Certified Components: Specifying and Reasoning about Formal Protocols

Hans-Dieter A. Hiep

2018

Contents

Preface

- ‘Foundation of Computing and Concurrency’

Chapter 1

Introduction

- The broad research context: foundations of interactive open systems

An open system is not a system that operates in isolation. It allows for information to flow in and out. It is not fully in control of everything required for its operation. Open systems are in contrast to closed systems: a system is closed when everything concerning its operation is assumed, in principle, knowable or predictable.

The assumption of closed systems seems justifiable before, say, the 1980s, when computing systems where large machines taking up the space of a single room that ran in isolation. The information flowing in and out of these machines was fully controlled and monitored.

However, since the 1980s, the assumption of closed systems no longer is justifiable: since widespread adoption of the Internet, computing systems began to run autonomously. The Internet is a well-known example of an open system. One may connect a device to the Internet, and by means of packet switching and routing, information can pass through the system. Every device connected is necessarily an open system, for it is unpredictable what information will be received now or later: the point is that "the network is unpredictable".

- Understanding computing: not pragmatically (what does it produce, what can it be used for), nor empirically (in so-and-so many evaluated cases it seems to work), but as a verificationist (how can you construct it properly).

In this thesis, we introduce components as the elements of a formal system for the construction and analysis of computing systems and their correctness properties.

Any system that implements a computable function, as is understood by the Church-Turing thesis, is a computing system. In this thesis, however, we argue that the class of computing systems is larger than the class of computable functions. The basis of our argument is that we can model an unknown environment that interacts with a computing system under consideration. To illustrate our point, we consider the simple question asked by Abramsky: what does the Internet compute [1]? We do consider the Internet as a computing system, but we do not know what computable function it implements:

To find out what computable function the Internet implements, one needs to inspect all its constituent nodes. In practice, the Internet cannot be completely frozen and inspected off-line: it is very hard to obtain the authorization to perform such action on every connected node. Surely, we must perform an on-line inspection. But this leaves us with many problems: the identification problem (how can one recognize that a node has already been inspected before?), the consistency of snapshots (how can one recognize that a node has not changed since last inspection?), and the myriads of devices (how can one recognize an unknown device?).

So, we conclude that it is not a valid assumption that every computing system is realized by a computable function.

1.1 Formal protocols

We shall consider formal protocols: formal protocols are sets of infinite sequences.

We compare this to formal languages, which are commonly understood as sets of words. Words are finite sequences of symbols of some alphabet. For an introduction see [16, 12]. Recall that a subclass of formal languages are regular languages. Regular languages are closed under union, intersection, (repeated) sequential composition.

Typically, symbols in an alphabet are uninterpreted. However, it is not unusual to consider an alphabet with more structure. For example, in Kleene Algebras with Tests [11], one takes an alphabet with the structure of a Boolean algebra. Each symbol is a proposition, which need not be atomic. Similarly, one could structure the alphabet by taking it to be the set of observations. An observation intends to

represent the observable behavior of a system at a particular instant.

Observations are different than actions. Actions are considered in process algebra (see [10] for an introduction to process algebra). Actions are global events, and a sequence of actions models one particular computation. A language consisting of acceptable sequences of actions models a parallel execution by accepting any interleaving of actions that happen in parallel. We have dependency and independency of actions, modeled by a 'happens before' relation that characterizes which events necessarily occurs before other events. Parallelism is partial commutation of independent actions.

We compare interleaving global actions and observations:

- Observations represent true concurrency. An observation captures interaction, by directly expressing which actions happen at the same instant, alleviating the need for interleaving. This allows one to express more behavior than with actions alone: parallelism (two events happen at the same time) and interleaving (two events happen in an arbitrary order) are now separated.
- Observations are local. An observation of some system does not exclude the possibility that another system also simultaneously acts. This follows from the construction of an observation, being a finite formula, that is unable to express properties of all stream variables. In other words, there always exists some other stream for which its behavior is unspecified within a single observation. Actions are atomic, indivisible and global.

An execution of a system is the set of all acceptable computations. Consider a sequence of observations. Such sequences are finite and represent a finite fragment of an execution. Sequences of observations allow us, in certain cases, to abstract away which particular computation occurs.

An important notion is that of infinite productivity: a system that keeps running indefinitely. In this thesis, what we call a formal protocol is the infinite counterpart of a formal language: a formal protocol is a set of infinite sequences of symbols of some alphabet. As mentioned before, we introduce additional structure to the alphabet by considering infinite sequences of observations.

Just as with formal languages, a subset of formal protocols are regular protocols. Regular protocols are also called ω-regular languages,

in the context of an alphabet consisting of symbols. One way to understand regular protocols is by means of Büchi automata. A Büchi automaton is the infinite counterpart to NFAs, meaning, it accepts or rejects infinite sequences. The acceptence criterium is called the Büchi objective: if an accepting state is visited infinitely often, the infinite run is accepted. The rejection criterium is called the coBüchi objective: if all accepting states are not visited infinitely often, the infinite run is rejected.

Similar to regular formal languages, set membership for regular formal protocols is decidable. This is done by means of playing bisimulation games (see [17, Section 3.5] for an introduction, and [9] for branching bisimulation games). However, we consider more than just regular formal protocols when specifying protocols. In particular in Section 5.3, we see an encoding of a Turing machine as a formal protocol. In the next chapters, we shall look at formal protocols over observations: we take as alphabet the set of observations and define formal protocols as the set of acceptable infinite sequences of observations.

1.2 Certified Components

- Component wrap 'computation'
    - Components are 'rational' or 'irrational'
    - Interaction between components
    - Understanding real computing: open systems, real world environment, 'the network is unpredictable'
    - What is a protocol? Coordination between independent computations
    - Examples: network protocols, distributed algorithms in general, interactive algorithms, concurrency
    - What is correctness? Why study properties of protocols?
    - Security properties (confidentiality, integrity, availability), safety properties (deadlock freedom, absence of bad behavior), lifeness properties (availability, starvation freedom), privacy properties (confidentiality, no cloning)
    - What is a certificate?
    - A proof in a type theory. Computational interpretation.

- What is certification?

1.3 Contributions

The main contributions of this thesis are:

1. To give a formalization of formal protocols, components encapsulating protocols, and properties of protocols. (Chapter 4)
2. Developed methods for showing equivalences and establishing properties of components. (Chapter 5)
3. To provide insight into the notion of duality for components: buffers are dual to prophets. (Chapter 3)

Chapter 2

Preliminaries

This section lays down the mathematical preliminaries required in the rest of this thesis. On the meta-level, we shall work in a dependently typed setting, and in the calculus of constructions to be precise. We assume the reader has a basic familiarity with this calculus, or with a dependently typed programming language such as Coq [4].

For an introduction to dependent type theory, see the book "Type Theory and Formal Proof: An Introduction" by Rob Nederpelt and Herman Geuvers [13]. We shall also refer to concepts from modal logic, such as bisimulations and frames. See the book "Modal Logic for Open Minds" by Johan van Benthem [17].

We consider data types as algebraic structures, such that carrier sets are countable. Our most important point here is the notion of absence of data, denoted by $*$ (Section 2.1).

Streams are used as foundation for understanding productive processes. We consider stream differential equations, stream equality and data stream equivalence (Section 2.2).

We make a distinction between streams and data types. The elements of a data type can be enumerated, resulting in a stream of data elements, and this enumeration is recursive, effective, computable. Streams in general includes enumerations which are not computable. Our motto is: streams are real, data types are rational.

We introduce a typed first-order logic in Section 2.3; its formal semantics is given in Section 4.2.1.

2.1 Data

In this thesis, data is represented by natural numbers. By $\mathbb{N}$ we denote the *natural numbers* $0, 1, 2, 3, \ldots$ We use the notational convention where variables k, l, m, n stand for arbitrary natural numbers, unless specified otherwise.

We assume data is algebraically structured. Whenever we talk about data, we may interchangeably refer to its representation as a natural number or its element in an algebraic structure. We leave encoding and decoding functions implict.

Let a *data type* be a structure $(D, *)$ where D is a countable carrier set and $* \in D$ is a designated constant. We shall speak of *data elements*, *input elements*, or *output elements*, to be those elements different from $*$. Whenever we speak of *elements*, we mean data elements or $*$. Intuitively, one may think of $*$ as standing for the absence of data, or being a 'null' value.

Data is represented by natural numbers. To do so, we have that all data types $(D, *)$ are associated to an *encoding* function $e : D \to \mathbb{N}$ and *decoding* function $d : \mathbb{N} \to D$, such that:

- $e(*) = 0$ and $d(0) = *$ encodes and decodes the absence of data,
- e is injective, i.e. every element has a unique representation,
- $d(e(x)) = x$ for every $x \in D$ and $d(y) = *$ for every $y \notin \mathrm{image}(e)$.

We say that a data type is *finite* if there are more natural numbers than elements; or, equivalently, that d is not injective. Natural numbers not in the image of e are decoded to $*$, and that is only the case for finite data types. Conversely, a data type is *infinite* if e is a bijection. Consequently, for infinite data types, d is the inverse of e.

Given an arbitrary countable set D, we may turn it for free into a data type by adding the element $*$ that is different from every element in D. The resulting data type is said to be *freely generated* by D. Since D is countable, we can find an e and d. Conversely, given a data type $(D, *)$ we may 'forget' its structure and obtain the set $D \backslash \{*\}$ that contains only data elements.

We shall write the Greek letters $\alpha, \beta, \ldots$ to denote data types. As convention, we write $a \in \alpha$ to mean some element a of the carrier set. In the sequel, let $\alpha = (A, *)$ and $\beta = (B, *)$ be arbitrary data types

with carriers A and B. We now turn to constructions of structured data called *regular data types*.

Data type $\alpha + \beta$ consists only of $*$ and the elements $\lfloor a \rfloor$ and $\lceil b \rceil$ for every data element $a \in \alpha$ and $b \in \beta$. Data type $\alpha \times \beta$ consists only of $*$ and the elements (a, b) for every data element $a \in \alpha$ and $b \in \beta$. Data type 0 consists only of $*$, i.e. the carrier is the singleton $\{*\}$. Data type 1 consists only of $*$ and some element tt distinct from $*$.

Equivalently, the data type 0 is freely generated by the empty set $\emptyset$, and the data type 1 is freely generated by any singleton set. Let $\hat{A} =$ and $\hat{B} = B \backslash \{*\}$ be the 'forgetful' sets of data elements of α and β. Then $\alpha + \beta$ is freely generated by the disjoint union $\hat{A} \uplus \hat{B}$, and $\alpha \times \beta$ is freely generated by the Cartesian product $\hat{A} \times \hat{B}$.

Alternatively, one thinks of $\alpha + \beta$ as having as carrier the disjoint union $A \uplus B$ modulo the equivalence $\lfloor * \rfloor \equiv \lceil * \rceil$, setting $* = \lfloor * \rfloor$. Similarly, one thinks of $\alpha \times \beta$ as having as carrier the Cartesian product $A \times B$ modulo the equivalence $(*, b) \equiv (a, *) \equiv (*, *)$, setting $* = (*, *)$.

A function from α to β is a function between their carrier sets, $f : A \to B$, such that $f(*) = *$. Two data types are equivalent if a bijection exists between the two data types, and by $\cong$ we denote the equivalence relation between data types. The operations $(+, \times, 0, 1)$ on data types forms a semiring structure with respect to $\cong$.

Finally, one may think of sequences as repeated products. Let α^n denote the n-repeated product, such that $\alpha^0 = 1$ and $\alpha^{n+1} = \alpha \times \alpha^n$, for some natural number n. By α^* we denote the data type of lists of arbitrary but finite length, where * is known as the Kleene star (not to be confused with the element $*$). We shall denote the data elements of α^* as $[a_1; a_2; \ldots; a_n]$ for data elements $a_1, \ldots, a_n \in \alpha$, and $[]$ for the empty list. We also have the convention that $[\ldots; *; \ldots] \equiv *$, that is, any list containing $*$ is equivalent to $*$.

2.2 Streams

Let a stream be a function from natural numbers to natural numbers. We denote streams by the Greek letters $\sigma, \tau, \ldots$. Intuitively, one thinks of streams as an enumeration. We may define streams directly as a function $\sigma : \mathbb{N} \to \mathbb{N}$. Alternatively, we may define streams as a stream differential equation. See [14, 15] for an elementary introduction.

A stream differential equation for some stream σ is given by its initial value $\sigma(0)$ and its stream derivative σ'. The derivative itself is also a stream such that $\sigma'(x) = \sigma(x+1)$. We have the repeated derivatives σ'', σ''', and so on: we define $\sigma^{(0)} = \sigma$ and $\sigma^{(n+1)} = (\sigma^{(n)})'$. We have that $\sigma(n) = \sigma^{(n)}(0)$. From an initial value and stream derivative we construct the stream $(\sigma(0), \sigma(1), \sigma(2), \ldots)$.

For example, the enumeration $(0, 0, 0, \ldots)$, that repeats 0 forever, is a stream. Given directly as a function, $\sigma(x) = 0$ defines this stream. Given as a stream differential equation, $\sigma(0) = 0$ and $\sigma' = \sigma$ also defines this stream.

Another example is $[n] = (n, 0, 0, \ldots)$, that contains n as initial value followed by zeroes forever. We shall denote this stream by $[n]$. Given directly as a function, $[n](0) = n$ and $[n](x) = 0$ for $x > 0$. Given as a stream differential equation, we have $[n](0) = n$ and $[n]' = [0]$.

There are more streams than natural numbers. The argument is a variation of Cantor's diagonalization argument that there exists more real numbers than natural numbers. Clearly, there are at least as many streams as there are natural numbers: for each natural number n we can construct the stream $[n]$.

Suppose towards contradiction that there are as many natural numbers as there are streams. We enumerate all streams in a table: let σ_0 denote the first stream, σ_1 the second stream, et cetera. Look at the diagonal and construct a stream τ such that $\tau(x) = \sigma_x(x) + 1$. Clearly, this stream τ differs from each enumerated stream σ_x in at least one position, x, thus it cannot be part of the enumeration. This contradicts that there are as many natural numbers as streams.

Intuitively, one may regard a stream as a model of a system in which each stream represents a state. The head of the stream is an observation of the system at that state, and the stream derivative is the next state of the system.

Equality of streams is established by bisimilarity [5]. A relation R on two streams is called a (stream) bisimulation if for all $(\sigma, \tau) \in R$,

$$\sigma(0) = \tau(0) \text{ and } (\sigma', \tau') \in R.$$

Two streams σ, τ are bisimilar if there exists a bisimulation relation R such that $(\sigma, \tau) \in R$, and we write $\sigma = \tau$.

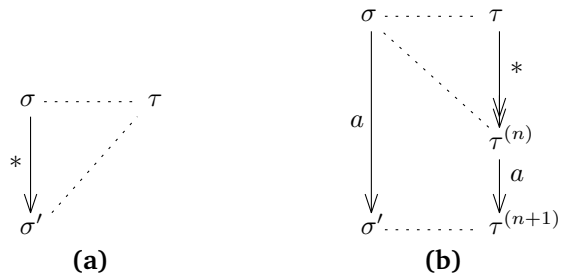

Figure 2.1: *A stream σ is depicted as $\sigma \overset{\sigma(0)}{\to} \sigma'$. A stream prefixed by n values $\tau = (v,\ldots,v,\tau(n),\ldots)$ is depicted as $\tau \overset{v}{\twoheadrightarrow} \tau^{(n)}$. The dotted lines represent branching bisimulation relation R.*

We represent data as natural numbers (Section 2.1). We may consider streams as functions from naturals to data types, say $\sigma : \mathbb{N} \to \alpha$ for some data type α. Such streams are called data streams.

Equivalence of data streams is established by branching bisimilarity. A relation R on two data streams is a (stream) branching bisimulation if for all $m < n$ and $(\sigma,\tau) \in R$,

(a) $\sigma(0) = *$ and $(\sigma',\tau) \in R$, or

(b) $\tau(m) = *$ and $\sigma(0) = \tau(n)$ and $(\sigma,\tau^{(n)}) \in R$ and $(\sigma',\tau^{(n+1)}) \in R$,

see also Figure 2.1. This definition is adapted from [9, 18]. Two data streams σ,τ are branching bisimilar if there exists a branching bisimulation R such that $(\sigma,\tau) \in R$, and write $\sigma \equiv \tau$. Branching bisimilarity is an equivalence relation [9, 6]. Intuitively, data stream equivalence is closed under the removal and insertion of $*$ within a stream.

2.3 Constraints

We consider a typed first-order logic as object language. Constraints are finite formulas in this logic, which is described in this section. In Section 4.1.1, we give a formal definition of formulas, and in Section 4.2.1 a semantics. In here, we only give an explanation that is useful when reading the sections preceding 4.1.1.

We have two kinds of variables: so-called time variables V and so-called port variables P. Time variables are written as $x, y, z, \ldots$, and port variables are written as $X, Y, Z, \ldots$

We consider time expressions to be certain expressions of arithmetic on time variables. Let i, j be *time expressions* formed by the grammar:

$$i, j ::= x \mid n \mid (i \times j) \mid (i + j)$$

where $x \in V$ is an arbitrary time variable. Any natural number n is a valid time expression. We treat parenthesis around arithmetical operations as usual, where $\times$ has a higher precedence than $+$. Time expressions denote references to a particular moment in time, potentially shifted ($+$) or stretched ($\times$). Our time starts at 0, just as every stream starts at index 0.

From time expressions we can construct value expressions. Data expressions denote elements, and are implicitly typed by some data type, say α. Let d, e be *data expressions* formed by the grammar:

$$d, e ::= * \mid c \mid X(i) \mid f(d, \ldots, e)$$

where $*$ is absence of data, c is any data element of α, and f is a function of some fixed arity that has α as codomain. Here, $X \in P$ is an arbitrary stream variable that is interpreted as a data stream of type α. The expression $X(i)$ denotes the element observed in stream X at time i.

We assume that each data type α has a signature consisting of constants $c : \alpha$ and functions $f : \alpha \to \cdots \to \alpha$. Furthermore, we allow any function $f : \alpha_1 \to \cdots \to \alpha_n \to \alpha$ that takes elements of possibly different data types $\alpha_1, \ldots, \alpha_n$ and produces an element of α. These facilities allow us to express operations on data.

Next we consider constraints. Constraints are formulas ϕ, ψ that are formed by the grammar:

$$\begin{aligned} \phi, \psi, \chi ::= {} & (i \leq j) \mid (d = e) \mid \top \mid \bot \\ & \neg\phi \mid (\phi \vee \psi) \mid (\phi \wedge \psi) \mid (\phi \to \psi) \\ & (\exists x.\phi) \mid (\forall x.\phi) \mid (\exists X.\phi) \mid (\forall X.\phi) \end{aligned}$$

where $\exists x.\phi$ and $\forall x.\phi$ binds variable x in subformula ϕ, and similar for $\exists X.\phi$ and $\forall X.\phi$. The notions of subformula and free occurrence are

standard. Abbreviations: $(d \neq e)$ for $\neg(d = e)$, $(i = j)$ for $(i \leq j) \wedge (j \leq i)$, $(i \neq j)$ for $\neg(i = j)$, and $(i < j)$ for $(i \leq j) \wedge (i \neq j)$. Parenthesis are treated conventionally, and binary connectives associate to the right.

We consider a *model* to be an assignment of variables $x, y, z, \ldots$ to natural numbers and an assignment of port variables $X, Y, Z, \ldots$ to data streams. We define the meaning of a formula with respect to a model. A formula restricts which models are satisfactory for the formula to be true. We start out with $\top$ that is always satisfied and $\bot$ that is never satisfied, that is, any model satisfies $\top$ and there is no model that satisfies $\bot$. Similarly, $d = e$ is satisfied if after evaluation of d and e, the resulting elements are equal. We interpret $i \leq j$ as an inequality in arithmetic: $i \leq j$ is satisfied if after evaluation of i and j the inequality holds. For formulas of the other shapes ($\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \rightarrow \psi$, $\exists x.\phi$, $\forall x.\phi$, $\exists X.\phi$, $\forall X.\phi$) we have a Tarskian truth definition.

For now, we identify three important subclasses of constraints:

- simple constraints, in which all occurring time expression are 0, and no quantification occurs.
- observations are simple constraints which are satisfied by a context, unique up to head equality of occurring streams.
- V-sentences, which are formulas without free time variables.

Simple constraints denote properties of the head of stream variables only. We call these constraints simple, because they restrict formulas to the quantifier-free fragment of the logic. Simple constraints captures inconsistency and non-determinism: an unsatisfiable simple constraint is inconsistent, and a simple constraint may be satisfied by more than one context. A simple constraint that is uniquely (up to head equality of occurring streams) satisfied is called an observation. In V-sentences we may quantify over time variables and may use complex time expressions to express constraints that range over time.

Chapter 3

A Library of Components

What is a component? Components are the elements of a formal system that is introduced in this chapter. We interpret each component as a computing system that interacts with some environment. In here, we give a basic intuition and demonstrate numerous components by example. In the next chapter, we formalize our intuition.

One may think of the environment of a component to be an always larger system, and the context in which a component is regarded. Similarly, one may think of a component to always consists of smaller systems that comprise the component. Since neither upwards nor downwards we expect to know everything, we choose the following approach: the deepest components we wish to consider are 'primitives,' and the overarching component that comprises everything we consider is the environment. (cf. Section 4.1.1)

Whatever fits in between the environment and the primitive components are composite components. Composite components are formed by a composition of other components. When we speak the environment of a composite component, we also mean every other sibling component. (cf. Section 4.1.2)

A component has ports that are hidden or visible. Visible ports allow for information to flow inward or outward between the interior of a component and its environment. Hidden ports are called memory. A port that could allow an inward flow of information is an input port, and a port that could allow an outward flow of information is an output port, and a port is either an input port, or an output port, or a memory. (cf. Section 4.1.3)

t	X	Y
0	d	$*$
1	$*$	$*$
2	$*$	$*$
3	$*$	d

(a)

t	X	Y
0	$*$	$*$
1	d	$*$
2	$*$	d

(b)

t	X	Y
0	d	$*$
1	$*$	d

(c)

Table 3.1: *Examples of observations*

The question we must ask ourselves is: who is in control of a port? The control of information flow is a shared responsibility between the component and its environment. Intuitively, information flow is a coordination game. Say, the environment initiates an inward flow, then the component may choose to block the inward flow, as if it applies backpressure, or allow it. Similarly, if the component initiates an inward flow, then the environment may choose to block the inward flow or allow it. The same game applies for outward flow. (cf. Section 4.2.2)

The flow of information at a port is modeled by data streams. For each port there is an associated data stream. The type of data that may flow through a port is the data type of the stream. Each data type consists of a special 'null' value $*$ that represents the absence of data, and within data streams it is used to indicate that no information is flowing at a moment. (cf. Section 4.2.1)

A component has a number of ports and we intend to observe all ports simultaneously. Intuitively, we consider a snapshot of a component's ports that together forms an observation. All information captured in the snapshots over time is assumed to be consistent and complete. A consistent snapshot abstracts any ordering of information flow and is a faithful representation of that what actually happened. A complete snapshot excludes any hidden information flows that are not captured over time. (cf. Section 4.2.3)

We say that a port 'fires' if there was an actual data element exchanged through the port. We say a port is 'inhibited' or 'blocked' if no data element is exchanged through the port: this is represented by the 'null' value $*$ in the stream corresponding to the port.

A useful way of thinking of a component and its ports is by considering a tabulation of observations. Say, we have an arbitrary component with two ports, X and Y. Here X is an input port, and Y is an output port. We shall observe it over some period of time. In Table 3.1b we see three observations: in the first observation (at $t = 0$) both X and Y have not fired. The second observation (at $t = 1$) shows some data element d flows through port X. Since X is an input port, the element d is supplied by the environment and deemed acceptable by the component: the environment and component have completed their coordination game and the result is that the data element d is exchanged through port X. The last observation (at $t = 2$) shows that the data element d, which is the same element as previously, flows out of output port Y: again a coordination game is completed.

The notion of consistency here means that the environment and component never get stuck in their coordination game. An environment and a component become stuck whenever they present contradicting constraints on the flow of data: say, the environment forces the inward flow at port X but the component inhibits any inward flow at port X. The result is an inconsistency. In case of any inconsistency, the behavior of the component is undefined: we say the component is 'destroyed' in case of inconsistency.

The notion of completeness here means that there was no hidden information flow: all information that flowed in or out the component between $t = 0$ and $t = 4$ is as shown. For example, this implies that between time $t = 0$ and $t = 2$, there was no activity at port Y. It may, however, be possible that the component and environment are playing a coordination game at port Y, but it only completed succesfully at $t = 2$: at all other times, either the environment or the component was blocking or inhibiting the port.

Components are building blocks. Components define protocols. Composite components are formed by expressions. We consider primitive components as indivisible building blocks (Section 3.1), and composite components formed by composition expressions denoted as circuits (Section 3.2). We employ a structured recursive approach to composition: these are the inductive components (Section 3.3).

Combining components to form a composite component is called composition. We shall see that input and output ports can be linked together to facilitate the exchange of data between them, causing for

a resolution of the coordination game to expand beyond what was initially considered the environment. (cf. Section 4.3.1)

We assume in this section that the environment of a component is ideal. An ideal environment is maximally consistent with respect to some component. Intuitively, this means that an ideal environment has the least restrictions on input and output ports, allowing all data elements to flow in that the component accepts, accepting any data elements that flows out of the component. (cf. Section 4.3.2)

In this chapter and all examples of components we shall see, it is not immediately clear how to realize these components. Concretely, and in practice, one would program an existing computing system to implement components. Readers are expected to be readily able to implement (in their favorite programming language) the components demonstrated here, and therefore we avoid any such practical examples. The point of giving these definitions is to logically analyze protocols. Theoretically, we implement components by providing a certificate. A certificate corresponds to a prototypical program that realizes a certified component. (cf. Section 4.3.3)

Before we embark in our library of components, we mention that components in this chapter have the property of independent progress. Independent progress is a property preserved by composition. We consider the sequence of observations in Table 3.1a and 3.1b and 3.1c to be semantically equivalent. Intuitively, semantic equivalence guarantees that each component can be stopped and resumed at arbitrary times: independent progress captures such guarantee. (cf. Section 5.2.1)

3.1 Primitive Components

In this section we will give a number of primitive components. Primitive components are also called primitives. The primitives considered in this section are not exhaustive: other primitives do exists.

A component is defined by giving a set of input and output ports and specifying a protocol. Components run forever, and we consider the streams of observations at its ports. A protocol is a set of streams of observations. A component's protocol restricts the behavior allowed by the specified component. The streams of observations is called acceptable if and only if it is in the protocol.

Whenever we consider a particular stream of observations, we may illustrate only a subsequence of the whole stream. This is done in the table format as seen before. Tables only show a particular part of an acceptable stream: whatever is omitted in the table is left undefined. We may use the meta-variables $d, e \ldots$ standing for data elements, and $*$ standing for the absense data.

To specify a protocol, and thus the acceptable streams of observations, we use V-sentences: formulas without free time variables. In Section 4.1.1, we give a formal semantics of formulas.

The format we employ to define components is the following:

Component:	*name*(*parameters*)
Input ports:	*description of input ports*
Memory:	*description of memory*
Output ports:	*description of output ports*
Protocol:	*formula*

The name signifies the name of the component defined. Parameters are meta-variables that are data types $\alpha, \beta, \ldots$ and (data) elements $a, b, \ldots$: these variables occur as placeholder in the definition. The port and memory descriptions specifies the names of the stream variables, and their type. Finally, the protocol is given as a V-sentence, where free stream variables are as designated by the component definition.

When we refer to a component, we either refer to its definition without giving any actual parameters, or we instantiate it by giving actual parameters: actual data types and actual (data) elements.

A component definition describes how the coordination game is played by the component defined. The elements flowing in input ports are determined by an environment; the component's protocol specifies when and what elements are allowed. Similarly, elements flowing out of output ports are determined by the component, as defined by its protocol. We work in an ideal environment.

We also introduce the notion of dual components. A component is dual to another component if and only if the former is defined by the same protocol as the latter, but has its input and output ports swapped.

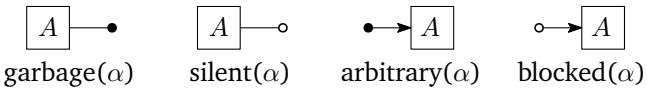

Figure 3.1: *Endpoints*

3.1.1 Endpoints

We introduce unary primitive components: endpoints. An endpoint has a single port that is either input or output. The endpoints introduced here are depicted in Figure 3.1.

Component:	garbage(α)
Input ports:	X of α
Output ports:	none
Protocol:	$\top$

A garbage can accept any data and throws it away. It is wasteful: any information it accepts is destroyed. Every stream of observations is acceptable. During the coordination game, a garbage accepts any constraint by the environment on its input port.

A silent never produces any output. The component accepts only streams of observations when port X is always $*$. The coordination game is played by enforcing the output not to fire: it is inconsistent if the environment forces any outward data flow.

Component:	silent(α)
Input ports:	none
Output ports:	X of α
Protocol:	$\forall t.X(t) = *$

We have dual components, where input and output are swapped. The dual of garbage is arbitrary, and the dual of silent is blocked.

Component:	arbitrary(α)
Input ports:	none
Output ports:	X of α
Protocol:	$\top$

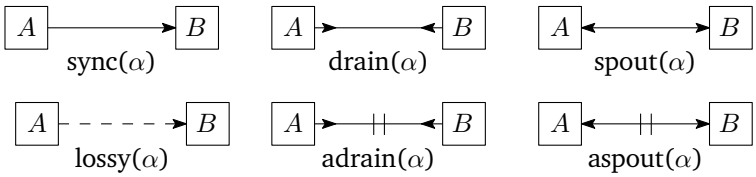

Figure 3.2: *Channels*

An arbitrary has an output without constraints. The protocol is the same as that of garbage by duality. However, it's operation is different than garbage. Arbitrary is an oracle, and produces every possible element non-deterministically. What is a possible output depends on the outcome of the coordination game.

Blocked is dual to silent, and thus have the same protocol: the component accepts only streams of observations when port X is always $*$. A blocked restricts its input to ensure it never produces anything: during the coordination game, this constraint is shared with the environment. When the environment forces any inward data flow, an inconsistency occurs.

Component:	blocked(α)
Input ports:	X of α
Output ports:	none
Protocol:	$\forall t.X(t) = *$

These four components are together called the unit components. It turns out that the components given here are units with respect to composition with nodes. We refer to Section 3.2.1 for a detailed explanation.

It is worthwhile to think of components that have multiple input ports or multiple output ports. We shall now work in that direction by considering components with two ports.

3.1.2 Channels

In this section we introduce binary primitive channels: channels. A channel has two ports. We distinguish three kinds of channels: channels with one input port and one output port, channels with

A	B
d	d

A	B
d	$*$

Table 3.2: *Example observations of lossy*

two input ports, and channels with two output ports. The channels introduced here are depicted in Figure 3.2.

A synchronous channel transports anything instantaneously from its input port to its output port. A channel is self-dual: we swap its input and output by reversing the arrow.

Component:	sync(α)
Input ports:	A of α
Output ports:	B of α
Protocol:	$\forall t.A(t) = B(t)$

A lossy synchronous channel either transports instantaneously, or its input is lost in transit, non-deterministically. This component alone does not guarantee that input arrives at its output.

Component:	lossy(α)
Input ports:	A of α
Output ports:	B of α
Protocol:	$\forall t.A(t) = B(t) \vee B(t) = *$

As an example, consider that the protocol for lossy accepts streams that could contain either of the subsequences of streams shown in Table 3.2. The left table shows that a data element d from input A is transported instantaneously to output B. The right table shows that a data element d from input A is lost. Both of these subsequences of behavior are acceptable, for any data element d. Note that these two cases correspond to the disjunction in the protocol of lossy.

The former channels have one input port and one output port. In contrast, the next channels have either two inputs or two outputs.

Synchronous drain loses all its input with the purpose of synchronization: either both input ports pass data instanteneously or both ports are absent of data. Passing data need not be related.

Component:	drain(α, β)
Input ports:	A of α, B of β
Output ports:	none
Protocol:	$\forall t. A(t) = * \leftrightarrow B(t) = *$

An asynchronous drain also loses all its input for the purpose of synchronization. At most one input port passes data instantaneously. Passing data is not related.

Drains could be implemented by playing the coordination game as follows. For synchronous drain: if only one input is ready to fire (the environment signals this to the component) it becomes blocked until the other input is also ready to fire. This guarantees that at one instant, both ports fire together. For asynchronous drain: if either input is ready to fire, the component picks precisely one input to fire and blocks the other one.

Component:	adrain(α, β)
Input ports:	A of α, B of β
Output ports:	none
Protocol:	$\forall t. A(t) \neq * \rightarrow B(t) = * \wedge$ $B(t) \neq * \rightarrow A(t) = *$

An asynchronous drain is not the dual of a synchronous drain because they have different protocols.

Synchronous spout has arbitrary outputs, but with the same protocol as synchronous drain: it either generates two unrelated data elements at the same time or both ports are silent. Synchronous spout is the dual of synchronous drain.

Component:	spout(α, β)
Input ports:	none
Output ports:	A of α, B of β
Protocol:	$\forall t. A(t) = * \leftrightarrow B(t) = *$

An asynchronous spout is the dual of an asynchronous drain: it also generates data, but never at the same time at both ports.

Component:	aspout(α, β)
Input ports:	none
Output ports:	A of α, B of β
Protocol:	$\forall t. A(t) \neq * \rightarrow B(t) = * \wedge$
	$B(t) \neq * \rightarrow A(t) = *$

We analyze two properties of components. The first property is synchronization, the second property is transportation. Synchronization is classified as: synchronous, asynchronous, or both. Transportation is classified as: instantaneous, non-instantaneous, or both.

An intuitive, and etymologically correct, metaphor for synchronization is the case of two clocks that always tick at the same time (synchronous), or never at the same time (asynchronous).

Synchronous channels relate ports such that either all port activity happens at the same time, or nothing happens at all. Asynchronous channels also relates ports by stating that port activity happens at only one port excluding at the same time any other port activity.

Transportation involves the movement of data elements. All channels we have seen until now are instantaneous: data elements move from input ports to output ports, that is, in space but without delay. In the next section we consider non-instantaneous channels, that move data elements in space and time.

The reader may object, by saying that instantaneous channels are practically impossible, that it is impossible to move data in space without considering time. The purpose of our theory is to consider synchronization and transportation as an abstraction.

3.1.3 One-place Buffers

A channel that deserves special attention is the buffer, which transports non-instantaneously. In this section we consider a buffer, and three variants. These components are depicted in Figure 3.3.

The components we consider in this section are stateful: it has memory that is either empty or full. Recall that memory is a hidden port, that is, it is part of the observation but it can not be influenced by the environment. Components are completely in control over memory. However, memory being a hidden port, we still have stream variables in our protocol corresponding to them.

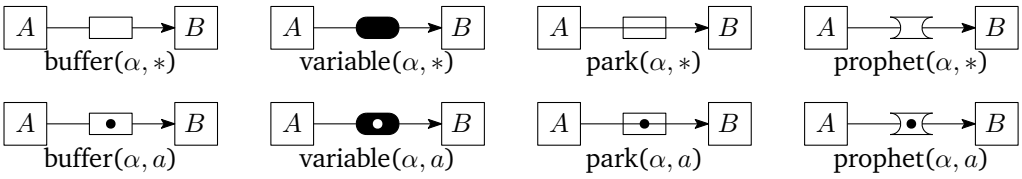

Figure 3.3: *Buffers and three variants: the dot inside marks full memory.*

Component:	buffer(α, a)
Input ports:	A of α
Memory:	M of α
Output ports:	B of α
Protocol:	$M(0) = a \wedge$

$$\forall t. \quad B(t) = * \quad \wedge M(t) = * \wedge M(t+1) = A(t) \quad \vee$$
$$A(t) = * \wedge B(t) = * \quad \wedge M(t) \neq * \wedge M(t+1) = M(t) \vee$$
$$A(t) = * \wedge B(t) = M(t) \wedge M(t) \neq * \wedge M(t+1) = *$$

Buffers transport data elements over time in a non-instantaneous way. One only observes an output element if in the past it was put in.

The output of a buffer must remain silent until some input element passes to memory. A buffers remembers its element indefinitely while the output remains silent. A buffer destructively reads its memory when a data element is put out. A buffer is asynchronous: never there is activity at both its input and output ports. Additionally, the input port is only blocked when the buffer is full.

We illustrate the buffer in Table 3.3a: first the input port fires with d, while the output port must block. The data element is stored in memory: as long as the output does not fire, memory is retained. As long as the buffer is full, the next input remains blocked. Then the output fires with d, the element stored in memory, and the memory is cleared in the next observation. Looking just at the input and output ports; we have Table 3.3b. Also see Table 3.1 for an earlier example.

The first variant of a buffer channel is a variable channel. It differs from a buffer in only two respects: a variable's input is not blocked when the buffer is full, and memory is not read destructively.

We illustrate the variable in Table 3.3c: first the input fires with e; the output port must block. Then the input fires d, which overwrites the previous memory, and the output must still block. When there

X	M	Y
d	$*$	$*$
$*$	d	d
e	$*$	$*$
$*$	e	e

(a) *buffer*

X	Y
d	$*$
$*$	d
e	$*$
$*$	e

(b) *buffer*

X	Y
e	$*$
d	$*$
$*$	d
$*$	d

(c) *var*

X	Y
$*$	d
$*$	$*$
d	$*$

(d) *prophet*

Table 3.3: *Example observations of buffers and two variants*

is no input, the output may fire; it fires d but it does not erase the memory. Hence, the next round when there is no input, the output port may fire again with d.

If a variable is full, any input element overwrites the existing value in memory. Additionally, if a variable is full, it remains full.

Component: variable(α, a)
Input ports: A of α
Memory: M of α
Output ports: B of α
Protocol: $M(0) = a \wedge$

$$
\begin{aligned}
\forall t.\quad & B(t) = * && \wedge M(t) = * \wedge M(t+1) = A(t) && \vee \\
& A(t) = * \wedge B(t) = * && \wedge M(t) \neq * \wedge M(t+1) = M(t) && \vee \\
& A(t) = * \wedge B(t) = M(t) && \wedge M(t) \neq * \wedge M(t+1) = M(t) && \vee \quad \text{(!)} \\
& A(t) \neq * \wedge B(t) = * && \wedge M(t) \neq * \wedge M(t+1) = A(t) && \text{(!)} \\
& A(t) \neq * \wedge B(t) = M(t) && \wedge M(t) \neq * \wedge M(t+1) = A(t) && \text{(!)}
\end{aligned}
$$

Another variant of a buffer is a park channel. Parks differs from buffers in only one respect: a park allows input and output to be synchronous if the park is empty, and this is not alllowed by buffers. Buffers are strictly asynchronous, in the sense that the input port and output port never fire at the same time. Parks are both synchronous and asynchronous.

Depending on the outcome of the coordination game, a park stores the input for later retrieval if the output port cannot fire, and in this case the park is asynchronous. A different outcome means that input and output may fire together, and the park is synchronous.

Component:	park(α, a)
Input ports:	A of α
Memory:	M of α
Output ports:	B of α
Protocol:	$M(0) = a \wedge$

$$\begin{aligned}\forall t.\quad & B(t) = * \quad \wedge M(t) = * \wedge M(t+1) = A(t) \quad \vee \\ & A(t) \neq * \wedge B(t) = A(t) \quad \wedge M(t) = * \wedge M(t+1) = * \quad \vee \quad \text{(!)} \\ & A(t) = * \wedge B(t) = * \quad \wedge M(t) \neq * \wedge M(t+1) = M(t) \vee \\ & A(t) = * \wedge B(t) = M(t) \wedge M(t) \neq * \wedge M(t+1) = *\end{aligned}$$

Only buffer and variable are asynchronous. Park is both synchronous and asynchronous. Differences in protocol with buffer are marked (!).

Finally, we consider the dual to buffer: a prophet. It has the same protocol as the buffer, but input and outputs are swapped.

Component:	prophet(α, a)
Input ports:	A of α
Memory:	M of α
Output ports:	B of α
Protocol:	$M(0) = a \wedge$

$$\begin{aligned}\forall t.\quad & A(t) = * \quad \wedge M(t) = * \wedge M(t+1) = B(t) \quad \vee \\ & B(t) = * \wedge A(t) = * \quad \wedge M(t) \neq * \wedge M(t+1) = M(t) \vee \\ & B(t) = * \wedge A(t) = M(t) \wedge M(t) \neq * \wedge M(t+1) = *\end{aligned}$$

Prophets first fire their output port and speculatively generate a data element. The input port fires after the prophet has made a prediction, and only the output which had predicted the input correctly is consistent—all other speculations that are incorrect are inconsistent.

The reader may object, by saying it is impossible that data travels back in time, as it does with the prophet. The prophet merely defines the protocol—in practice, one could implement a prophet, say, by speculative execution and backtracking if the wrong element was chosen.

We analyze two another properties of components. A component is linear if for every input port, every input element maps uniquely to the same output element at some output port at some time. It is causal if for every output port and every output element, there exists an equal input element at some input port at some time in the past.

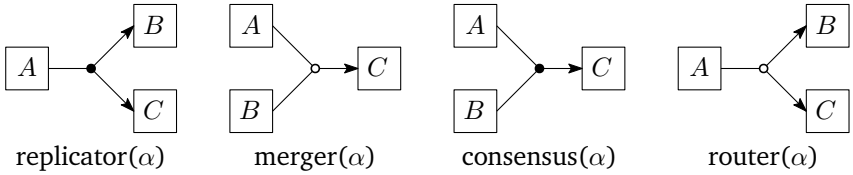

Figure 3.4: *Nodes*

Intuitively, we keep track of every input element and consider where and when it is output: linearity implies no duplication and no loss. Causality implies that an input element is either forgotten, or is output somewhere in the future: but input never leads to output in the past.

Buffer, park and prophet are linear channels: every input is output once and once only. A variable is not linear: an input element may never appear as output because it may be overwritten, or an input element may appear as output multiple times. Buffer, variable and park are causal channels, and prophet is not a causal component.

3.1.4 Binary Nodes

The last primitive components we consider are nodes. Nodes are ternary components and have one input and two outputs, or two inputs and one output. Nodes are used to graphically connect components (Figure 3.4).

Nodes are a generalization of channels to three endpoints. In the next sections we explore further generalizations of synchronous channels into components that consist of more than three ports.

The nodes we consider are instantaneous components: data elements move between inputs and outputs without delay.

A replicator component transports instantaneously by duplicating data elements from its input port to two output ports.

Component:	replicator(α)
Input ports:	A of α
Output ports:	B, C of α
Protocol:	$\forall t. A(t) = B(t) \wedge A(t) = C(t)$

A merger transports instantaneously at most one data element from one input to its output, while the other input is blocked. If we analyze the input ports of a merger in an ideal environment, we establish they must be asynchronous: if both A and B fire at the same time, we have an inconsistency.

Component:	merger(α)
Input ports:	A, B of α
Output ports:	C of α
Protocol:	$\forall t. A(t) = C(t) \wedge B(t) = * \vee$
	$B(t) = C(t) \wedge A(t) = *$

We also have the duals of replicator and merger, where input and output are swapped, similar to the duals of endpoints.
A consensus requires two inputs to always agree on all elements, and instantaneously transports the agreed element to a single output.

Component:	consensus(α)
Input ports:	A, B of α
Output ports:	C of α
Protocol:	$\forall t. A(t) = C(t) \wedge B(t) = C(t)$

A router transports an input element to exactly one output port.

Component:	router(α)
Input ports:	A of α
Output ports:	B, C of α
Protocol:	$\forall t. A(t) = B(t) \wedge C(t) = * \vee$
	$A(t) = C(t) \wedge B(t) = *$

We can also analyze linearity for nodes. If an input element appears to be duplicated, or not output at all, the component is not linear.

Both merger and router are linear, since every input element occurs at exactly one output. Both replicator and consensus are not linear. A replicator duplicates its input element, and a consensus loses one of its input elements.

Overview

In this section we introduced nineteen primitive components. Primitive components serve as basic building blocks for protocols. Later, we shall introduce more primitive components when necessary.

The primitives considered in this section are not exhaustive. In principle, any formulated protocol together with an input and output designation of its ports can be considered as a primitive. In Section 3.2, we shall compose components to form composite components. Some of the primitive components given here are equivalent to compositions of other primitives (see Section 5.1.2).

Furthermore, the four properties we have considered (synchronization, transportation, linearity and causality) are not yet defined in a precise manner. We will formalize these properties in Section 5.2 where we study them more closely.

Table 3.4 is an overview of the primitive components in this section.

3.2 Composite Components

Composition is a binary operation on components: given two components, their composition forms another component. The resulting component is called a composite component, and a composite component is always the result of some composition.

- Grammar of compositions (see Section 4.1.2).

Composition of two protocols can be understood as the conjunction of the two protocols. In this case, the resulting set of streams of observations is the intersection of the two sets of streams of observations. Intersection restricts behavior, since the streams corresponding to a shared port now have to comply to two protocols.

For components, we understand composition as two operations. Parallel composition is a binary operation to juxtapose two components to form a new component, and identification is a unary operation on components, taking an input port and an output port of the same data type, and forms a new component in which those ports are linked together and hidden.

For parallel composition, we may assume that the two components to be juxtaposed do not share any port names to prevent name clashes

	#	(I/O)	sync	instant	linear	causal
garbage	1	(1/0)	✓	✓		✓
silent	1	(0/1)	✓	✓	✓	✓
arbitrary	1	(0/1)	✓	✓		
blocked	1	(1/0)	✓	✓	✓	✓
sync	2	(1/1)	✓	✓	✓	✓
lossy	2	(1/1)		✓		✓
drain	2	(2/0)	✓	✓		✓
adrain	2	(2/0)	✗	✓		✓
spout	2	(0/2)	✓	✓		
aspout	2	(0/2)	✗	✓		
buffer	2	(1/1)	✗	✗	✓	✓
variable	2	(1/1)	✗	✗		✓
park	2	(1/1)			✓	✓
prophet	2	(1/1)	✗	✗	✓	
replicator	3	(1/2)	✓	✓		✓
merger	3	(2/1)		✓	✓	✓
consensus	3	(2/1)	✓	✓		✓
router	3	(1/2)		✓	✓	✓

Table 3.4: *A non-exhaustive table of primitive components. The first columns display #number of visible ports, (input ports/output ports). The third column displays the synchronization property: synchronous (✓) or asynchronous (✗) or both. The fourth column displays the transportation property: instantaneous (✓) or non-instantaneous (✗) or both. The last columns display the property of linearity (✓) and the property of causality (✓).*

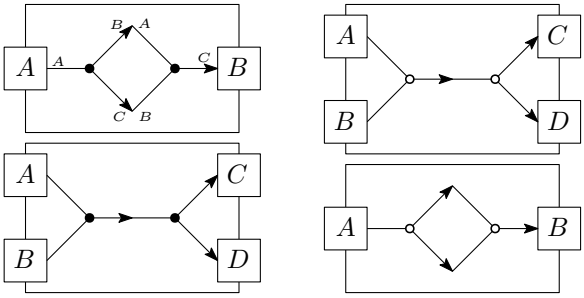

Figure 3.1: *Composing nodes and their dual*

by suitable renaming of ports. As a result, neither protocol shares any variable with the other protocol, and we can take as protocol the conjunction of the protocols of its constituent components. This composition preserves the behavior of the two constituent components.

- Algebraic identities of compositions

After parallel composition, one performs identification. Identification of two ports intentionally restricts the behavior of a component. The linked input port and output port are renamed to a fresh name which becomes a memory. Identification hides the ports from further linking. Only the behavior of the original component for which all observations at both input and output port agree is retained.

- Intuitive explanation of composition in terms of sets of tables/protocols.

3.2.1 n-ary Nodes

The first composed nodes we consider are shown in Figure 3.1. Each composition is depicted as a box, on which the ports on the boundary are input and output ports of the resulting composition.

For the top-left composition, we explain the construction. We take the juxtaposition of two new instances of the components defined in the previous section: replicator(α) and consensus(α). Replicator output ports B and C are linked to the consensus input ports A and B, respectively. The replicator input port A is the shown input port A, and the consensus output port C is the resulting output port B.

Port names are annotated in the diagram. Such annotations quickly become tedius, and we leave them out if doing so is unambiguous.

- compsing with units (endpoints)
- 3-ary nodes

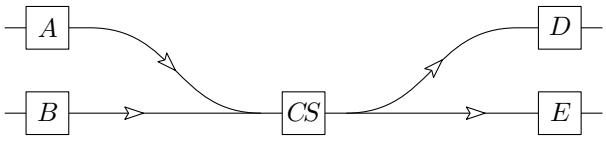

Figure 3.2

- generalize to n-ary nodes (replicator, merger, consensus, router)
- useful identities of nodes (formally established in 5.1.3)

3.2.2 Alternator & Sequencer

- examples from literature
  - uses data from input in buffer

3.2.3 Variable

- alternative construction (from literature)
  - useful identity (formally established in 5.1.2)

3.2.4 Mutual Exclusion

Consider a railroad[1] as depicted in Figure 3.2. Suppose there are only two trains, one at A with destination D and the other at B with destination E. Both trains want to enter central station on the same track. To prevent collision, only a single train may proceed. Suppose we let the train at B proceed first. After visiting CS, the train proceeds to E. Now CS is clear again. Next, the train at A may proceed to CS, and after visiting CS it proceeds to D. We obtain a similar sequence of observations when we we let the train at A proceed first.

Again consider Figure 3.2 but now suppose there are only three trains: at $A, B,$ and D. Again the trains at A and B have destinations D and E, thus want to enter Central Station. Let the train at A proceed first. This train enters CS but may not leave, since D is not clear. After this, B may not enter CS because it too is not clear. This configuration is a deadlock, as no more progress can be made.

[1] If the reader wishes, he may substitute 'program' for 'railroad', 'thread' for 'train', 'call flow' for 'track', and 'critical section' for 'central station'.

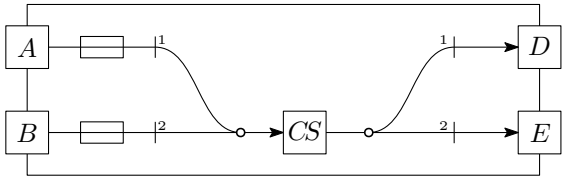

Figure 3.3

Our assumption that there is a train at D is unreasonable, because we have limited our analysis only the railroad depicted. In reality, D is connected to the rest of the railroad network.

We shall assume that A and B are input, and that D and E are output. By input we mean it is unknown to us when trains arrive. By output we mean that the trains eventually leave. We shall analyze our railroad as a component to discover its ideal environment: what are the restriction on the inputs imposed by our component?

Our component is composite, meaning it is defined as a composition of primitive components. Consider the following components:

- A parking spot on a single track. Here a train may wait.
- A tagged merger of two incoming tracks into one outgoing track. Allows only one train to pass. Tags any such train with a label.
- A tagged router of one incoming track into two outgoing tracks. A destination is decided by inspecting the label of a passing train and subsequently removing its label.
- An unknown component, the station CS. We shall assume it behaves as a buffer.

These components are composed as in Figure 3.3: we have two inputs A and B, on the boundary of our component. Following A and B we have two parks. These are combined by a tagged merger, connected to CS, followed by a tagged router. The latter is connected to two outputs D and E on the boundary.

The park was defined in Section 3.1.3. We now consider the definition of the primitive components: a tagged merger and a tagged router. See Figure 3.4.

A tagged merger transports an input element to its output and wraps the input element into a sum type, signaling the source port.

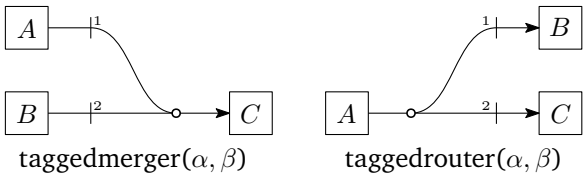

Figure 3.4

Component:	taggedmerger(α, β)
Input ports:	A of α, B of β
Output ports:	C of $\alpha + \beta$
Protocol:	$\forall t. \lfloor A(t) \rfloor = C(t) \wedge B(t) = * \vee$
	$\lceil B(t) \rceil = C(t) \wedge A(t) = *$

A tagged router takes an input element from a sum type and destructs it. The destination is decided based on the constructor.

Component:	taggedrouter(α, β)
Input ports:	C of $\alpha + \beta$
Output ports:	A of α, B of β
Protocol:	$\forall t. C(t) = \lfloor A(t) \rfloor \wedge B(t) = * \vee$
	$C(t) = \lceil B(t) \rceil \wedge A(t) = *$

3.3 Inductive Components

3.3.1 Constructing and Using Data

- Recall coproduct: taggedmerger, taggedrouter
    - Product: busses and splitters
    - Unit: unary signal
    - Zero: never fires
    - Natural Number: generates multiple signals, captures multiple signals

3.3.2 Unbounded Buffers

- List: α^* finite terms of sequences of type α
    - Capture list elements and store in memory

- Restore list and output individual elements
- Unbounded but finite buffer
- Unbounded but finite prophet

3.4 Bibliographical Notes

- buffer is called asynchronous FIFO1 in literature.
- park is called synchronous FIFO1 in literature.

Chapter 4

A Theory of Components

4.1 Language

4.1.1 Definition

We recall the basic definitions from Section 2.3. We shall define a typed first-order predicate logic of two sorts of elements: time and data streams. Data streams are typed according to its data type, which we simply call its *type*. Types are denoted $\alpha, \beta, \ldots$.

We have two kinds of variables. Let V be the countably infinite set of *time variables*. Let P be the countably infinite set of *data stream variables*. Time variables are denoted $x, y, z, \ldots$ and data stream variables are denoted $X, Y, Z, \ldots$ Data stream variables are also called *port variables*.

Definition 1. A *time expression* is as formed by the grammar:

$$i, j ::= x \mid n \mid (i \times j) \mid (i + j)$$

where $x \in V$ is an arbitrary time variable, and n is an arbitrary natural number.

Let S be a countably infinite set of *constant and function symbols*, and let $|s|$ denote the *arity* for each symbol $s \in S$. A constant s has arity $|s| = 0$.

Definition 2. A *data expression* is as formed by the grammar:

$$d, e ::= * \mid c \mid X(i) \mid f(d_1, \ldots, d_n)$$

where $c \in S$ such that $|c| = 0$, and $f \in S$ such that $|f| = n$ equals the number of arguments in the expression $f(d_1, \ldots, d_n)$, and where $X \in P$ is an arbitrary port variable.

Time expressions and data expressions are both *expressions*. In the next sections we consider the denotational semantics of terms. The notion of occurrence of time variables and data port variables in expressions is standard.

Definition 3. A *formula* is as is formed by the grammar:

$$\begin{aligned}\phi, \psi, \chi ::= {} & (i \leq j) \mid (d = e) \mid \top \mid \bot \\ & \neg\phi \mid (\phi \vee \psi) \mid (\phi \wedge \psi) \mid (\phi \to \psi) \\ & (\exists x.\phi) \mid (\forall x.\phi) \mid (\exists X^{\alpha}.\phi) \mid (\forall X^{\alpha}.\phi)\end{aligned}$$

where $\exists x.\phi$ and $\forall x.\phi$ binds variable x in subformula ϕ, and similar for $\exists X^{\alpha}.\phi$ and $\forall X^{\alpha}.\phi$. The type annotation α may be omitted if unambiguous or clear from context. We consider equality of formulas modulo renaming of bound variables.

The notions of *subexpression* and *subformulas*, *occurrence*, *free variables*, and *substitution* are standard. By $V(\phi)$ we denote the set of time variables occurring in ϕ, and by $P(\phi)$ we denote the set of port variables occurring in ϕ. The set $FV(\phi)$, respectively $FP(\phi)$, denotes the set of time variables, resp. port variables, occurring free in ϕ.

We write $\phi[i/x]$ for the *capture-avoiding substitution* of i for freely occurring time variables x, and similarly $\phi[d/X]$ for port variables.

A formula ϕ is called a *V-sentence* if $FV(\phi)$ is empty. A formula ϕ is called a *sentence* if both $FV(\phi)$ and $FP(\phi)$ are empty.

We now consider typing judgments: let $X : \mathbb{N} \to \alpha$ denote that port variable X has type α, and let $x : \mathbb{N}$ denote that time variable x is well-typed. We also consider *typed time expressions* $i : \mathbb{N}$ and *typed data expressions* $d : \alpha$ to denote that time expression i is well-typed, and that data expression d has type α. A *typing context* is a set consisting of typing judgments denoted by Γ. We define the relation $\vdash$ that relates typing contexts to formulas (and to typed data expressions and to typed time expressions), as given by Figure 4.1.

Given a formula ϕ, if there exists a typing context Γ such that $\Gamma \vdash \phi$, then we say that ϕ is well-typed. We assume to work only with

$$\frac{x : \mathbb{N} \in \Gamma}{\Gamma \vdash x : \mathbb{N}} \quad \frac{}{\Gamma \vdash n : \mathbb{N}} \quad \frac{\Gamma \vdash i : \mathbb{N} \quad \Gamma \vdash j : \mathbb{N}}{\Gamma \vdash (i \times j) : \mathbb{N}} \quad \frac{\Gamma \vdash i : \mathbb{N} \quad \Gamma \vdash j : \mathbb{N}}{\Gamma \vdash (i + j) : \mathbb{N}}$$

$$\frac{}{\Gamma \vdash * : \alpha} \quad \frac{c : \alpha}{\Gamma \vdash c : \alpha} \quad \frac{X : \mathbb{N} \to \alpha \in \Gamma \quad \Gamma \vdash i : \mathbb{N}}{\Gamma \vdash X(i) : \alpha}$$

$$\frac{f : \alpha_1 \to \cdots \to \alpha_n \to \alpha \qquad \Gamma \vdash d_1 : \alpha_1 \quad \cdots \quad \Gamma \vdash d_n : \alpha_n}{\Gamma \vdash f(d_1, \ldots, d_n) : \alpha}$$

$$\frac{\Gamma \vdash i : \mathbb{N} \quad \Gamma \vdash j : \mathbb{N}}{\Gamma \vdash (i \leq j)} \quad \frac{\Gamma \vdash d : \alpha \quad \Gamma \vdash e : \alpha}{\Gamma \vdash (d = e)} \quad \frac{}{\Gamma \vdash \top} \quad \frac{}{\Gamma \vdash \bot}$$

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \neg\phi} \quad \frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi} \quad \frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi} \quad \frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \to \psi}$$

$$\frac{\Gamma, x : \mathbb{N} \vdash \phi}{\Gamma \vdash \exists x.\phi} \quad \frac{\Gamma, x : \mathbb{N} \vdash \phi}{\Gamma \vdash \forall x.\phi} \quad \frac{\Gamma, X : \mathbb{N} \to \alpha \vdash \phi}{\Gamma \vdash \exists X^{\alpha}.\phi} \quad \frac{\Gamma, X : \mathbb{N} \to \alpha \vdash \phi}{\Gamma \vdash \forall X^{\alpha}.\phi}$$

Figure 4.1: *Typing rules for formulas.*

well-typed formulas. In particular, $\emptyset \vdash \phi$ holds if and only if well-typed formula ϕ is a sentence.

4.1.2 Composition

We now consider compositions and the well-formedness and well-typedness of compositions.

Let $\alpha, \beta, \ldots$ be elements of a countably infinite set of types. Let V be a countably infinite set of *instance variables*, denoted by $x, y, z, \ldots$ Let P be a countably infinite set of *port variables*, denoted by $X, Y, Z, \ldots$ Stream variables are also called port variables.

Definition 4. A *reference* is as given by the following grammar:

$$p, q, r, s ::= x.X^{\alpha} \mid X^{\alpha}$$

where x is an instance variable, X is a stream variable, and α a type annotation. The type annotation α may be omitted if unambiguous or clear from context.

References are either *qualified* or *unqualified*. Qualified and unqualified references are used in compositions and their interface, and only unqualified references are used in the interface of composite components. We refer to the ports at the external boundary of a component instance by the qualified reference of an instance variable and an external port variable. We refer to the ports at the internal boundary of a component as an unqualified reference.

References have a natural order that is derived from the natural order of instance variables, port variables and types.

Definition 5. A *composition* is as given by the following grammar:

$$c, d, e ::= x \mid (c \parallel d) \mid (c)^p_q$$

such that the following laws hold:

$$(c \parallel c) = c$$
$$(c \parallel d) = (d \parallel c)$$
$$(c \parallel (d \parallel e)) = ((c \parallel d) \parallel e)$$
$$((c)^p_q)^p_q = (c)^p_q$$
$$((c)^p_q)^r_s = ((c)^r_s)^p_q$$
$$((c)^p_q \parallel d) = ((c \parallel d)))^p_q$$

That is, parallel composition is idempotent, commutative, and associative; identification is idempotent, commutative, and distributes over parallel composition.

A *composition* is an instance variable, a *parallel composition* of two compositions, or an *identification* of two references and a composition. The top reference of an identification is called the *source*, and the bottom reference is called the *sink*. We have the notion of *occurrence* of instance variables and references.

A composition can be simplified into a normal form. First, we push out all identifications by distributivity to obtain a composition in which all parallel compositions are deep, and identifications are on the surface. Next, we associate all nested parallel compositions to the right to form a list of instances. Next, we sort the instances according to the natural order of instance variables, removing duplicates. The surface identifications also form a list of pairs of references. We sort

this list according to the lexicographic orders of pairs of references, removing duplicates. The result has the shape $((((x \parallel (\ldots \parallel z)))^{p}_{q})\ldots)^{r}_{s}$ such that instances are ordered, and references are lexicographically ordered.

We shall work with compositions that are in normal form. We call the *inner part* of a composition to be the parallel composition of instance variables $(x \parallel (\ldots \parallel z))$, and the outer part consists of all surrounding identifications.

We want to prevent certain compositions: forbidding the identification of ports of unknown instances, forbidding identifying references more than once, and forbidding references to occur both as source and sink. This ensures that every reference resolves to an instance which occurs in the composition, and that identification is in some sense affine: a port is never referenced more than once.

Definition 6. A composition is *well-formed* if:

- every occurring qualified reference's instance occurs,
- every reference occurs at most once.

The last condition implies that a reference occurs exclusively as a source or a sink, and that a reference is not identified with itself. A well-formed composition is easily recognized by looking at its normal form. The first condition is checked by verifying that each qualified reference's instance occurs in the sorted list of instances deeper in the composition. The last condition is checked by verifying that a reference never occurs twice in a row in either normal form.

An example of a well-formed compositions are: $(x)^{x.X}_{Y}$ denotes the composition of instance variable x of which its external port $x.X$ is identified to the internal port Y at the boundary of our composition.

Here are some negative examples. $(y)^{x.Y}_{X}$ is not well-formed because x does not occur. $((y)^{y.Y}_{X})^{y.Y}_{Z}$ is not well-formed because $y.Y$ occurs twice. $(y)^{X}_{X}$ and $((y)^{X}_{Y})^{Z}_{X}$ are not well-formed because X is both a source and sink.

We still want to prevent more compositions: compositions must only identify ports of the same type, and we want to keep track of the interface of a component to resolve references against. Towards this, we first introduce interfaces.

$$\frac{}{\Gamma, x :: U \vdash x : x.U} \qquad \frac{\Gamma \vdash c : \langle \Delta \mid \Theta \rangle}{\Gamma \vdash (c)^{X}_{Y} : \langle X^{\alpha}, \Delta \mid \Theta, Y^{\alpha} \rangle} \qquad \frac{\Gamma \vdash c : \langle x.X^{\alpha}, \Delta \mid \Theta, y.Y^{\alpha} \rangle}{\Gamma \vdash (c)^{x.X}_{y.Y} : \langle \Delta \mid \Theta \rangle}$$

$$\frac{\Gamma \vdash c : U \quad \Gamma \vdash d : V}{\Gamma \vdash (c \parallel d) : U \cup V} \qquad \frac{\Gamma \vdash c : \langle \Delta \mid \Theta, y.Y^{\alpha} \rangle}{\Gamma \vdash (c)^{X}_{y.Y} : \langle X^{\alpha}, \Delta \mid \Theta \rangle} \qquad \frac{\Gamma \vdash c : \langle x.X^{\alpha}, \Delta \mid \Theta \rangle}{\Gamma \vdash (c)^{x.X}_{Y} : \langle \Delta \mid \Theta, Y^{\alpha} \rangle}$$

Figure 4.2: *Typing rules for compositions.*

Definition 7. An *interface* is a pair of two disjoint sets of references, denoted $\langle p_1, \ldots, p_n \mid q_1, \ldots, q_k \rangle$. An *unqualified interface* is an interface consisting only of unqualified references. A *qualified interface* is an interface consisting only of qualified references.

The empty interface is denoted as $\langle\rangle$. Each component definition has an associated unqualified interface (cf. Section 4.1.3). We define the following operations on interfaces.

Given an unqualified interface $U = \langle X_1, \ldots, X_n \mid Z_1, \ldots, Z_k \rangle$, we may *qualify* it by an instance x, denoted $x.U$, to mean the qualified interface $\langle x.X_1, \ldots, x.X_n \mid x.Z_1, \ldots, x.Z_k \rangle$.

By $U^{\perp}$ we denote the dual interface: $U^{\perp} = \langle Z_1, \ldots, Z_k \mid X_1, \ldots, X_n \rangle$. The left-hand side of an interface are called *input* port variables and the right-hand side are called *output* port variables. The dual of an interface swaps input and output.

We may also lift union to interfaces. Let $\Gamma, \Delta, \Theta, \Pi$ be sets of references. Given two interfaces $U = \langle \Gamma \mid \Theta \rangle$ and $V = \langle \Delta \mid \Pi \rangle$, then by $U \cup V$ we mean pairwise union of the two interfaces to form $\langle \Gamma \cup \Delta \mid \Theta \cup \Pi \rangle$.

Now we introduce typed compositions. Consider the typing judgment $x :: U$ of an instance variable x and unqualified interface U. A *typed compositions* $c : U$ is a well-formed composition c and interface U. Let a typing context be a set of typing judgments, as denoted by Γ. We define the relation $\vdash$ between typing contexts and typed compositions, as given by Figure 4.2.

Here, we take $U \cup V$ to mean the pairwise union of two interfaces. As a side-condition to these rules, we assume that for p, Δ it holds that $p \notin \Delta$, and for Θ, q it holds that $q \notin \Theta$. We have not written type

annotations in compositions for brevity: they are the same as the annotation given in the interface.

Given a well-formed composition c, if there exists a typing context Γ and interface U such that $\Gamma \vdash c : U$, then we say that c is a *well-typed composition*. The intention of a well-typed composition is to ensure that references are used linearly and that identification of two references are of the same type. We shall assume that all compositions we work with in the sequel are well-typed (and thus well-formed), unless mentioned otherwise.

4.1.3 Component

Next, we consider the construction of components. Our intention is that a component is either a primitive component, or a composite component consisting of primitive components. To do so, we consider the construction of components as either a well-typed composition, or a binding of an instance variable to a primitive component. We assume a given set of primitive components, denoted P, Q, R, S.

Definition 8. A *component* is as given by the following grammar:

$$C, D, E ::= c \mid \textbf{new}\ R \mid (\textbf{let}\ x = C\ \textbf{in}\ D)$$

where c is a well-typed composition, **new** R is a primitive component R, and **let** binds the instance variable x in D. We shall consider composite components equal modulo renaming of bound instance variables.

Certain components contain redundancies. We simplify components according to the following three rules. The first rule removes bindings for non-occurring instances:

$$\textbf{let}\ x = C\ \textbf{in}\ D \to D$$

with the side-conditions that x does not occur in D, hence x is unused and the binding can be eliminated. The second rule permutes bindings:

$$\textbf{let}\ x = (\textbf{let}\ y = C\ \textbf{in}\ D)\ \textbf{in}\ E \to \textbf{let}\ y = C\ \textbf{in}\ (\textbf{let}\ x = D\ \textbf{in}\ E)$$

with the side-condition that y does not occur in E, or if it does it is suitably renamed: the nested **let** binding is pulled back to the outer level. The third rule substitutes compositions:

$$\textbf{let } x = c \textbf{ in } C \to C[x.c/x][x]$$

where $C[c/x]$ denotes substitution, $x.c$ denotes the qualification of composition c by x, and $C[x]$ denotes the linking through of references. A composition c is qualified to x by substituting every occurring unqualified reference X by $x.X$. A substitution $C[c/x]$ denotes standard substitution of each occurrence of an instance variable x by the composition c. A non-well-formed composition c is linked through qualified reference x, by finding identifications with sink p and source $x.X$ and sink $x.X$ and source q for each port X, removing these two identifications, and identifying p and q in the resulting composition if $p \neq q$. We denote linking through of a non-well-formed composition c as $c[x]$. Linking through is lifted to components $C[x]$ by replacing each occurring composition c by $c[x]$.

Qualification and linking through preserves well-formed composition: given a well-formed composition c bound to x, and given a well-formed composition d, then the composition $d[x.c/x][x]$ is well-formed. Clearly, substituting an instance variable by a qualified composition makes a non-well-formed composition, e.g. qualifying $(y)^A_B$ to x results in $(y)^{x.A}_{x.B}$ and its substitution for x in $(x)^{x.B}_{x.A}$ results in non-well-formed composition $((y)^{x.B}_{x.A})^{x.A}_{x.B}$. By linking through, we obtain $(y)^{x.B}_{x.B}$ or $(y)^{x.A}_{x.A}$, both of which link through to y.

We given an example of simplification and substitution. Take for example **let** $x = ((y)^{y.X}_A)^B_{y.Y}$ **in** $((x)^{x.A}_Z)^W_{x.B}$. We assume y is a component with interface $\langle X \mid Y \rangle$. What follows is that within the composition of x, we link $y.X$ and $y.Y$ to the unqualified references A and B, where A is a sink and B is a source. We qualify every unqualified reference by its instance variable, substitute the composition for each occurrence of the bound instance variable, and then resolve the identifications by linking them through. The first step results in the qualification of references: $((y)^{y.X}_A)^B_{y.Y}$ becomes $((y)^{y.X}_{x.A})^{x.B}_{y.Y}$. The second step results in $((((y)^{y.X}_{x.A})^{x.B}_{y.Y})^{x.A}_Z)^W_{x.B}$ where x is replaced by $((y)^{y.X}_{x.A})^{x.B}_{y.Y}$. The last step removes identifications by linking through: we link $y.X$ to Z via $x.A$ and remove $x.A$ to obtain $(((y)^{x.B}_{y.Y})^{y.X}_Z)^W_{x.B}$, and we link $y.Y$ to W

$$\frac{\Gamma \vdash c : U \quad U \text{ is unqualified}}{\Gamma \vdash c :: U} \qquad \frac{}{\Gamma \vdash \textbf{new}\ R :: U} \qquad \frac{\Gamma, x :: C^{\perp} \vdash D :: U}{\Gamma \vdash \textbf{let}\ x = C\ \textbf{in}\ D :: U}$$

Figure 4.3: *Typing rules for composite components.*

via $x.B$ and remove $x.B$ to obtain $((y)^{y.X}_{Z})^{W}_{y.Y}$.

After no more rules of simplification applies, we have obtained components in normal form. These components are either of the form of being a primitive component, **new** R, or a composite component, **let** $x =$ **new** R **in** $(\ldots(\textbf{let}\ z =$ **new** S **in** $c)\ldots)$, with zero or more bindings. The latter is also written **let** $x =$ **new** $R, \ldots, z =$ **new** S **in** c.

We now consider typing judgments and typing contexts of components. Let U be an unqualified interface, and $C :: U$ a typing judgment, and let the typing context Γ denote a set of typing judgments. We define a relation between typing contexts and a single typing judgment of a composite component, denoted $\Gamma \vdash C :: U$, as given by Figure 4.3.

All the rules are remarkable:

- in the first rule, recall that the difference between $C :: U$ and $c : U$ is that the former judges only unqualified interfaces, whereas the latter judges arbitrary interfaces. Hence, for a composition to become a composite component, it is required to have an unqualified interface. This is possible by identifying all qualified references.
- The second rule is a family of rules, one rule for each primitive component R with unqualified interface U. We shall admit this rule for each primitive component R that has U as its unqualified interface.
- The third rule shows that x can be bound in D. This dualizes the interface of C, bound to x in the context of component D: what we used to consider an output for C, now is an input to D; and what we used to consider an input to C, now is an output for D.

We call components C for which $\emptyset \vdash C$ is derivable *closed components*.

4.2 Interpretation

4.2.1 Denotational Semantics

In this section, we consider the denotation of time expressions, data expressions, formulas, compositions and components.

By $\mathbb{N}$ we mean the set of natural numbers. By $\mathbb{S}$ we denote the domain of all elements. An element in $\mathbb{S}$ is an element $a : \alpha$ such that $a \in \alpha$ and α is a (data) type. Elements in $\mathbb{S}$ are implicitly coerced to the element they represent. Recall from Section 2.1, that types are recursively enumerable. Hence, elements in $\mathbb{S}$ of the same type α have decidable equality, which we shall denote by $=_\alpha$.

A time expression denotes a natural number, a data expression denotes an element of a data type. Let $\gamma : V \to \mathbb{N}$ denote an assignment of time variables, and $\delta : P \to (\mathbb{N} \to \mathbb{S})$ denote an assignment of port variables to data streams. We write $\gamma[x := n]$ to denote the assignment in which x is mapped to n and all variables not equal to x are mapped by γ, and similar for $\delta[X := \tau]$ for some data stream $\tau : \mathbb{N} \to \mathbb{S}$. We also write $\tau : \mathbb{N} \to \alpha$ to denote the data stream which maps only to elements of type α.

Let $\mathcal{M} = (\gamma, \delta)$ denote a *model*. Interpretation of time expressions and data expressions in a model $\mathcal{M}$ is denoted by the (overloaded) operator $[\![\cdot]\!]^{\mathcal{M}}$. We define the interpretation $[\![i]\!]^{\mathcal{M}}$ of a time expression i in model $\mathcal{M}$ recursively:

$$
\begin{aligned}
[\![x]\!]^{\mathcal{M}} &= \gamma(x) \\
[\![n]\!]^{\mathcal{M}} &= n \\
[\![i + j]\!]^{\mathcal{M}} &= [\![i]\!]^{\mathcal{M}} + [\![j]\!]^{\mathcal{M}} \\
[\![i \times j]\!]^{\mathcal{M}} &= [\![i]\!]^{\mathcal{M}} \times [\![j]\!]^{\mathcal{M}}
\end{aligned}
$$

Similarly, we define the interpretation $[\![d]\!]^{\mathcal{M}}$ of a data expression d in model $\mathcal{M}$, where d is well-typed by type α, recursively:

$$
\begin{aligned}
[\![*]\!]^{\mathcal{M}} &= * : \alpha \\
[\![c]\!]^{\mathcal{M}} &= c : \alpha \\
[\![X(i)]\!]^{\mathcal{M}} &= \delta(X)([\![i]\!]^{\mathcal{M}}) : \alpha \\
[\![f(d_1, \ldots, d_n)]\!]^{\mathcal{M}} &= f([\![d_1]\!]^{\mathcal{M}}, \ldots, [\![d_n]\!]^{\mathcal{M}}) : \alpha
\end{aligned}
$$

We define satisfaction of a formula ϕ in model $\mathcal{M} = (\gamma, \delta)$, denoted $\mathcal{M} \vDash \phi$, recursively:

$$\mathcal{M} \vDash (i \leq j) \text{ iff } [\![i]\!]^{\mathcal{M}} \leq [\![j]\!]^{\mathcal{M}}$$
$$\mathcal{M} \vDash (d = e) \text{ iff } [\![d]\!]^{\mathcal{M}} =_{\alpha} [\![e]\!]^{\mathcal{M}} \text{ where } d, e \text{ have type } \alpha$$
$$\mathcal{M} \vDash \top \qquad \mathcal{M} \nvDash \bot$$
$$\mathcal{M} \vDash \neg\phi \text{ iff } \mathcal{M} \nvDash \phi$$
$$\mathcal{M} \vDash \phi \vee \psi \text{ iff } \mathcal{M} \vDash \phi \text{ or } \mathcal{M} \vDash \psi$$
$$\mathcal{M} \vDash \phi \wedge \psi \text{ iff } \mathcal{M} \vDash \phi \text{ and } \mathcal{M} \vDash \psi$$
$$\mathcal{M} \vDash \phi \to \psi \text{ iff } \mathcal{M} \vDash \phi \text{ implies } \mathcal{M} \vDash \psi$$
$$(\gamma, \delta) \vDash \exists x.\phi \text{ iff } (\gamma[x := n], \delta) \vDash \phi \text{ for some } n \in \mathbb{N}$$
$$(\gamma, \delta) \vDash \forall x.\phi \text{ iff } (\gamma[x := n], \delta) \vDash \phi \text{ for all } n \in \mathbb{N}$$
$$(\gamma, \delta) \vDash \exists X^{\alpha}.\phi \text{ iff } (\gamma, \delta[X := \tau]) \vDash \phi \text{ for some } \tau \in (\mathbb{N} \to \alpha)$$
$$(\gamma, \delta) \vDash \forall X^{\alpha}.\phi \text{ iff } (\gamma, \delta[X := \tau]) \vDash \phi \text{ for all } \tau \in (\mathbb{N} \to \alpha)$$

Next, we consider formal protocols. A formal protocol is a set of streams of observations. We assume each port variable X has an associated type α. An observation assigns to a port a finite number of elements. We call the assigned element (which is either null or a data element) a value. Intuitively, an observation represents a consistent 'snapshot' of the ports of a component made by an independent observer.

Definition 9. An *observation* is a partial function with finite support $o : (P \to \mathbb{S})$, such that $o(X) \in \alpha$ where α is the type of X for a finite number of port variables X, and $o(Y) = *$ for all other port variables Y. A silent observation $o : (P \to \mathbb{S})$ maps each port X to $*$, i.e. $o(X) = *$ for all X. A *formal protocol* is a set of streams of observations $\sigma : \mathbb{N} \to (P \to \mathbb{S})$.

Compare assignments $\delta : P \to (\mathbb{N} \to \mathbb{S})$ to the elements of protocols $\sigma : \mathbb{N} \to (P \to \mathbb{S})$. This suggests that there is a close connection between protocols and formulas. In particular, a V-sentence defines a formal protocol.

Definition 10. Given a V-sentence ϕ. The formal protocol $\mathcal{L}(\phi)$ defined by ϕ is $\mathcal{L}(\phi) = \{\sigma \mid (\gamma, \delta) \vDash \phi\}$ for arbitrary γ.

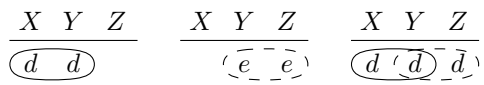

Figure 4.1: *Intersection of frame conditions.*

This definition makes sense if we have that $(\gamma_1, \delta) \vDash \phi$ if and only if $(\gamma_2, \delta) \vDash \phi$, for a V-sentence ϕ and arbitrary γ_1, γ_2. This is verified by the argument, that if $(\gamma_1, \delta) \vDash \phi$ holds but $(\gamma_2, \delta) \vDash \phi$ not, then γ_1 and γ_2 must have a different assignment for a time variable that occurs free in ϕ. But by definition of V-sentence, ϕ has no free time variables.

We reason about formal protocols by considering it as a set of tables of observations.

We have the standard correspondence between sets and propositions. Let ϕ, ψ be two V-sentences. The formula $\neg\phi$ is the complement of the protocol of $\mathcal{L}(\phi)$, i.e. every element not in $\mathcal{L}(\phi)$. The formula $\phi \wedge \psi$ is the intersection $\mathcal{L}(\phi) \cap \mathcal{L}(\psi)$ of the two protocols $\mathcal{L}(\phi)$ and $\mathcal{L}(\psi)$. And similar for the other propositional connectives.

We give some examples of equalities and how to understand them as *frame conditions* on sets of tables of observations. Take, for example, the replicator or consensus component from Section 3.1.4. The frame condition is $X(0) = Y(0)$ and $Y(0) = Z(0)$, which applies to only the first row. These two frame conditions are overlapped, thus restricting the number of allowed observations in the first element, as in Figure 4.1. The first constraint allows any value to appear at Z in the first row. The second constraint allows any value to appear at X at the first row. But the constraints combined only allow elements that are equal to all X, Y, and Z. These constraints, however, do not restrict any other value at other ports.

The intended behavior of a replicator, however, persists over time. We thus consider the previous frame condition to be applied to every time. Hence the frame condition is $X(t) = Y(t)$ and we universally quantify over t, to obtain $\forall t.X(t) = Y(t)$. Similarly, the second has as frame condition $Y(t) = Z(t)$, and universally quantified it becomes $\forall t.Y(t) = Z(t)$. The intuition of universal quantification is to *slide* the frame condition over all rows. Hence, the first V-sentence only allows observations for which at every time X and Y have the same value; it still does not restrict Z in any way. Similarly for the second V-sentence. The combination $\forall t.X(t) = Y(t) \wedge Y(t) = Z(t)$ results in

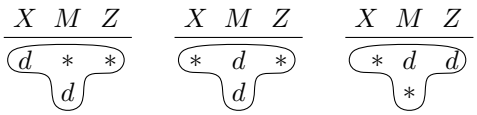

Figure 4.2: *Three frame conditions that make up a buffer.*

t	X	M	Z
0	$*$	$*$	$*$
1	d	$*$	$*$
2	$*$	d	$*$
3	$*$	d	$*$
4	$*$	$*$	d
5		$*$	

Figure 4.3: *Example sequence of observations, with constraints of a buffer.*

the constraint that all ports, at all times, must have the same value.

Another interesting frame condition is that of the buffer and the prophet. The basic condition ranges over two rows, as in Figure 4.2. The first frame condition specifies that: if the memory is empty, and port X has value d and Z has no value, then it is stored in the memory in the next place. The second frame condition specifies that memory is copied whenever there is no input or output. The third frame condition specifies that the contents of memory is the same as at port Z, and that memory is cleared in the next row. The buffer definition takes the union of these three frame conditions. We demonstrate how these conditions apply to an arbitrary sequence of observations in Figure 4.3.

From the previous example, we observe a pattern that we can directly describe the relation between the port X and Z, using a variable-sized frame condition. The frame conditions are given in Figure 4.4. These frame conditions apply for each row and are specified by:

$$
\begin{aligned}
&\forall t.Z(t) = * \wedge X(t) = * \vee \\
&\quad (Z(t) = * \wedge \exists j.t < j \wedge X(j) = * \wedge Z(j) = X(t) \wedge \\
&\qquad\qquad \forall i.t < i \wedge i < j \rightarrow X(i) = * \wedge Z(i) = *) \vee \\
&\quad (X(t) = * \wedge \exists j.j < t \wedge X(j) = Z(t) \wedge Z(j) = * \wedge \\
&\qquad\qquad \forall i.j < i \wedge i < t \rightarrow X(i) = * \wedge Z(i) = *)
\end{aligned}
$$

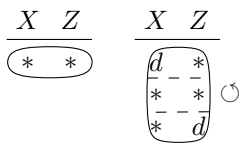

Figure 4.4: *Alternative frame conditions that make up a buffer.*

It means that for each row (at t), whenever X has value d, there must exists some future row (at j) such that Z has the same value d. The values at both ports that are intermediate between these two rows are required to be $*$. It also means that for each row (at t), whenever Z has value d, there must exists a previous row with the same value, and all intermediate rows are required to be $*$. Note that our frame condition is still sliding here: the condition that $Z(t) = * \wedge X(t) = *$ is applied for all rows that are intermediate between the element accepted by X and then returned by Z.

We now move over to primitive components and composite components. Components refine protocols by introducing the notion of input and output ports. A component is interpreted as a particular formula, consisting of two V-sentences that define the component protocol and the environment protocol. One may, intuitively, regard the environment protocol as an assumption: the component is defined only in those environments that conform to the environment protocol.

Definition 11. A *component interpretation* consists of the following data:

- a *component formula* ϕ which is a V-sentence,
- a partition of $FP(\phi)$ into *input port*, *memory*, and *output port*.

where $I(\phi)$ are input port variables, $M(\phi)$ are memory variables, and $O(\phi)$ are output port variables.

Recall that each component has an interface: we can infer it from its interpretation as $\langle X_1, \ldots, X_n \mid Z_1, \ldots, Z_k \rangle$ where $I(\phi) = \{X_1, \ldots, X_n\}$ and $O(\phi) = \{Z_1, \ldots, Z_k\}$. Again, we leave type annotation implicit.

We considered previously two forms of component: a primitive component and a composite component. In the first case, we require component interpretations for each primitive component. In the latter case, we may define a component interpretation of a composite component recursively by the structure of a composite component. The

interpretation for each primitive component is *a priori* assumed. For example, in Chapter 3 we have seen many such primitive components.

A composition denotes a finite set of instance variables and a finite relation of references. The set of instance variables of a composition precisely occur in that composition, and similar for references. More precisely, x is represented by the set $\{x\}$ and the empty relation, $(c \parallel d)$ is represented by the union of the sets and relations of the representations of c and d, and $(c)^p_q$ is represented by adding (p, q) to the relation of the representation of c. For example, $(((x)^{y.Z}_{x.Y} \parallel y)^{y.X}_{x.Z}$ is represented by $\{x, y\}$ and $\{(y.Z, x.Y), (y.X, x.Z)\}$, and so is its normal form $((x \parallel y)^{y.X}_{x.Z}))^{y.Z}_{x.Y}$.

A well-formed composition is verified by ensuring that every qualified reference has an instance that is contained in the set of instance variables, and that the relation is a partial function (each element is related to at most one other), injective (each related element is mapped to by a unique element), irreflexive (no element is related to itself), and acyclic (no element is transitively related to itself).

The component interpretation of a composite component is the following. For each primitive component we have a component formula and an interface. We work towards constructing a conjunction of all component formulas. To do so we rename each component formula to ensure each port name is unique: we do so, by using the qualified reference as variable name. Then we take a conjunction of all renamed component formulas. For each identification of references p and q we add an equation $\forall t.p(t) = q(t)$ to the conjunction.

The result is a component interpretation of the composite component. All qualified references are memory variables, unqualified references that occur as source are input variables, unqualified references that occur as sink are output variables.

4.2.2 Constraint Solving

We have a subclass of formulas called *simple constraints*:

$$\begin{aligned} d, e &::= * \mid c \mid f(d_1, \ldots, d_n) \mid X(0) \\ \phi, \psi, \chi &::= (d = e) \mid \top \mid \bot \\ &\quad \neg\phi \mid (\phi \vee \psi) \mid (\phi \wedge \psi) \mid (\phi \to \psi) \end{aligned}$$

- Constraints are a subset of formulas
- Algorithm for constraint solving
- Relates to simplification of free coproduct

4.2.3 Operational Semantics

- Concurrent constraint automata
  - Constraints as transitions
  - Compare to Petri nets and state machines
  - Difference between ‘committed choice’ and ‘parallel choice’

4.3 Logical Analysis

- Encoding in calculus of constructions
  - Provability of a sentence
  - Different ‘proof’ stands for different behavior
  - Problem: given a set of ‘primitives‘ as axioms, and rules of inference are ‘composition‘: then what ‘components‘/theorems are derivable?

4.3.1 Quantification

- Realizability of a provable sentence
  - Universal as function of stream to proof
  - Existential as pair of stream, proof
  - Argue why this encoding makes sense (stream transducer)

4.3.2 Polarization

- From a V-sentence to a sentence
  - Input: universal, memory: existential, environment: protocol, output: existential, protocol
  - Problem: find ideal environment
  - Forcing
  - Duality

4.3.3 Certification

- Definition of certified component

4.4 Bibliographical Notes

Chapter 5

Certifying Components

5.1 Showing Equivalences

5.1.1 Definition-Definition

- Showing alternative definition of (1-place) buffer using inequalities and no memory
    - Establish equivalence with original definition in 3.1.3

5.1.2 Definition-Composition

- Showing equivalence of variable (definition in 3.1.3) and alternative construction (composition in)

5.1.3 Composition-Composition

- Proving identities of n-ary nodes
    - Associativity
    - Commutativity
    - Distributivity (merger/consensus, router/replicator)
    - Identity laws

5.2 Establishing Properties

5.2.1 Independent Progress

- Formalization
    - Example: show that buffer has independent progress
    - Counter-example: show that components can be defined that force

5.2.2 Linearity

- Formalization
    - Example: show that buffer and unbounded buffer are linear
    - Counter-example: show that variable and replicator are both not linear

5.2.3 Causality

- Formalization
    - Example: show that buffer and unbounded buffer is causal
    - Counter-example: show that prophet and unbounded prophet is not causal

5.3 Case Study

- Encoding of a Turing machine

5.4 Bibliographical Notes

Chapter 6

Conclusion

Bibliography

[1] Samson Abramsky. Two puzzles about computation. *Unpublished paper*, 2014.

[2] Farhad Arbab. Abstract behavior types: a foundation model for components and their composition. *Science of Computer Programming*, 55(1):3 – 52, 2005. Formal Methods for Components and Objects: Pragmatic aspects and applications.

[3] Farhad Arbab. Proper protocol. In *Theory and Practice of Formal Methods*, pages 65–87. Springer, 2016.

[4] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual*. PhD thesis, INRIA, 1997.

[5] Henning Basold, Helle Hvid Hansen, Jean-Éric Pin, and Jan Rutten. Newton series, coinductively. In *International Colloquium on Theoretical Aspects of Computing*, pages 91–109. Springer, 2015.

[6] Twan Basten. Branching bisimilarity is an equivalence indeed! *Information Processing Letters*, 58(3):141–147, 1996.

[7] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[8] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover

(system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[9] David De Frutos Escrig, Jeroen J. A. Keiren, and Tim A. C. Willemse. Games for bisimulations and abstraction. *Logical Methods in Computer Science*, 13(4:15), 2017.

[10] Wan Fokkink. *Introduction to process algebra*. Springer Science & Business Media, 2013.

[11] Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):427–443, 1997.

[12] Peter Linz. *An introduction to formal languages and automata (fifth edition)*. Jones & Bartlett Learning, 2011.

[13] Rob Nederpelt and Herman Geuvers. *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, 2014.

[14] Jan Rutten. On streams and coinduction. 2002.

[15] Jan Rutten. *The method of coalgebra: exercises in coinduction*. Unpublished manuscript, 2019.

[16] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[17] Johan van Benthem. *Modal Logic for Open Minds*. CSLI Publications, Stanford University, 2010.

[18] Rob J. Van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

Acknowledgments

About the Author

Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Protocol Certification

Author:	Hans-Dieter A. Hiep	(student number)
1st supervisor:	Jasmin C. Blanchette	(VU)
cosupervisor:	Farhad Arbab	(CWI)
2nd reader:	Femke van Raamsdonk	(VU)

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 14, 2018