Bound}(\phi)$ denotes a multiset, whereas $\mathbb{X}$ denotes an ordinary set. Then, observe that $\mathsf{Bound}(\phi) \cap \mathbb{X}$ contains only the distinct elements in $\mathsf{Bound}(\phi)$. Thus, if the multiset $\mathsf{Bound}(\phi)$ contains more elements than the ordinary set $\mathsf{Bound}(\phi) \cap \mathbb{X}$, at least two existential quantifiers in ϕ bind the same data variable; condition (ii.b) forbids this.

I proceed by defining *first-order constraint automata with memory*, each of which models a protocol, usually called just "constraint automata" in this thesis. Formally, I define a constraint automaton $\boldsymbol{a}$ as a tuple consisting of a set of states Q, a triple of three sets of ports $(P^{\text{all}}, P^{\text{in}}, P^{\text{out}})$, a set of memory cells M, a transition relation $\longrightarrow$, and an initial *configuration* (q^0, μ^0). Set P^{all} contains all ports that participate in the protocol modeled by $\boldsymbol{a}$, while P^{in} and P^{out} contain only its input ports and its output ports (where "input" and "output" qualify ports from the protocol perspective). Although P^{all} contains the union of P^{in} and P^{out}, the converse not necessarily holds true: beside input and output ports, P^{all} may contain also *internal ports*. If a constraint automaton has internal ports, I call it a *composite*; otherwise, I call it a *primitive*.

Definition 18 (states). *A state is an object.* $\mathbb{Q}$ *denotes the set of all states, ranged over by* q. $2^{\mathbb{Q}}$ *denotes the set of all sets of states, ranged over by* Q.

Definition 19 (constraint automata). *A constraint automaton is a tuple:*

$$(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, (q^0, \mu^0))$$

where:

- $Q \subseteq \mathbb{Q}$ *(states)*
- $(P^{\text{all}}, P^{\text{in}}, P^{\text{out}}) \in 2^{\mathbb{P}} \times 2^{\mathbb{P}} \times 2^{\mathbb{P}}$ *such that:* *(ports)*

$$P^{\text{in}}, P^{\text{out}} \subseteq P^{\text{all}} \ \textbf{ and } \ P^{\text{in}} \cap P^{\text{out}} = \emptyset$$

- $M \subseteq \mathbb{M}$ *(memory cells)*
- $\longrightarrow \subseteq Q \times 2^{P^{\text{all}}} \times \mathsf{Good}(P^{\text{all}} \cup {}^{\bullet}M \cup M^{\bullet}) \times Q$ *such that:* *(transitions)*

$$\begin{bmatrix} q \xrightarrow{P,\phi} q' \ \textbf{implies} \\ \phi \in \mathsf{Good}(P \cup {}^{\bullet}M \cup M^{\bullet}) \end{bmatrix} \ \textbf{for all} \ q, q', P, \phi$$

- $(q^0, \mu^0) \in Q \times (M \to \mathbb{D})$ *(initial configuration)*

$\mathbb{A}\text{UTOM}$ *denotes the set of all constraint automata, ranged over by* $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}$. $2^{\mathbb{A}\text{UTOM}}$ *denotes the set of all sets of constraint automata, ranged over by* A, B. $2^{2^{\mathbb{A}\text{UTOM}}}$ *denotes the set of all sets of sets of constraint automata, ranged over by* $\mathcal{A}$.

Figure 2.2 shows a first example. In graphical representations of constraint automata, I annotate ports in synchronization constraints with superscripts "in" and "out" to indicate their polarity; internal ports have no explicit annotation. Henceforth, let $\mathsf{Stat}(\boldsymbol{a})$ denote the state space of a constraint automaton $\boldsymbol{a}$, let $\mathsf{Port}(\boldsymbol{a})$ denote its set of ports, let $\mathsf{Input}(\boldsymbol{a})$ and $\mathsf{Output}(\boldsymbol{a})$ denote its set of input and output ports, let $\mathsf{Memor}(\boldsymbol{a})$ denote its set of memory cells, let $\mathsf{Trans}(\boldsymbol{a})$ denote its transition relation, let $\mathsf{init}(\boldsymbol{a})$ denote its initial configuration, and let $\mathsf{Dc}(\boldsymbol{a})$ denote the set of data constraints that occur on its transitions.

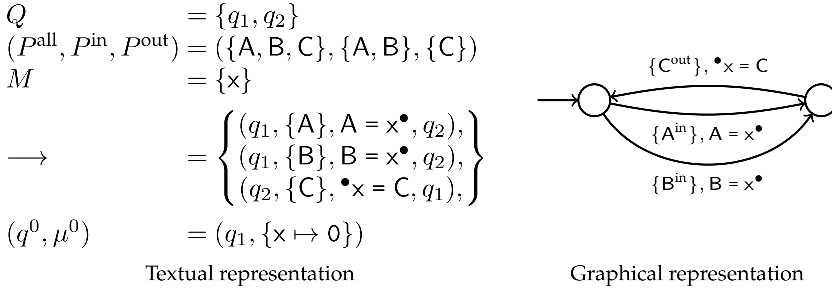

Figure 2.2: Constraint automaton for the LateAsyncMerger2 protocol in the producer/consumer example in Chapter 1. One producer has access to port A, the other producer has access to port B, the consumer has access to port C, and the producers and the consumer use buffer x for temporary storage of data.

"This thesis' constraint automata" (i.e., first-order constraint automata with memory) generalize "original constraint automata" previously developed by Baier et al. [BSAR06]: original constraint automata constitute the subset of this thesis' constraint automata without memory cells and without existential quantification, data functions, and data relations in data constraints. By subsequently removing also data constraints, this thesis' constraint automata further reduce to *port automata*, first studied by Koehler and Clarke [KC09]. (In turn, the semantic domain of the *connector algebras* developed by Bliudze and Sifakis essentially consist of single-state port automata [BS08, BS10].) Extensions of original constraint automata with memory (but still without existential quantification, data functions, and data relations in data constraints) include *constraint automata with state memory*, used in work of Pourvatan et al. and formalized in a categorical setting by Krause et al. [KGdV13, PSAB12, PSHA12]. To my knowledge, Klüppelholz and Baier first articulated the distinction between explicit representations of interaction letters (*concurrent* I/O *operations* in their terminology) and symbolic representations of sets of interaction letters as transition labels [KB09]. Pushing their symbolic representation one step further than here, though, Klüppelholz and Baier in fact combine synchronization constraints and data constraints into single symbolic objects, called I/O *constraints*. As another novelty at that time, their calculus of I/O constraints supports arbitrary data relations (but no data functions or existential quantification). Under some restrictions [Klü12], the model checker developed by Baier et al. can verify original constraint automata extended with data functions and data relations [BBK$^+$10, BBKK09a]. Finally, the encoding of original constraint automata as purely logical constraints developed by Clarke et al. and Proença also supports existential quantification, unary data functions, and data relations, as this thesis' constraint automata [CPLA11, PC13a, PC13b, Pro11].

Behavior, Equivalence, and Congruence

The memory cells in constraint automata may remind one of *stacks* in classical *pushdown automata* [HMU06]: both memory cells and stacks register behaviorally relevant—yet ultimately hidden—information, beyond observable behavior. In defining the *runs* of a constraint automaton, I therefore recall and adopt the following concepts from pushdown automata theory. An *instantaneous description* of a pushdown automaton consists of three elements: its current state, the remaining *input tape*, and the current content of its stack. A pushdown automaton can *move* from one instantaneous description to the next by firing a transition out of its current state, thereby possibly changing its state, certainly consuming the first *input symbol* on the tape (i.e., a letter), and possibly changing its stack. A sequence of successive moves, starting from an initial instantaneous description, results in a run. By replacing "input tape" with "interaction word", "input symbol" with "interaction letter", and "stack" with "set of memory cells", the previous concepts become applicable also to constraint automata. First, I formally define an instantaneous description as a triple (q, w, μ) consisting of a state q, an interaction word w, and a memory snapshot μ.

Definition 20 (instantaneous descriptions). $\mathbb{D}\text{ESCR} = \mathbb{Q} \times \mathbb{W}\text{ORD} \times \mathbb{S}\text{NAPSH}$ *denotes the set of all instantaneous descriptions.*

For a constraint automaton ***a*** with memory cells M to move from an instantaneous description $(q, \lambda w, \mu)$ to another instantaneous description (q', w, μ'), several conditions must hold. Obviously, ***a*** should have a transition (q, P, ϕ, q') from state q to state q'. Second, memory snapshots μ and μ' should have exactly M as their domain (i.e., in making a transition, ***a*** cannot affect memory cells that it does not know about). Third, interaction letter λ should satisfy the synchronization constraint of the transition: λ should have exactly P as its domain. Finally, the data assignment composed of λ, μ and μ' should satisfy data constraint ϕ. These conditions ensure that at least the first instance of interaction in the chain of interaction modeled by interaction word λw respects the protocol modeled by ***a***. Importantly, unless ϕ explicitly states otherwise (e.g., by using the `K` predicate), the content of memory cells in M can nondeterministically change during a move. As an alternative to the previous conditions, ***a*** can also move from (q, w, μ) to (q', w, μ') if a transition from q to q' with an empty synchronization constraint exists: such an *unobservable transition* does not contribute to the observable chain of interaction modeled by w.

Definition 21 (moves-to). $\vdash \subseteq \mathbb{A}\text{UTOM} \times \mathbb{D}\text{ESCR} \times \mathbb{D}\text{ESCR}$ *denotes the smallest relation induced by the rules in Figure 2.3.*

Let w denote an interaction word, and let ***a*** denote a constraint automaton with initial configuration (q^0, μ^0). If ***a*** has an infinite run starting from instantaneous description (q^0, w, μ^0), interaction word w belongs to the interaction language of ***a***. In that case, ***a*** accepts w. Because internal choices in a constraint

$$\frac{\begin{array}{l} q \xrightarrow{P,\phi} q' \\ \textbf{and } \mathrm{Dom}(\mu) = \mathrm{Dom}(\mu') = M \\ \textbf{and } \mathrm{Dom}(\lambda) = P \\ \textbf{and } \lambda \cup \{{}^\bullet m \mapsto \mu(m) \mid m \in M\} \cup \{m^\bullet \mapsto \mu'(m) \mid m \in M\} \models \phi \end{array}}{(q, \lambda w', \mu) \vdash_{(\cdot,\cdot,M,\longrightarrow,\cdot)} (q', w', \mu')} \tag{2.11}$$

$$\frac{\begin{array}{l} q \xrightarrow{\emptyset,\phi} q' \\ \textbf{and } \mathrm{Dom}(\mu) = \mathrm{Dom}(\mu') = M \\ \textbf{and } \{{}^\bullet m \mapsto \mu(m) \mid m \in M\} \cup \{m^\bullet \mapsto \mu'(m) \mid m \in M\} \models \phi \end{array}}{(q, w, \mu) \vdash_{(\cdot,\cdot,M,\longrightarrow,\cdot)} (q', w, \mu')} \tag{2.12}$$

Figure 2.3: Addendum to Definition 21

automaton do not matter for modeling protocols in this thesis (i.e., I model protocols only in terms of observable data-flows on ports), I consider the set of all interaction words accepted by $\boldsymbol{a}$ the behavior of $\boldsymbol{a}$.

Definition 22 (behavior). $\mathsf{Behav} : \mathbb{A}\textsc{utom} \to \mathbb{L}\textsc{ang}$ *denotes the function defined by the following equation:*

$$\mathsf{Behav}(\boldsymbol{a}) = \{w \mid \mathsf{init}(\boldsymbol{a}) = (q, \mu) \ \textbf{ and } \ (q, w, \mu) \vdash_{\boldsymbol{a}} (q', w', \mu') \vdash_{\boldsymbol{a}} \cdots\}$$

The existence of an infinite run of a constraint automaton $\boldsymbol{a}$ on an interaction word w essentially means that the protocol modeled by $\boldsymbol{a}$ admits every step of "the way data is exchanged through ports" modeled by w. As such, $\boldsymbol{a}$—and in particular its transition relation—indeed models "a set of rules that control the way data is exchanged through ports", thereby faithfully capturing the dictionary definition on page 27.

Two constraint automata model the same protocol if those two automata have the same behavior, up to internal choices (i.e., they accept exactly the same interaction words). This intuition induces a straightforward and natural notion of *behavioral equivalence* on constraint automata, based on equality of their accepted interaction languages.

Definition 23 (behavioral equivalence). $\approx\ \subseteq \mathbb{A}\textsc{utom} \times \mathbb{A}\textsc{utom}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\mathsf{Behav}(\boldsymbol{a}_1) = \mathsf{Behav}(\boldsymbol{a}_2)}{\boldsymbol{a}_1 \approx \boldsymbol{a}_2} \tag{2.13}$$

Proving behavioral equivalence between constraint automata plays an important role in establishing the correctness of protocol optimizations, including those presented later in this thesis. As in Milner's work on CCS [Mil89], however, the previous behavioral equivalence based on interaction languages has

a practical problem: although it *does* denote an equivalence relation in its technical sense, $\approx$ *does not* denote a congruence relation under certain operations (discussed shortly). This complicates proving behavioral equivalences.

Inspired by Milner's *bisimulation*—but no less by the variant of bisimulation developed by Baier et al. [BSAR06]—I therefore introduce a congruence relation on constraint automata that subsumes $\approx$. Then, in the rest of this thesis, to prove behavioral equivalence between constraint automata, instead, I prove *behavioral congruence* to imply behavioral equivalence. Keep in mind, however, that only behavioral equivalence truly matters in the end; behavioral congruence just serves as a means to achieve that end.

First, I define the *behavioral preorder* to establish when a constraint automaton $\boldsymbol{a}_2$ *simulates* a constraint automaton $\boldsymbol{a}_1$. In that case, a relation R on the states of $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ exists such that, for every state q_1 of $\boldsymbol{a}_1$, if:

- R relates q_1 to a state q_2 of $\boldsymbol{a}_2$
- and $\boldsymbol{a}_1$ has a transition from q_1 to a state q_1' that admits a set of interaction letters Λ,

then:

- $\boldsymbol{a}_2$ has a transition from q_2 to a state q_2' that admits *at least* the interaction letters in Λ
- and R relates q_1' to q_2'.

In other words, $\boldsymbol{a}_2$ can always simulate every transition that $\boldsymbol{a}_1$ can make, even its unobservable transitions. One may weaken this notion of strong simulation by ignoring unobservable transitions, but I neither need nor pursue such a notion of weak simulation in this thesis.

Definition 24 (behavioral preorder). $\preceq \subseteq 2^{\mathbb{Q}\times\mathbb{Q}} \times \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\begin{array}{l} R \subseteq Q_1 \times Q_2 \ \ \textbf{and} \ \ q_1^0 \ R \ q_2^0 \\ \textbf{and} \ \left[\begin{array}{c} [\left[\begin{array}{l} q_1 \xrightarrow{P,\phi_1}_1 q_1' \\ \textbf{and} \ \ q_1 \ R \ q_2 \end{array}\right] \ \textbf{implies} \ \ \phi_1 \Rightarrow \bigvee \left\{ \phi_2 \left| \begin{array}{l} q_2 \xrightarrow{P,\phi_2}_2 q_2' \\ \textbf{and} \ \ q_1' \ R \ q_2' \end{array}\right.\right\}] \\ \textbf{for all} \ \ q_1, q_1', q_2, P, \phi_1 \end{array}\right] \end{array}}{\begin{array}{l} (Q_1, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow_1, (q_1^0, \mu^0)) \\ \preceq_R (Q_2, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow_2, (q_2^0, \mu^0)) \end{array}} \tag{2.14}$$

(Technically, the ternary relation $\preceq$ does not denote a preorder because of its third operand, but I ignore this minor detail here by abuse of terminology.) Out of the behavioral preorder, I construct the behavioral congruence.

Definition 25 (behavioral congruence). $\simeq \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\big[\boldsymbol{a}_1 \preceq_R \boldsymbol{a}_2 \ \ \textbf{and} \ \ \boldsymbol{a}_2 \preceq_{R^{-1}} \boldsymbol{a}_1\big] \ \ \textbf{for some} \ \ R}{\boldsymbol{a}_1 \simeq \boldsymbol{a}_2} \tag{2.15}$$

The following theorem states that behavioral congruence implies behavioral equivalence. This result shows that I can indeed use $\simeq$ to establish $\approx$.

Theorem 1. $\boldsymbol{a}_1 \simeq \boldsymbol{a}_2$ **implies** $\boldsymbol{a}_1 \approx \boldsymbol{a}_2$

In the next subsection, I present actual congruence results for $\simeq$.

Multiplication and Subtraction

As in all engineering disciplines, *composition*—the act of building more complex objects out of simpler ones—and *abstraction*—the act of hiding objects' irrelevant details—play an important role in software engineering. This holds true also for implementing protocol specifications. Therefore, I define two operations on constraint automata: *multiplication* for composition and *subtraction* for abstraction.

Multiplication consumes two constraint automata $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ as input and produces a constraint automaton as output. I formally define multiplication on constraint automata as a partial function. This partiality models that not all protocols can compose into a new one: two protocols can compose only if (i) each of their *shared ports* serves as an input port in one protocol and as an output port in the other and (ii) these two protocols have no shared buffers. Because constraint automata have a rather involved structure, the formal definition of multiplication may look deceivingly complex. Therefore, I first present a more informal description to explain the main concepts involved. Let $\boldsymbol{a}_1$ denote $(Q_1, (P_1^{\text{all}}, P_1^{\text{in}}, P_1^{\text{out}}), M_1, \longrightarrow_1, (q_1^0, \mu_1^0))$, and let $\boldsymbol{a}_2$ denote $(Q_2, (P_2^{\text{all}}, P_2^{\text{in}}, P_2^{\text{out}}), M_2, \longrightarrow_2, (q_2^0, \mu_2^0))$. Assuming that conditions (i) and (ii) hold true, I take the following steps to multiply $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$.

- First, I take the Cartesian product of Q_1 and Q_2 as the new set of states, and I take the pair of q_1^0 and q_2^0 as the new initial state (inside the new initial configuration).
- Second, I take the union of P_1^{all} and P_2^{all} as the new set of all ports. Subsequently, I put every port in $P_1^{\text{in}} \cup P_2^{\text{in}}$ in the new set of input ports, except those that serve *also* as output ports. Similarly, I put every port in $P_1^{\text{out}} \cup P_2^{\text{out}}$ in the new set of output ports, except those that serve *also* as input ports. Ports with "mixed polarity" (e.g., those that serve as input port in $\boldsymbol{a}_1$ and as output port in $\boldsymbol{a}_2$) become internal ports in the product.
- Third, I take the union of M_1 and M_2 as the new set of memory cells, and I take the union of μ_1^0 and μ_2^0 as the new initial memory snapshot (inside

the new initial configuration). Because (ii) holds true, $\mu_1^0 \cup \mu_2^0$ denotes a well-defined function over domain $M_1 \cup M_2$.

- Finally, I must construct a new transition relation out of $\longrightarrow_1$ and $\longrightarrow_2$. I do so with three rules.

  The first rule states that a transition of $\boldsymbol{a}_1$ involving a shared port can fire iff a transition of $\boldsymbol{a}_2$ involving that same shared port synchronously fires. In other words, $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ must *agree* on synchronously firing transitions involving shared ports. The concept of agreement plays an important role in Chapter 5. Here, I call the kind of agreement required between $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ *weak*. The reason for this particular modifier becomes clear in Chapter 5, where I also introduce a notion of *strong* agreement.

  The second rule states that a transition of $\boldsymbol{a}_1$ involving no shared ports can fire at any time. This means that the protocol modeled by $\boldsymbol{a}_1$ admits the instances of interaction controlled by that transition regardless of the protocol modeled by $\boldsymbol{a}_2$ in the composition of those protocols. Indeed, if the protocol modeled by $\boldsymbol{a}_2$ does not know about a port, it cannot exercise any kind of control over how interaction occurs on that port. (Formally, I should carefully ensure that firings of such a transition in $\boldsymbol{a}_1$ do not affect the memory cells in $\boldsymbol{a}_2$. After all, during every move, the content of memory cells may nondeterministically change unless explicitly stated otherwise.) The third rule states the same as the second rule but with $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ reversed.

For technical convenience—especially later in this thesis—I define multiplication as just explained in three steps. First, I define weak agreement. Second, I define an agreement-parametric multiplication, which takes three instead of two operands: an agreement relation and two constraint automata. Although not directly useful, in Chapter 5, agreement-parametric multiplication enables me to straightforwardly define a different multiplication, based on another form of agreement. Third, I instantiate generalized multiplication with weak agreement.

Definition 26 (agreement)**.** *An agreement relation is a relation $*$ such that:*

$$* \subseteq (2^{\mathbb{P}} \times 2^{\mathbb{P}}) \times (2^{\mathbb{P}} \times 2^{\mathbb{P}}) \ \textbf{and} \ \left[\begin{array}{c}\left[\begin{array}{r}(P_1^{\text{all}}, P_1) * (P_2^{\text{all}}, P_2) \ \textbf{implies} \\ \left[\begin{array}{r}(P_2 \setminus P_1) \cap P_1^{\text{all}} = \emptyset \\ \textbf{and} \ (P_1 \setminus P_2) \cap P_2^{\text{all}} = \emptyset\end{array}\right]\end{array}\right] \\ \textbf{for all} \ P_1, P_1^{\text{all}}, P_2, P_2^{\text{all}}\end{array}\right]$$

$\mathbb{A}$GREEM *denotes the set of all agreement policies.*

$$\frac{q_1 \xrightarrow{P_1,\phi_1}_1 q_1' \ \textbf{and}\ q_2 \xrightarrow{P_2,\phi_2}_2 q_2' \ \textbf{and}\ (P_1^{\text{all}}, P_1) * (P_2^{\text{all}}, P_2)}{q \xrightarrow{P_1 \cup P_2, \phi_1 \wedge \phi_2}_{\circledast} q'} \tag{2.17}$$

$$\frac{q_1 \xrightarrow{P_1,\phi_1}_1 q_1' \ \textbf{and}\ q_2 \in Q_2 \ \textbf{and}\ P_2^{\text{all}} \cap P_1 = \emptyset}{(q_1, q_2) \xrightarrow{P_1, \phi_1 \wedge \mathtt{K}(M_2)}_{\circledast} (q_1', q_2)} \tag{2.18}$$

$$\frac{q_2 \xrightarrow{P_2,\phi_2}_2 q_2' \ \textbf{and}\ q_1 \in Q_1 \ \textbf{and}\ P_1^{\text{all}} \cap P_2 = \emptyset}{(q_1, q_2) \xrightarrow{P_2, \phi_2 \wedge \mathtt{K}(M_1)}_{\circledast} (q_1, q_2')} \tag{2.19}$$

Figure 2.4: Addendum to Definition 28

Definition 27 (weak agreement). $\Diamond \subseteq (2^{\mathbb{P}} \times 2^{\mathbb{P}}) \times (2^{\mathbb{P}} \times 2^{\mathbb{P}})$ *denotes the smallest relation induced by the following rule:*

$$\frac{P_1 \subseteq P_1^{\text{all}} \ \textbf{and}\ P_2 \subseteq P_2^{\text{all}} \ \textbf{and}\ P_1^{\text{all}} \cap P_2 = P_2^{\text{all}} \cap P_1}{(P_1^{\text{all}}, P_1) \Diamond (P_2^{\text{all}}, P_2)} \tag{2.16}$$

Lemma 1. $\Diamond \in \mathbb{A}\text{GREEM}$

Definition 28 (agreement-parametric multiplication).
$\circledast : \mathbb{A}\text{GREEM} \times \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM} \rightharpoonup \mathbb{A}\text{UTOM}$ *denotes the partial function defined by the following equation:*

$$\begin{pmatrix} Q_1, \\ \begin{pmatrix} P_1^{\text{all}}, \\ P_1^{\text{in}}, \\ P_1^{\text{out}} \end{pmatrix}, \\ M_1, \\ \longrightarrow_1, \\ (q_1^0, \mu_1^0) \end{pmatrix} \circledast_* \begin{pmatrix} Q_2, \\ \begin{pmatrix} P_2^{\text{all}}, \\ P_2^{\text{in}}, \\ P_2^{\text{out}} \end{pmatrix}, \\ M_2, \\ \longrightarrow_2, \\ (q_2^0, \mu_2^0) \end{pmatrix} = \begin{pmatrix} Q_1 \times Q_2, \\ \begin{pmatrix} P_1^{\text{all}} \cup P_2^{\text{all}}, \\ (P_1^{\text{in}} \cup P_2^{\text{in}}) \setminus (P_1^{\text{out}} \cup P_2^{\text{out}}), \\ (P_1^{\text{out}} \cup P_2^{\text{out}}) \setminus (P_1^{\text{in}} \cup P_2^{\text{in}}) \end{pmatrix}, \\ M_1 \cup M_2, \\ \longrightarrow_{\circledast}, \\ ((q_1^0, q_2^0), \mu_1^0 \cup \mu_2^0) \end{pmatrix}$$

$$\textbf{if} \begin{bmatrix} P_1^{\text{all}} \cap P_2^{\text{all}} = \\ \quad (P_1^{\text{in}} \cap P_2^{\text{out}}) \cup (P_1^{\text{out}} \cap P_2^{\text{in}}) \\ \textbf{and}\ M_1 \cap M_2 = \emptyset \end{bmatrix}$$

where $\longrightarrow_{\otimes}$ *denotes the smallest relation induced by the rules in Figure 2.4.*

Definition 29 (multiplication). $\otimes : \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM} \rightharpoonup \mathbb{A}\text{UTOM}$ *denotes the partial function defined by the following equation:*

$$\boldsymbol{a}_1 \otimes \boldsymbol{a}_2 = \boldsymbol{a}_1 \circledast_{\Diamond} \boldsymbol{a}_2$$

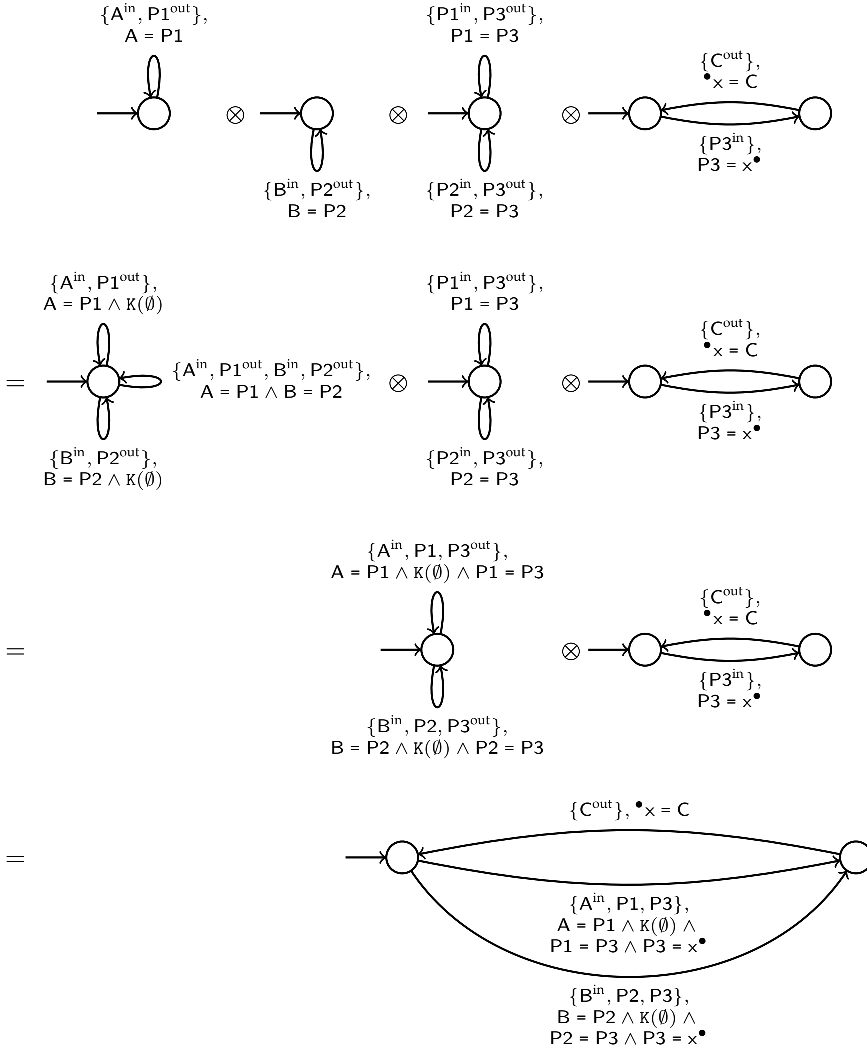

Figure 2.5: Multiplication of four constraint automata

Figure 2.5 shows an example. Henceforth, whenever I write "multiplication", I always mean multiplication as in Definition 29 unless explicitly stated otherwise. Multiplication satisfies commutativity and associativity up-to behavioral congruence.

Essentially, multiplication glues together the constituent protocols modeled by its multiplicands on their shared ports. In particular, every rule involving a shared port in one constituent protocol (i.e., a transition in a constraint automaton) must "synchronize" with every rule involving the same shared port in the other constituent protocol. Such synchronization ensures that every in-

stance of interaction admitted by their composition abides by every relevant rule in *both* constituent protocols. This kind of composition has two interesting properties: *multiparty synchronization* and *indirect synchronization*. Multiparty synchronization means that through successive applications, $\otimes$ can synchronize transitions in one constraint automaton with transitions in multiple other constraint automata. For instance, $\otimes$ synchronizes the transitions in the middle constraint automaton on the second line in Figure 2.5 with transitions in both the left constraint automaton (multiplied on the third line) and the right constraint automaton (multiplied on the fourth line). Indirect synchronization means that through successive applications, $\otimes$ can synchronize transitions in a constraint automaton with transitions in another constraint automaton via a number of "intermediate" constraint automata. For instance, $\otimes$ synchronizes the transitions in the left constraint automaton on the second line in Figure 2.5 with the lower transition in the right constraint automaton (multiplied on the fourth line) via the transitions in the middle constraint automaton (multiplied on the third line). Indirect synchronization enables compositional construction of globally synchronous composites out of locally synchronous primitives.

The following theorems state that $\simeq$ denotes a congruence under $\circledast$ and $\otimes$.

Theorem 2. $\begin{bmatrix} \boldsymbol{a}_1 \circledast_* \boldsymbol{a}_3, \boldsymbol{a}_2 \circledast_* \boldsymbol{a}_4 \in \mathbb{A}\text{UTOM} \\ \textbf{and } \boldsymbol{a}_1 \simeq \boldsymbol{a}_2 \textbf{ and } \boldsymbol{a}_3 \simeq \boldsymbol{a}_4 \end{bmatrix}$ **implies** $\boldsymbol{a}_1 \circledast_* \boldsymbol{a}_3 \simeq \boldsymbol{a}_2 \circledast_* \boldsymbol{a}_4$

Theorem 3. $\begin{bmatrix} \boldsymbol{a}_1 \otimes \boldsymbol{a}_3, \boldsymbol{a}_2 \otimes \boldsymbol{a}_4 \in \mathbb{A}\text{UTOM} \\ \textbf{and } \boldsymbol{a}_1 \simeq \boldsymbol{a}_2 \textbf{ and } \boldsymbol{a}_3 \simeq \boldsymbol{a}_4 \end{bmatrix}$ **implies** $\boldsymbol{a}_1 \otimes \boldsymbol{a}_3 \simeq \boldsymbol{a}_2 \otimes \boldsymbol{a}_4$

Subtraction consumes a constraint automaton $\boldsymbol{a}$ and a port p as input and produces a constraint automaton as output. To subtract p from $\boldsymbol{a}$, I remove p from every set of ports that $\boldsymbol{a}$ consists of, including synchronization constraints of transitions, and I existentially quantify p away in every data constraint.

Definition 30 (subtraction). $\ominus : \mathbb{A}\text{UTOM} \times \mathbb{P} \to \mathbb{A}\text{UTOM}$ *denotes the function defined by the following equation:*

$$\begin{aligned} &(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, (q^0, \mu^0)) \ominus p = \\ &\quad (Q, (P^{\text{all}} \setminus \{p\}, P^{\text{in}} \setminus \{p\}, P^{\text{out}} \setminus \{p\}), M, \longrightarrow_\ominus, (q^0, \mu^0)) \end{aligned}$$

where $\longrightarrow_\ominus$ *denotes the smallest relation induced by the following rule:*

$$\frac{q \xrightarrow{P, \phi} q'}{q \xrightarrow{P \setminus \{p\}, \exists p. \phi}_\ominus q'} \tag{2.20}$$

Figure 2.6 shows an example. (The constraint automaton in Figure 2.6 accepts exactly the same interaction language as the interaction language accepted by the constraint automaton in Figure 2.2, modulo the subtracted ports.)

Subtraction as defined in Definition 30 specializes a more general subtraction on constraint automata, where also the right-hand side denotes a con-

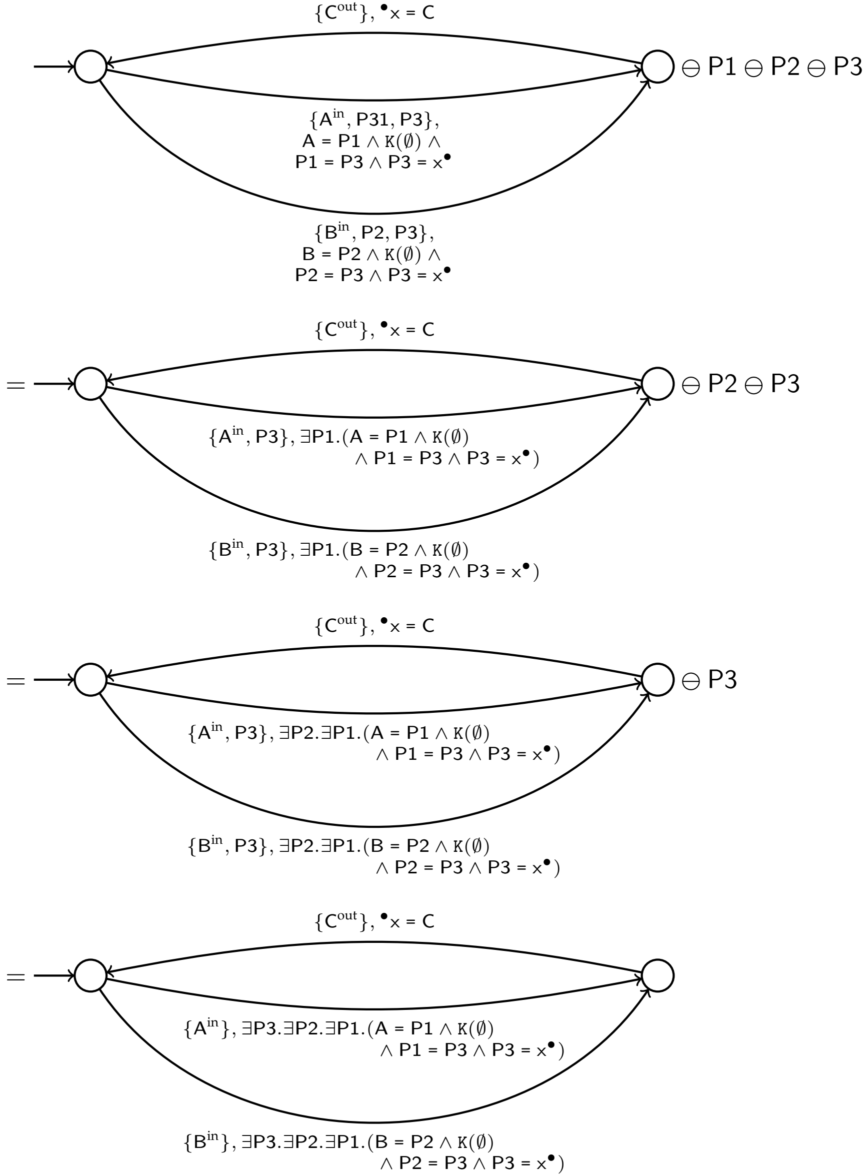

Figure 2.6: Subtraction of ports from the constraint automaton in Figure 2.5

straint automaton (instead of just a port). Because I do not need such general subtraction in this thesis, I skip it here and present only Definition 30.

The following theorem states that $\simeq$ denotes a congruence under $\ominus$.

Theorem 4. $\big[p \in \mathbb{P}$ **and** $\boldsymbol{a}_1 \simeq \boldsymbol{a}_2\big]$ **implies** $\boldsymbol{a}_1 \ominus p \simeq \boldsymbol{a}_2 \ominus p$

The following theorem states that subtraction *of unshared ports* distributes over multiplication up-to behavioral congruence. Consequently, I can incrementally subtract internal ports, which constraint automata cannot share by definition, during a bigger multiplication.

Theorem 5.

$$\begin{bmatrix} p \notin \mathsf{Port}(\boldsymbol{a}_1) \cap \mathsf{Port}(\boldsymbol{a}_2) \\ \textbf{and}\ \ \boldsymbol{a}_1 \otimes \boldsymbol{a}_2 \in \mathbb{A}\mathrm{UTOM} \end{bmatrix} \ \textbf{implies}\ \ (\boldsymbol{a}_1 \otimes \boldsymbol{a}_2) \ominus p \simeq (\boldsymbol{a}_1 \ominus p) \otimes (\boldsymbol{a}_2 \ominus p)$$

Definition 29 of $\otimes$ extends the definition of multiplication on original constraint automata developed by Baier et al. [BSAR06], mainly by accounting for memory cells in Rules 2.18 and 2.19. Pourvatan et al. only informally define a multiplication on constraint automata with state memory, without mentioning a `K`-like predicate to account for nondeterministic changes to the content of memory cells [PSAB12, PSHA12]. Krause et al. generalize the multiplication of Pourvatan et al. as pullbacks in a category of constraint automata with state memory [KGdV13], allowing those automata to synchronize not only on ports but also on memory cells and states. The definition of Krause et al. generalizes also Definition 29, but I do not need this level of generality in this thesis. Finally, Klüppelholz introduces another generalization of the multiplication of Baier et al. that, as Definition 29, takes into account ports' direction [Klü12].

Definition 30 of $\ominus$ significantly differs from the subtraction developed by Baier et al. [BSAR06], which not only removes ports but also eliminates unobservable transitions. Klüppelholz calls the former kind of subtraction *structure-preserving* and the latter kind *aggregating* [Klü12]. Memory cells in constraint automata make elegantly defining aggregating subtraction quite challenging; I leave this for future work.

2.2 Practice

(I have not yet submitted the material in this section for publication.)

I developed a Java library for constraint automata, multiplication, and subtraction, with separate classes for constraint automata (class `Automaton`), their states (class `State`), their transitions (class `Transition`), data constraints (class `Constraint`), ports (class `Port`), and more. Importantly, I use these classes only for *representing* constraint automata and not for running them. In particular, class `Port` does not implement interfaces `InputPort` and `OutputPort` in Figure 1.9; I come to that later, in Chapter 4.

To multiply constraint automata, the library often needs to evaluate the weak agreement relation in Definition 27 (i.e., k_1k_2 times, where k_1 and k_2 denote the number of transitions in the multiplicands). To do this efficiently, in terms of both time and space, I wrote a special data structure for sets. Every

time the library constructs a new Port, it gives this Port a unique positive integer id. Subsequently, it can represent set membership of Ports as a binary string, where a 1 in the i-th position means that a set includes the Port with id i (similarly, a 0 means exclusion). The library can subsequently store such binary strings as arrays of ints (whose length depends on the total number of Ports in the two multiplicands) and perform operations on sets—containment, union, intersection, complementation, difference–using integer arithmetic. This special data structure for sets led to significantly better performance than, for instance, java.util.HashSet, which I used in earlier versions of this library.

Software engineers using the library can construct Automatons either directly or indirectly. In the direct method, software engineers directly use the constructor of Automaton (through a factory design pattern) to obtain an empty Automaton. Subsequently, because they have an Automaton object at their disposal, those software engineers can directly use methods of that object for adding States and Transitions. However, because the direct method exposes Automaton objects to software engineers, those software engineers can use *all* public methods of those objects. For software engineers who work on the library itself (i.e., me), the risks involved seem controllable and reasonably within those software engineers' field of responsibility. For software engineers who merely use the library, in contrast, exposing Automaton objects seems a bad idea. After all, those objects have public methods that only the library itself should invoke. Such software engineers, therefore, need a more controlled environment in which they can construct Automatons.

In the indirect method, software engineers write a subclass of abstract class UserDefinedAutomaton. This abstract class has an Automaton among its private fields (inaccessible from its subclasses) and, through its protected methods (accessible from its subclasses), exposes only methods for adding States and Transitions. The library sets the private Automaton in a UserDefinedAutomaton through a package-visible method (i.e., Automaton and UserDefinedAutomaton must live in the same package for this to work). This indirect method allows software engineers who merely use the library to construct their own Automatons for subsequent multiplication and subtraction in a safe manner.

The library for constraint automata forms a crucial component of the compiler that I present in Chapters 4–8.