Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

# A Reo Semantics for Reasoning about Speculative Execution


| | | |
|---|---|---|
| Author: | Hans-Dieter A. Hiep | (2526195) |
| 1st supervisor: | dr. Jasmin C. Blanchette | (VU) |
| cosupervisor: | prof. dr. Farhad Arbab | (CWI) |
| 2nd reader: | dr. Femke van Raamsdonk | (VU) |


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

October, 2018

**Abstract**

Speculative execution is a technique used in popular processors designed in the past decade, e.g. by Intel and AMD. Take, for example, this imperative program:

```
int outcome = slowOperation();
if (outcome < 0) doThis();
else doThat();
```

Before the `slowOperation()` is finished executing, one may speculate on its outcome. There is a *branch prediction function* that chooses a value, which we call a *speculation*, used to evaluate the **`if`**-statement concurrently with the `slowOperation()`. The outcome can be negative or non-negative, leading to the speculative execution of `doThis()` or `doThat()`, respectively. Once the `slowOperation()` returns, its actual outcome is known. If the speculation is incorrect when compared with the actual outcome, then we need to undo any of the computational steps performed based on such a *false speculation*. This ensures that no unwanted steps are visible outside the processor. Otherwise, if the speculation is correct, then we have gained throughput by already performing steps ahead of time based on a *true speculation*.

Reo is a language for compositional construction of protocols for the coordination of concurrent and distributed systems. As far as we know, Reo has never before been used to study speculative executions. We design a typed language of compositions and components inspired by Reo. We introduce components not modeled before in Reo: PROPHET and PULL. PROPHET is a component that generate speculations. PULL is a component that enforces progress and forbids deadlocks. The behavior of components are constrained using formulas in a first-order logic with data types and data streams. This logic is expressive enough to formulate properties such as *deadlock-freedom* and *livelock-freedom*, and other properties important for understanding speculative executions in concurrent and distributed systems: *delay insentivity, independence*, *(a)synchronicity*, *termination*, *instantaneousness*, *linearity* and *causality*.

# Contents

# Acknowledgments

Many thanks to Farhad Arbab for inspiring discussions, suggesting interesting research topics, and reading thoroughly through this thesis while providing lots of comments on this document. Thank you, Jasmin Blanchette, for the many discussions how to write effectively, for providing useful feedback, and giving the right definition of the logical formalism.

Thank you a lot, Benjamin Lion, for sharing an office, always willing to listen to ideas, and having many deep discussions on concurrency, Reo, Kleene, Russels, and Spinoza. Thanks for the oportunities for discussion and insightful comments given by Kasper Dokter and Sung-Shik Jongmans.

Other people that deserve thanks: Femke van Raamsdonk (for being the second reader of this document), Jacco van Splunter & Roy Overbeek (for discussion), and Wan Fokkink (for his inspiring teaching).

Special thanks go to the people of the Centrum voor Wiskunde & Informatica (CWI) for allowing work on this thesis during an internship period of fourty-five weeks. In particular, thank you, Steven Pemberton, for giving a good reason to come to CWI, and Frank de Boer for approving the internship. Thanks to Jana Wagemaker, Keyvan Azadbakht, Vlad Serbanescu, Jan Rutten, and others in the Formal Methods group for being nice collegues.

To those other people not listed here, but who still provided useful support: thank you!

*This page is intentionally left blank.*

# Chapter 1

# Introduction

A recent interest in the security of microcode architectures of mainstream processors revealed issues (popularly known as Meltdown, Spectre, and Foreshadow; see Kocher et al. [54]). These architectural issues demonstrate side-channel attacks caused by a combination of branch prediction, cache hierarchy, simultaneous multi-threading and speculative execution. In particular, it turns out that when computation is not reversible under certain conditions, this leads to a systematic leakage of privileged information. Mitigation of these issues has serious impact for large-scale computing service providers: under certain workloads perceived performance is reduced by 50% [4].

Speculative execution is a well known technique that allows for the optimization of concurrent systems. Over the course of the critical path of an execution, the number of idle resources varies. One may exploit idle resources by speculative execution that potentially shortens the critical path length to increase throughput. However, speculative execution may also negatively affect throughput if a false speculation must be reverted.

The essence of reversible computing is that every operation can be reverted. This has beneficial properties in itself: ideally, a reversible computation does not dissipate power and thus is highly energy efficient [77]. Some claim that adoption of reversible computing is necessary [40]: the rate of performance improvements of general-purpose computing, as seen in the last decades, comes at greater energy cost than before. Reversible computing allows for significantly less energy usage without negatively affecting performance, thus allowing for performance improvements to continue.

In this thesis, we aim to gain an intuitive understanding of concurrency, speculative execution, and reversible computing. Our vision is to establish logical foundations for concurrent and distributed systems. We employ a structured approach by modeling interactive computing systems as coordination protocols using Reo. Reo is a coordination language for compositional construction of interaction protocols [9, 8]. Coordination is the study of dynamic topologies of interaction between computing systems [5].

On the practical side, Reo manifests as a high-level declerative programming language for constructing concurrent and distributed systems. For example, recently developed compilers produce concurrency glue code which links together single-threaded code. Programmers write single-threaded code in a supported programming language, and design interaction patterns in Reo: the compiler

fills in the gap. This is beneficial to programmers who no longer need to have in-depth technical knowledge of concurrency, likely leading to an increase in productivity and decrease in the number of concurrency-related software bugs.

There are two benefits of raising the level of abstraction that programmers use to define interactions. First, abstraction alleviates programmers from directly working with low-level concurrency primitives (e.g. semaphores, locks, send/receive) and debugging concurrency errors (e.g. race conditions, deadlocks). Instead, programmers specify coordination protocols decleratively, to define the permissible interactions among external single-threaded programs, thus separating their concerns for computation and concurrency. Second, experiments show that concurrency code generated by Reo compilers has run-time performance similar to hand-crafted concurrency code [49]. This is achieved by performing program optimization directly on the higher-level coordination protocols, as opposed to the lower-level of concurrency primitives.

Over the years Reo has developed a rich theory, as demonstrated by over thirty formal semantics [50]. These semantics can be roughly divided into three groups: co-algebraic models (e.g. streams), operational models (e.g. automata), and others. We introduce yet another semantics of Reo, and use it to explore fundamental properties of concurrent and distributed systems. Properties related to speculative execution are more easily expressed in our semantics than in other Reo semantics.

Our presentation of Reo is novel in a way: we focus on the detection of inconsistencies. An inconsistency is a situation in which no valid behavior is specified. Such inconsistencies are effectively resolved by tracing back to a nearest branch point from which execution can be resumed safely. We shed light on dualities present in our language, by the definition of so-called BUFFERS and PROPHETS, that correspond to history variables and future variables [1]. Our language is a dialect of Reo. We define a calculus of graphical notations for components. We have designed a type checker that is closely related to classical sequent calculus [29, 34].

The main result of this thesis is the establishment of a logical formalism for defining specifications of the behavior of components, and properties important for understanding speculative execution. We define the syntax of our typed coordination language (Chapter 2). We use a first-order logic for expressing primitive component as formulas, and compositionally interpret our coordination language as *coordination protocols* (Chapter 3). Coordination games are introduced in Section 3.4. We provide a non-exhaustive overview of components (Chapter 4). We establish the properties of *delay insensitivity*, *independence*, *synchronicity* and *asynchronicity*, *progress* and *termination*, *deadlock-freedom* and *livelock-freedom*, *instantaneousness*, *linearity*, and *causality* (Chapter 5). We finally argue that this thesis is a contribution to the logical foundations of concurrent and distributed systems, and the logical foundations of Reo in particular (Chapter 6).

## 1.1 Speculative Execution

Central processing units (CPUs) with an instruction pipeline architecture employ out-of-order execution as a technique to increase performance of single-threaded code [76]. The behavior of a single-threaded program is defined by the order of its instructions. With out-of-order execution, instructions can execute ahead-

of-time such that the overall system behavior is correspondingly equivalent to that of an in-order execution. Reordering instructions in a pipeline increases the throughput of executed instructions, by efficiently planning the use of computational resources such as arithmetic and logic units (ALUs) and floating point units (FPUs), that results in better run-time performance.

At the level of CPUs, speculative execution is a variant of out-of-order execution. A *speculation* predicts a future state of a processor, and consequent instructions are performed ahead-of-time under the assumption of validity of that future state. At some time after a prediction, actual processor state is compared with the earlier predicted state. If the predicted future state is valid, then the processor has performed a *true speculation* and the consequent instructions correspond to that of an in-order execution. However, if the predicted future state is not valid, then the effects of instructions performed under a *false speculation* need to be reverted.

Speculative execution does not apply to hardware only. For example, speculative multi-threading implements speculative execution in software [18]. Speculative execution is also conceivable in distributed systems.[1]

Speculative execution in concurrent systems can either increase or decrease throughput [44]. The rate of increased throughput depends on the particular implementation technique applied, of which there are multiple. We consider a simplistic model of the two extreme cases: embarrassingly parallel and backtracking.

An *embarrassingly parallel* implementation branches into multiple isolated systems and performs computation in all branches concurrently. The separate branches are isolated systems and thus may not communicate with each other. The number of branches is the size of the domain of the prediction, e.g., a Boolean prediction branches off into two systems. Precisely one branch assumes the true speculation. This technique has the highest gain in throughput because every possibility executes concurrently. Branches that compute based on a false speculation are simply discarded.

A *backtracking* implementation chooses a single branch at a time, as in our previous example of CPUs. We require a branch prediction function, that is typically chosen to maximize the likelihood of a true speculation. The branch prediction function determines which branch to perform first. If this branch is based on a false speculation, the branch is discarded by reverting computation back to the point where the true speculation branches off. This technique requires computation in a branch to be reversible, to be able to revert in the case of a false speculation. Throughput may be affected negatively if one takes a branch assuming a false speculation, since the branch first needs to be reverted before the true computation may commence. Thus, the qualitiy of the branch prediction function greatly impacts the cost of a backtracking implementation.

In summary, we think of these two extremes as a trade-off between time and space. The *embarrassingly parallel* implementation has negligible time overhead, but has a space overhead linear in the number of branches to compute speculatively. The *backtracking* implementation has negligible space overhead, but requires time linear in the depth of the speculative computation that needs to be reverted. Other techniques exists between these extremes: for example, *ea-*

[1] It is an urban legend that sending a specific HTTP responses before any HTTP request is received increases throughput of web-servers. This was supposedly used in early webcam software [63].

*ger execution* executes all branches with negligible space overhead by preemptive scheduling, similar to an iterative deepening search strategy [79].

Throughout this section, we construct a (toy) processing unit to explain speculative execution in more detail. Our toy is used as the running example. The construction of a processing unit is not necessarily centralized (like CPUs), and can also be considered a decentralized processing unit. It is not our goal to construct a fully functional CPU. Our approach is as follows:

1. We construct our processor in a modular fashion to allow reasoning about smaller individual pieces first. Compositionality of reasoning ensures that the end result of composing all pieces together behaves as designed. This explains the basics of how to use Reo to compose components.
2. We give a high-level and low-level explanation of how components communicate and cooperate. This gives us a better intuition for understanding the formal theory, and helps us understand the nature of speculative execution.

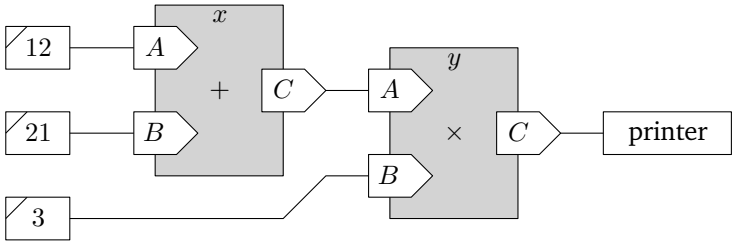
The intended purpose of our example is to demonstrate three layers:

1. Syntax. The functionality of our processor is specified by combining specifications of more primitive components. Our syntax makes this combination explicit and unambiguous.
2. Semantics. Components interact by means of playing a coordination game. The objective of the game is to avoid any inconsistent configurations.
3. Logic. We reason about compositional properties, and the necessity of reversibility of computations based on speculation, to argue that backtracking is an effective implementation strategy.

## 1.2 Running Example

Essential units in processors are arithmetic units. It is well known that arithmetical operations, such as addition, multiplication, and division, are not always constant-time operations: complexity of these operations depens on trade-offs made by processor architects.

Suppose we wish to compute $(12+21)\times 3$. We consider the following two functional components: ADDITION and MULTIPLICATION. The following circuit combines these components, and connects the input to the values 12, 21, 3 and the output to some printing device.




The ADDITION component simultaneously takes the values 12 and 21 and computes their sum. The MULTIPLICATION component takes this result and the value 3 and multiplies them. Finally, the result of 99 is printed.

A component consists of ports to allow for external information flow, or *interaction* with the outside world. These ports form the interface of a component.

An input port allows external information to flow in, and an output port allows information to flow out.

This circuit forms a *composition* of two components. There are two instances: $x$ and $y$. The first is an instance of an ADDITION component, the latter an instance of a MULTIPLICATION component. Each instance of a component has a number of ports, which we write qualified as $x.A$, $x.B$, $x.C$, $y.A$, $y.B$, $y.C$. In our composition, we link ports together to indicate that these ports are identical: here $x.C$ and $y.A$ are linked together to mean that the output port $C$ of $x$ *is* the input port $A$ of $y$. We call this *identification*.

The white boxes around our composition are part of an experimental setup: the values 12, 21, 3 are ready for consumption and the printer is ready to accept an outcome. These are not part of the composition, and instead form a testing environment. Ports $x.A$, $x.B$, $y.B$, $y.C$ are linked to the testing environment.

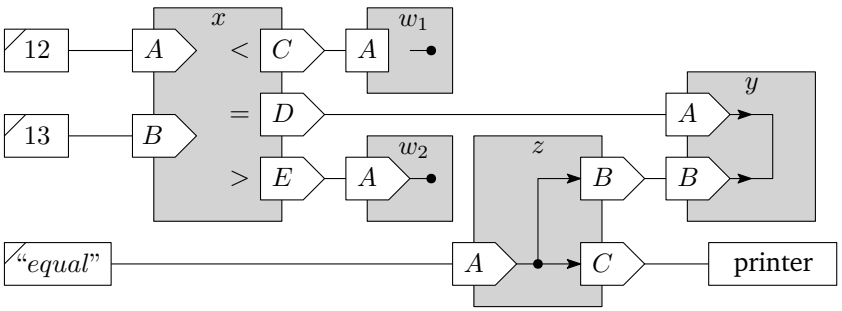
If we were to observe the data that flows through ports during experimentation we can collect a trace. The arithmetical components are functional, meaning that they only operate if all of their inputs are available. The component asynchronously computes its output at some later time. Typically, one abstracts this fact by assuming that arithmetic is performed instantaneously, but in our example we use arithmetic to demonstrate speculative execution.

Other essential units are logical units that can compare values and perform logical operations. The outcome of a logical operation allows so-called branching, where we test a condition and conditionally perform operations.

Consider, for example, comparing 12 with 13. If they are equal, we print some value; otherwise, we do nothing. In a typical imperative programming language, e.g. pseudo Java, one would write:

```
if (12 == 13) {
  return "equal";
}
```

This is given declaratively by the following circuit:



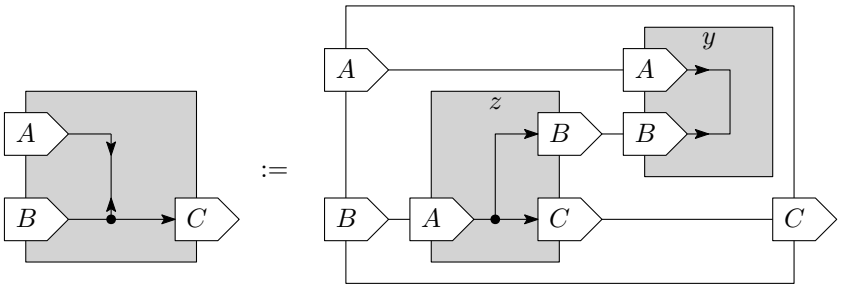

There are five components in this picture, again surrounded by a testing environment. The component $x$ is a comparator that compares $x.A$ and $x.B$. We have two instances of the same component: $w_1$ and $w_2$ are both a so-called GARBAGE component that disposes all its inputs. The component $y$ is a SYNCHRONOUS DRAIN that mediates synchronization in this circuit. Component $z$ is a REPLICATOR that instantaneously transports its input $z.A$ by duplicating it to $z.B$ and $z.C$.

The semantics of this circuit is as follows: if $x.A$ is smaller than $x.B$, then $x.C$ fires a signal; if $x.A$ equals $x.B$, then $x.D$ fires; if $x.A$ is larger than $x.B$, then $x.E$

fires. The REPLICATOR and SYNCHRONOUS DRAIN act as a CONTROL VALVE. If $x.D$ fires a signal, only then the SYNCHRONOUS DRAIN allows an input on $y.B$. However, if $x.D$ does not fire, then $y.B$ is inhibited. This inhibition spreads through the REPLICATOR, blocking its input $z.A$. Thus, in case the two inputs are equal, we have that $y$ synchronizes this result with the REPLICATOR, allowing it to output to $z.C$ into the printer.

In the imperative program, the conditional check is typically assumed to be performed causally before the print statement. In our circuit, the conditional check and the output to the printer are synchronous: they happen instantaneously to the observer. This means that, in principle, the output can be sent to the printer before the conditional check is performed. Intuitively, synchronous actions express atomic groupings of actions, which represents an abstraction of the precise ordering of individual actions, as this ordering is irrelevant.

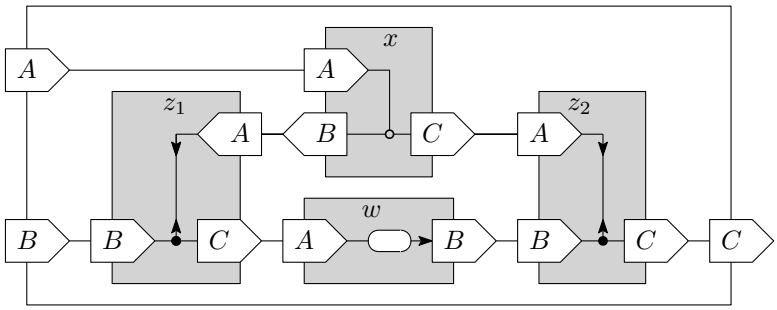
That the REPLICATOR and SYNCHRONOUS DRAIN act as a CONTROL VALVE can be made explicit, by turning it into a composite component:



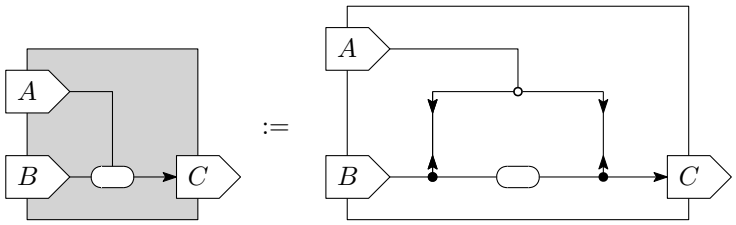

The left-hand side of $:=$ is the graphical mnemonic that defines the composite component on the right-hand side of $:=$. The composite component is defined by a composition of the primitive components $y$ and $z$. Primitive here means that we no longer decompose such component any further. Our composite component has three ports on its boundary: $A$, $B$, $C$. We make explicit which port we mean, by qualifying the ports by its instance. The ports at the boundary of the defined composite component remain unqualified. Here we link $A$ to $y.A$ and $B$ to $z.A$. Internally, we link $z.B$ to $y.B$. The output of $z.C$ is linked to $C$.

Whenever we use the composite component, $A$ and $B$ act as input ports that accept values. We may link only output ports to input ports. From the interior perspective, $A$ and $B$ act as input ports. However, at the exterior of $y$ and $z$, these ports $A$ and $B$ are output ports of the surrounding composition. The direction of a boundary port in the interior is opposite to the exterior. For example, from the perspective of the interior of our composition and the exterior of $y$, we link the input port $A$ to the output port $y.A$. However, the port $y.A$ acts as an input port from the interior perspective of $y$.

Another essential unit in processors are registers and memory banks. Typically, memory in modern CPUs is layed out in a cache hierarchy. This means that accessing memory has variable and dynamic latency. We specify a simple memory bank by first considering memory cells. Memory cells are modeled using CONTROLLED VARIABLES, for which we reuse our earlier defined CONTROL VALVE:




Our composition consists of two CONTROL VALVES ($z_1$ and $z_2$), a ROUTER ($x$), and a VARIABLE ($w$). Port $A$ controls one of the two valves on $B$ and $C$, with a VARIABLE in between. In this picture all input and output ports are explicitly denoted. This quickly clutters the view.

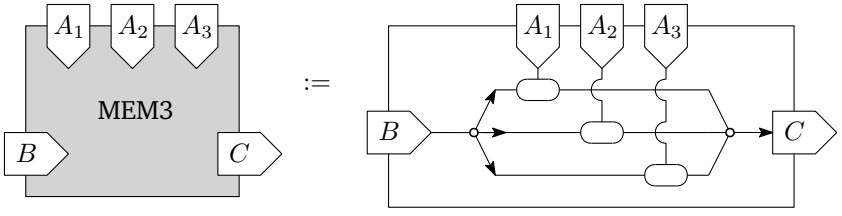



In this picture we no longer draw the component boundaries.

The so-called VARIABLE is a primitive component which we consider in Example **??**. For now, think of VARIABLES as a component that continuously outputs its most recently supplied input value, if any. Thus a VARIABLE may output its value multiple times. Reading of an empty VARIABLE is blocked. VARIABLES may be overwritten. The CONTROLLED VARIABLE only accepts input or supplies output if port $A$ fires. Due to the ROUTER of signal $A$, our CONTROLLED VARIABLE does not accept input and supply output at the same time. It is impossible for all ports to fire simultaneously: $B$ or $C$ fire only if $A$ fires.

A VARIABLE is not reversible because we can overwrite a value without it ever being read: consider that it is possible to overwrite a previously stored value by supplying a value to $B$ twice in a row. The write cannot be reverted, since the previous value is lost in the process of overwriting. Since a VARIABLE is not reversible, so is a CONTROLLED VARIABLE: the presence of $A$, the ROUTER, and CONTROL VALVES does not prevent the possibility of overwriting.

A memory bank consists of multiple memory cells, and a single memory cell can be activated using an address. The address encodes which memory cell is active. We consider a memory bank of three memory cells:




To read from this memory bank, we supply an address to $A$ and observe the output port $C$. To write, we supply an address to $A$ and supply a value to $B$.

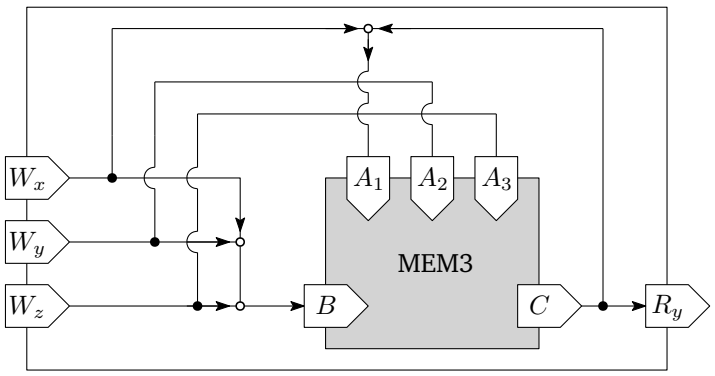
The memory bank consists of these components: a demultiplexer, that takes an address value (e.g. $1,2,3$) and translates it into a signal (e.g. $A_1,A_2,A_3$). We have three CONTROLLED VARIABLES that model three memory cells, connected to these signals: a ROUTER takes the input at $B$ and is connected to the input of each cell. A MERGER takes the output of each cell and merges into the output $C$. Both are depicted with a white dot, but the reader can infer from the direction of the arrow which component is a ROUTER and which is a MERGER.

We assume that the initial value of all memory cells is empty. An empty memory cell cannot be read, meaning that its output port does not fire. For a non-empty memory cell, reading it produces its current value as output.

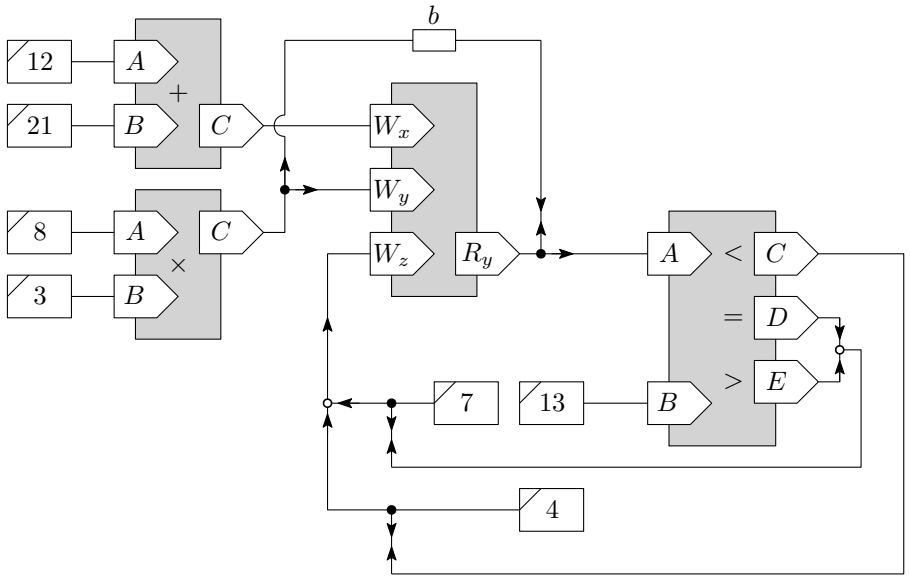
Now consider the following pseudo-Java code, that assigns to three VARIABLES:

```
x = 12 + 21;        W_x
y = 8 * 3;          W_y
if (y < 13) {       R_y
  z = 4;            W_z
} else {
  z = 7;            W_z
}
```

We have the following memory actions: $W_x$, $W_y$, $W_z$ for write actions, and $R_y$ for a read action. Each action happens only once in our snippet. We take that $x,y,z$ are stored in memory cells $1,2,3$. The addresses are encoded by constants. We create the following components:




These memory actions are connected to the arithmetical and logical units:



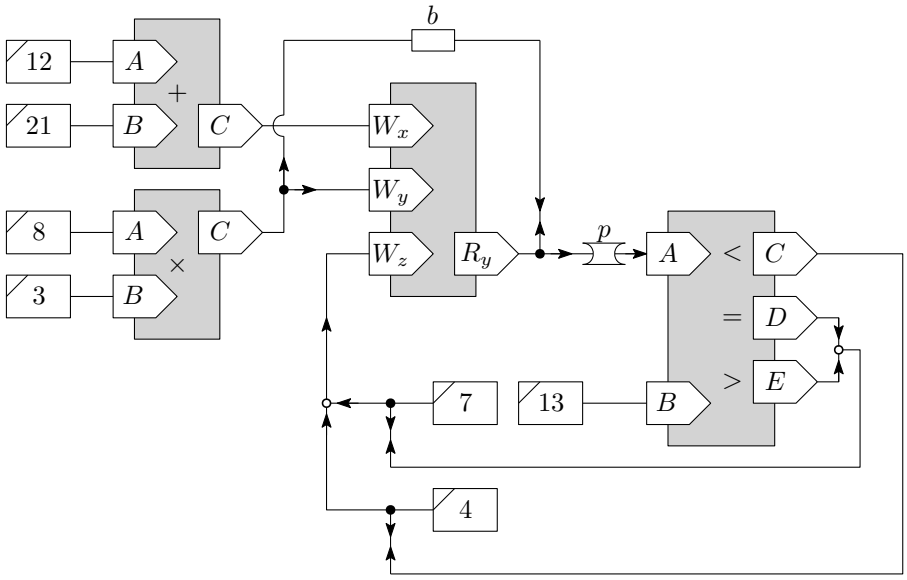

This circuit demonstrates the intended out-of-order behavior of our processor. The $+$ and $\times$ components may independently compute their results. Suppose that $12+21$ was finished computing after $8\times 3$, then $W_y$ happens before $W_x$. Since $R_y$ must happen after $W_y$, we have inserted a BUFFER component (named $x$ in the diagram). The BUFFER adds a causal dependency. After reading $R_y$ and comparing it with 13, we either assign 7 or 4 to $z$: all these actions happen after $W_y$. It is still possible in our circuit that $W_x$ is delayed, after $W_z$.

*A* BUFFER is a component. For now, it intuitively behaves as follows. It waits for an input to arrive, and stores it in its single-place memory. While the BUFFER is full, no new input may arrive. After some time, it may release its output and clears its memory.

In our circuit, the BUFFER $b$ stores the computed value of $y$, but this value is not significant: only the memory value that is output by $R_y$ is being compared. In this circuit these two values are identical. However, in a more complicated example where memory can also be affected by other components, the expected behavior would be to compare the memory value.

The BUFFER is closely related to another component, which we call a PROPHET. A PROPHET generates a speculation, and stores that speculation in its memory. It then waits for the arrival of the true value. If that value equals the speculation the PROPHET reset its speculated memory, similar to a BUFFER that outputs a value, and them becomes empty. If the arrived value is different than the speculation, then it means we had done a false speculation. A PROPHET blocks its input port from firing with the true value, typically resulting in a deadlock.

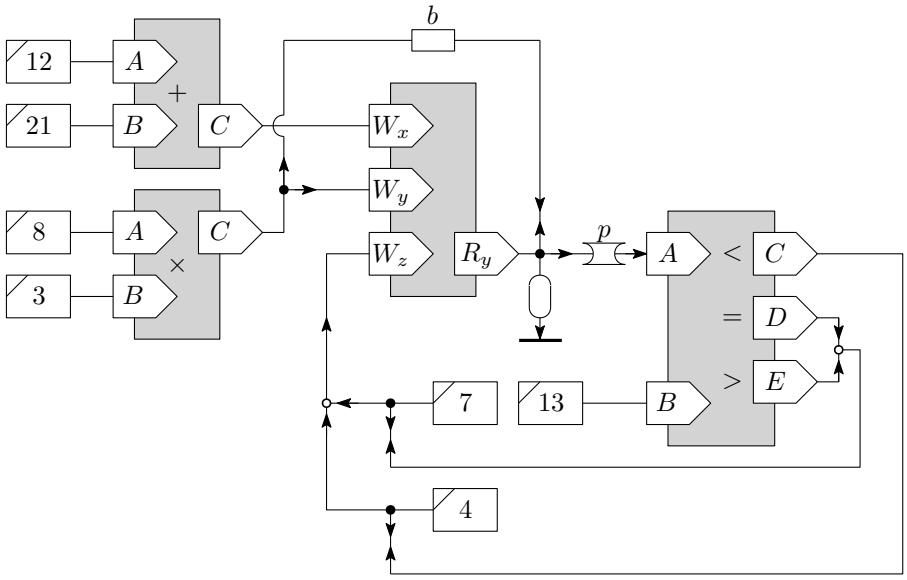
Consider inserting a PROPHET $p$ between the REPLICATOR after $R_y$ and before $A$ of the comparator, as shown in the following circuit:




The intended behavior of this circuit is now different: we speculate on the value of $y$ in our program, before we actually know the value of $R_y$. Suppose that $y < 13$: then $W_z$ fires with 4. Suppose that $y \geq 13$: then $W_z$ fires with 7. Both actions could occur before $W_y$.

If $y \geq 13$ was speculated, and some time later the actual value is read from memory and $R_y$ fires (here with 21), then we have a true speculation. Thus, we have already performed computation (writing 7 to $z$), thereby gaining throughput. Moreover, if $y < 13$ was speculated, and some time later the actual value is read (again 21), then we have a false speculation. Thus, we must discard or revert our computation, thereby losing throughput.

However, there is no guarantee that the value actually be read from memory: the PROPHET allows blocking its input port indefinitely if the speculation differs from the true value. We must enforce that eventually a flow from $R_y$ to the input of the PROPHET takes place, otherwise the components deadlock in the case of a false speculation. We modify our circuit to add the constraint of progress, that forces an inconsistency in case of a false speculation:




The forcing component, called PULL, acts as a suction pipe: adding this component forbids, by specification, that these components enter a deadlock configuration. However, addition of only a PULL means that $R_y$ and $A$ always keep firing, which is too strict. Thus, we additionally add a variable in between the REPLICATOR and the PULL, to indicate that $R_y$ must fire *at least once*.

We explore this example in more depth by deconstructing each component and understanding them individually. For that, we first require a language that allows us to express compositions, interfaces and components.

*This page is intentionally left blank.*

# Chapter 2

# Language

We define a formal language for the construction of components. The formal language works on three levels:

1. Compositions describe how individual components are wired together. We refer to individuals by instance variables and their ports.
2. Interfaces describe the types and the names of boundary ports. We define how to check the interface of a composition.
3. Components comprise layers of compositions of composites and primitives. We have components bound to instance variables, we show how to simplify components by substitution by flattening the layers, and how to check whether a component is well-typed.

Our language is graphical, and was used in the earlier running example. This work on syntax based on Reo is similar in purpose to Dokter's textual Reo [33], in which he allows user-defined component types. The crucial difference is the explicit treatment of nodes in this work, whereas Dokter implicitly uses merger-replicators nodes. Furthermore, we describe a mechanism for type checking compositions and components, and simplification of components.

The theory described in this chapter was developed alongside a prototype Java program that parses, normalizes and type checks compositions and components. The developed prototype is not included with this thesis, since it likely is incorrect, and incomplete, and thus not of high quality.

The design of the type system for checking components and compositions is based on the $\bar{\lambda}\mu\tilde{\mu}$-calculus by Curien and Herbelin [29]. The notion of duality of components and swapping input and output ports, is based on the work by Downen and Ariola [34].

## 2.1 Compositions

We first introduce compositions and how to link components together. We consider well-formedness of compositions and typed compositions. The motivation for doing so is to structure our understanding of composition, and thus allow us to do structural reasoning on compositions.

Let there be a set of *data types*. Data types are denoted $\alpha, \beta, \ldots$, instance variables are denoted $x, y, z, \ldots$, and port variables are denoted $X, Y, Z, \ldots$

**Definition 1.** 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 port variable, and $\alpha$ a type annotation. The type annotation $\alpha$ may be omitted if unambiguous or clear from context.

Let $R$ denote the set of references. References are either *qualified* ($x.X$) or *unqualified* ($X$). Intuitively, references allow us to point to a port: if we point to a port of an instance, then it is a qualified reference; if we point to a boundary port of a composite component, then it is an unqualified reference.

**Definition 2.** A *composition* is as given by the following grammar:

$$c,d,e ::= x \mid (c \parallel d) \mid (c)^{p}_{q}$$

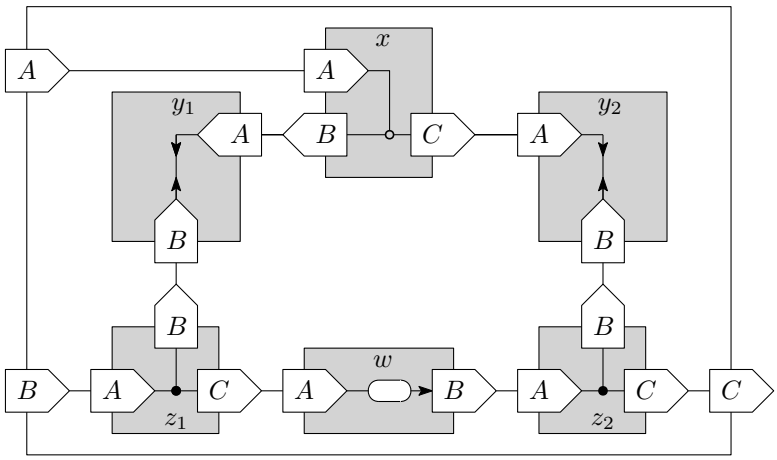
where $x$ is an instance variable, $(c \parallel d)$ a *parallel composition*, and $(c)^{p}_{q}$ an *identification*, such that the following laws hold:

$$(c \parallel c) = c$$
$$(c \parallel d) = (d \parallel c)$$
$$(c \parallel (d \parallel e)) = ((c \parallel d) \parallel e)$$
$$((c)^{p}_{q})^{p}_{q} = (c)^{p}_{q}$$
$$((c)^{p}_{q})^{r}_{s} = ((c)^{r}_{s})^{p}_{q}$$
$$((c)^{p}_{q} \parallel d) = ((c \parallel d))^{p}_{q}$$

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*. Identification $(c)^{p}_{q}$ respects sources and sinks, that is $(c)^{p}_{q} \neq (c)^{q}_{p}$. We have the notion of *occurrence* of instance variables (being atomic compositions) and references (of identifications).

It should be noted that a source or a sink is an output or an input, depending on ones perspective. Whenever a component is used, then its interface is flipped (its inputs become outputs, or equivalently, its sources become sinks) allowing one to connect a source to a sink.



**Example 3.** In the above figure we have the following references: $x.A$, $x.B$, $x.C$ for the ROUTER, $y_1.A$, $y_1.B$ and $y_2.A$, $y_2.B$ for the SYNCHRONOUS DRAINS, $z_1.A$, $z_1.B$, $z_1.C$, and $z_2.A$, $z_2.B$, $z_2.C$ for the REPLICATORS, and $w_1.A$, $w_1.B$ for the VARIABLE: types are omitted. The inner part of the composition is: $(x \parallel y_1 \parallel y_2 \parallel z_1 \parallel z_2 \parallel w)$ formed out of all components without their identifications. Since parallel composition is associative we do not write parentheses. The composition with all identifications is:

$$(\cdots(x \parallel y_1 \parallel y_2 \parallel z_1 \parallel z_2 \parallel w)^{A}_{x.A})^{x.B}_{y_1.A})^{x.C}_{y_2.A})^{B}_{z_1.A})^{z_1.B}_{y_1.B})^{z_1.C}_{w.A})^{w.B}_{z_2.A})^{z_2.B}_{y_2.B})^{z_2.C}_{C}$$

■

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 some 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 $(\cdots(((x \parallel (\ldots \parallel z)))^{p}_{q})\cdots)^{r}_{s}$ such that instances are ordered, and references are lexicographically ordered. All compositions have a unique normal form.

We typically 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.

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 its normal form is $((x \parallel y)^{y.X}_{x.Z})^{y.Z}_{x.Y}$.

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 as both source and sink. This ensures that every reference resolves to an instance that occurs in the composition, and that identification is in some sense affine: a port is never referenced more than once.

**Definition 4.** A composition is *well-formed* if:

- every reference qualified by an instance has that instance occur in the composition,
- every reference is used at most once.

The last condition implies that a reference is used 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 well-formed compositions is: $(x)_Y^{x.X}$ denotes the composition of instance variable $x$ of which its external port $x.X$ is identified with the internal port $Y$ at the boundary of our composition.

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

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).

## 2.2 Interfaces

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.

**Definition 5.** 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 $\langle | \rangle$. We define the following operations on interfaces. Let $X_1, \ldots, X_n$ and $Z_1, \ldots, Z_k$ be port variables. 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 variables on the left-hand side of the $\mid$ in an interface are called *input* port variables and those on the right-hand side are called *output* port variables. The dual of an interface swaps input and output. We may implicitly coerce interfaces to a set of references, being the union of the two disjoint sets of references the interface comprises.

We may also lift union to interfaces. Let $\Delta_1, \Delta_2, \Theta_1, \Theta_2$ be sets of references. Given two interfaces $U = \langle \Delta_1 \mid \Theta_1 \rangle$ and $V = \langle \Delta_2 \mid \Theta_2 \rangle$, by $U \cup V$ we mean the interface obtained by pairwise union of the two parts of the interfaces to form $\langle \Delta_1 \cup \Delta_2 \mid \Theta_1 \cup \Theta_2 \rangle$.

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 an interface $U$. Let a typing context be a set of typing judgments $\Gamma$. We define the relation $\vdash$ between typing contexts and typed compositions, as given by Figure 2.1.

Since atomic compositions are always fully qualified, the union of interfaces never has overlapping names. As a side-condition to these rules, we assume that for $p, \Delta$ it holds that $p \notin \Delta$, and for $\Theta, q$ 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.

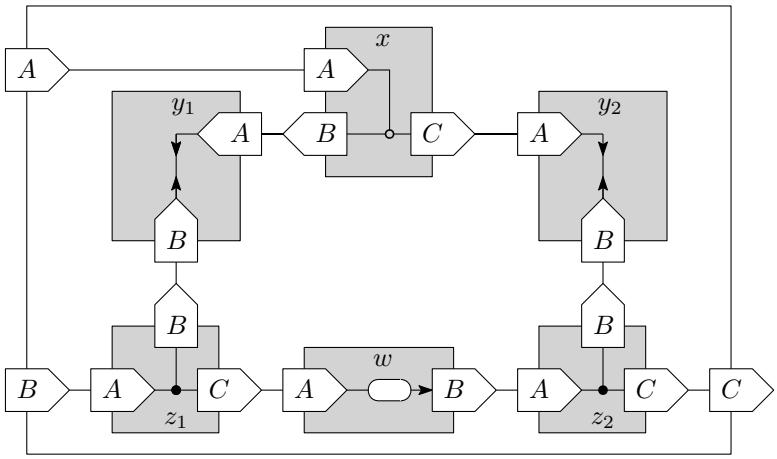
Given a well-formed composition $c$. If there exists a typing context $\Gamma$ and interface $U$ such that $\Gamma \vdash c : U$, then we say that $c$ is a *well-typed composition*. The intention of a well-typed composition is to ensure that references are used

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

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

Figure 2.1: Typing rules for compositions.

linearly and that identification of two references are of the same type. We assume that all compositions we work with in the sequel are well-typed (and thus well-formed), unless stated otherwise.




**Example 6.** In the above figure, the interface of the ROUTER is $\langle B, C \mid A \rangle$, that of the SYNCHRONOUS DRAIN is $\langle \mid A, B \rangle$, that of the REPLICATOR is $\langle B, C \mid A \rangle$ and that of the VARIABLE is $\langle B \mid A \rangle$. These interfaces are flipped because we have used these components in a composition. In the composition $(x \parallel y_1)$, we refer to $x$ and $y_1$ and thus take the qualified interfaces for $x$ and $y_1$: $\langle x.B, x.C \mid x.A \rangle$ and $\langle \mid y_1.A, y_1.B \rangle$, which composed are: $\langle x.B, x.C \mid x.A, y_1.A, y_1.B \rangle$. We can then identify the output $x.B$ and input $y_1.A$, resulting in $\langle x.C \mid x.A, y_1.A \rangle$. The rest of the composition is then formed. Ultimately, we end up with $\langle A \mid B, C \rangle$ as interface for the composition with all identifications.

## 2.3 Components

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, where each primitive component is denoted by name.

**Definition 7.** 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 (see Definition 4), **new** $R$ is a primitive component $R$, and **let** binds the instance variable $x$ in $D$. We consider composite components equal modulo renaming of bound instance variables.

Certain components contain redundancies. We simplify components according to the following three rules, that rewrites the pattern on the left-hand side of $\longrightarrow$ to the right-hand side. The first rule removes bindings for non-occurring instances:

$$(\textbf{let } x = C \textbf{ in } D) \quad \longrightarrow \quad D \tag{2.1}$$

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

$$(\textbf{let } x = (\textbf{let } y = C \textbf{ in } D) \textbf{ in } E) \quad \longrightarrow \quad (\textbf{let } y = C \textbf{ in } (\textbf{let } x = D \textbf{ in } E)) \tag{2.2}$$

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:

$$(\textbf{let } x = c \textbf{ in } C) \quad \longrightarrow \quad C[x.c/x][x] \tag{2.3}$$

A substitution $C[c/x]$ denotes standard substitution of each occurrence of an instance variable $x$ by the composition $c$. A composition $c$ is qualified with instance variable $x$, which is denoted $x.c$, by substituting every occurring unqualified reference $X$ by $x.X$. A non-well-formed composition $c$ is *linked through* the instance variable $x$, 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, as in the example qualifying $(y)^A_B$ with $x$ results in $(y)^{x.A}_{x.B}$ and its substitution of $y$ in $(x)^{x.B}_{x.A}$ results in the 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 $y$.

**Lemma 8.** *Component rewriting using these rules terminates.*

*Proof.* We measure the number **let** bindings and the depth of left nested bindings, and in all rules either one decreases. □

**Example 9.** Take the term **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

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

Figure 2.2: Typing rules for composite components.

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

Once no simplification rule applies anymore, we obtain components in normal form. These components are either of the form of a primitive component, **new** $R$ for some primitive $R$, or a composite component, **let** $x =$ **new** $R_1$ **in** $(\ldots(\mathbf{let}\ z =$ **new** $R_n$ **in** $c)\ldots)$, with zero or more bindings to primitives $R_1$ up to $R_n$. The latter is also written as **let** $x =$ **new** $R_1, \ldots, z =$ **new** $R_n$ **in** $c$.

**Lemma 10.** *Every component has a unique normal form.*

*Proof.* We apply the critical pair method to establish confluence. By confluence and termination, we have unique normal forms. There are five critical pairs:

1. Rules 2.1 and 2.3: (**let** $x = c$ **in** $D$) either discards $x$ because it does not occur in $D$, or substitutes and links through. If $x$ does not occur in $D$, we have $D[x.c/x] = D$ and $D[x] = D$ since substitution without occurrence leaves the term untouched, and linking through without occurrence leaves the term untouched for well-formed compositions.
2. Rules 2.2 and 2.2: (**let** $x =$ (**let** $y =$ (**let** $z = C$ **in** $D$) **in** $E$) **in** $F$) is joined to (**let** $z = C$ **in** (**let** $y = D$ **in** (**let** $z = E$ **in** $F$)))
3. Rules 2.1 and 2.2: (**let** $x =$ (**let** $y = C$ **in** $D$) **in** $E$) reduces in one step to $E$ if $x$ does not occur in $E$, or in three steps if both $x$ and $y$ do not occur in $E$.
4. Rules 2.2 and 2.1: (**let** $x =$ (**let** $y = C$ **in** $D$) **in** $E$) reduces in one step to (**let** $x = D$ **in** $E$) if $y$ does not occur in $D$, or in two steps if $y$ does not occur in $D$ and $E$.
5. Rules 2.2 and 2.3: (**let** $x =$ (**let** $y = c$ **in** $D$) **in** $E$) reduces in one step to (**let** $x = D[y.c/y][y]$ **in** $E$) and in two steps to the same term, since $y$ is not contained in $E$ the same argument as for the first case applies that restricts the effects of substitution and linking through only to the $D$ subterm.

□

We now consider typing judgments and typing contexts of components. Let $U, V$ be unqualified interfaces, and $C :: U$ a typing judgment, and let the typing context $\Gamma$ 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 2.2. We consider the rules from left to right:

1. 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.

2. This is a family of rules, one rule for each primitive component $R$ with unqualified interface $U$. We admit this rule for each primitive component together with $U$ as its unqualified interface.
3. This rule shows that $x$ can be bound in $D$. This dualizes the interface of $C$ and binds it 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 $D$. By $\Gamma,\Delta$ we mean that $\Gamma$ and $\Delta$ are disjoint: this ensures that an instance can be used in only a single composition.

A component $C$ such that $\emptyset \vdash C$ is derivable is called a *closed component*.

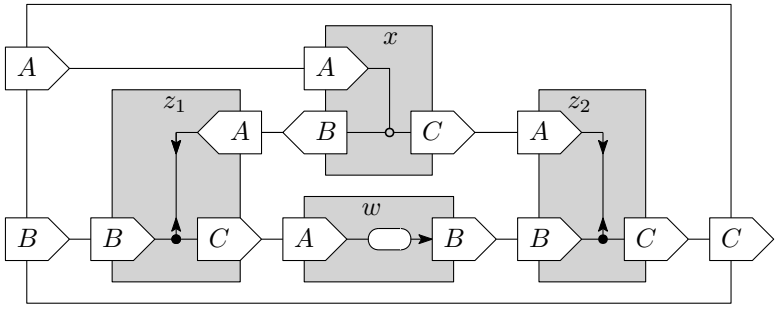
**Example 11.** We now take a component from before and show how it is normalized. The first things to fix are the primitive components, of router :: $\langle A \mid B,C\rangle$, valve :: $\langle A \mid B,C\rangle$ and variable :: $\langle A \mid B\rangle$. The component valve is formed by composing replicator :: $\langle A \mid B,C\rangle$ and drain :: $\langle A,B \mid\rangle$ as:

$$\textbf{let } x = \textbf{new } \text{replicator},\ y = \textbf{new } \text{drain } \textbf{in } ((((x \parallel y)^{A}_{y.A})^{B}_{x.A})^{x.B}_{y.B})^{x.C}_{C}$$

this forms the following derivation:

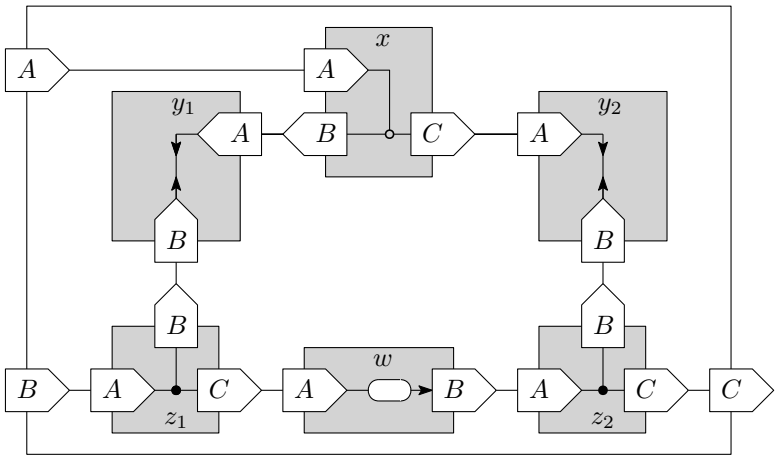
$$\dfrac{\vdash \textbf{new } \text{replicator} :: \langle A \mid B,C\rangle \qquad \dfrac{\vdash \textbf{new } \text{drain} :: \langle A,B \mid\rangle \qquad \dfrac{\vdots}{x :: \langle B,C \mid A\rangle, y :: \langle \mid A,B\rangle \vdash c :: \langle A \mid B,C\rangle}}{x :: \langle B,C \mid A\rangle \vdash \textbf{let } y = \textbf{new } \text{drain } \textbf{in } c :: \langle A \mid B,C\rangle}}{\vdash \textbf{let } x = \textbf{new } \text{replicator},\ y = \textbf{new } \text{drain } \textbf{in } c :: \langle A \mid B,C\rangle}$$

where $c$ is our composition $((((x \parallel y)^{A}_{y.A})^{B}_{x.A})^{x.B}_{y.B})^{x.C}_{C}$ that can be checked to be well-typed with the unqualified interface $\langle A \mid B,C\rangle$. Let $C$ denote this closed component. We can form the component of the controlled memory by reusing our earlier component, as follows:




$$\textbf{let } x = \textbf{new } \text{router},\ z_1 = C,\ z_2 = C,\ y = \textbf{new } \text{variable } \textbf{in}$$
$$(\cdots(x \parallel z_1 \parallel z_2 \parallel y)^{A}_{x.A})^{B}_{z_1.B})^{x.B}_{z_1.A})^{x.C}_{z_2.A})^{y.B}_{z_2.B})^{z_1.C}_{y.A})^{z_2.C}_{C}$$

After normalization and renaming we obtain the component shown below:




**let** $x =$ **new** router, $y_1 =$ **new** drain, $z_1 =$ **new** replicator,
$y_2 =$ **new** drain, $z_2 =$ **new** replicator, $w =$ **new** variable **in**
$(\cdots(w \parallel x \parallel y_1 \parallel y_2 \parallel z_1 \parallel z_2)^{A}_{x.A})^{B}_{z_1.A})^{w.B}_{z_2.A})^{x.B}_{y_1.A})^{x.C}_{y_2.A})^{z_1.B}_{y_1.B})^{z_1.C}_{w.A})^{z_2.B}_{y_2.B})^{z_2.C}_{C}$

*This page is intentionally left blank.*

# Chapter 3

# Foundation

We consider the formulation of constraints over time to specify the behavior of components. These formulas are called *coordination protocols*. Every primitive component is specified by its interface and its coordination protocol. Of a composite component, we inductively construct its coordination protocol from the structure of composition and the coordination protocols of its underlying primitives.

To do so formally, in this chapter we describe a particular many-sorted logic and its standard interpretations as data constraints on streams. This is useful, since we can not only use the logical formalism to define the behavior of primitive components, but also properties of components.

1. We first lay down the mathematical preliminaries required in the rest of this chapter: the domains of data types and streams.
2. We define our many-sorted logic and introduce standard interpretations. Satisfiability can be understood as finding sets of assignments of ports to streams.
3. We specify the behavior of primitive components by coordination protocols. We consequently lift component specifications for composite components by induction on their construction.
4. We define coordination games as the interaction between environment and component.

Our loigcal formalism is inspired by Dokter's formal rule-based semantics [32]. He defines coordination protocols as relations of data streams. He defines a modal logic equiped with stream constraints that are constructed using stream head equality, stream derivatives, and modal operators. In our formalization, we employ standard first-order logic to define coordination protocols. It seems that this change does not result in the loss of any expressiveness.

## 3.1 Data Streams

Data types are algebraic structures. Streams are used to model the flow of data. By $\mathbb{N}$ we denote the set of *natural numbers* $0, 1, 2, 3, \ldots$

Let a *data type* be a structure $(D, *)$ where $D$ is a carrier set and $* \in D$ is a designated constant. We speak of *data elements* as those elements in $D$ different from $*$. Whenever we speak of *elements* or *values*, we mean data elements or $*$.

Intuitively, one may think of $*$ as standing for the absence of data, being a 'null' value. We assume equality of data types is decidable. We write $\alpha, \beta, \ldots$ to denote data types. As convention, we write $a \in \alpha$ to mean some element $a$ of the carrier set. The set $\mathbb{N}$ is a data type $(\mathbb{N}, 0)$ where we take $* = 0$.

Let a stream be a function from natural numbers to some set. We denote streams by the Greek letters $\sigma, \tau, \ldots$. Intuitively, one thinks of streams as an enumeration. *Data streams* are functions from naturals to data types, e.g. $\sigma : \mathbb{N} \to \alpha$ for some data type $\alpha$. Alternatively, we may define streams by stream differential equation. See [71, 70] for an elementary introduction.

A stream differential equation for some stream $\sigma$ is given by its initial value $\sigma(0)$ and its stream derivative $\sigma'$. The derivative itself is also a stream such that $\sigma'(x) = \sigma(x+1)$. We have the repeated derivatives $\sigma''$, $\sigma'''$, 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, let $\text{Nat} = \{\mathbb{N}, *\}$ be some data type where $* = 0$. 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.

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. There are at least as many streams as there are natural numbers: let $n$ a natural number, then we can construct the stream $(n, *, \ldots)$, where the first element is $n$, followed by $*$ repeated forever.

Suppose towards contradiction that there are as many natural numbers as there are streams. We enumerate all streams in a table: let $\sigma_0$ denote the first stream, $\sigma_1$ the second stream, and so on. Look at the diagonal and construct a stream $\tau$ such that $\tau(x) = \sigma_x(x) + 1$. Stream $\tau$ differs from each enumerated stream $\sigma_x$ 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.

Equality of streams is established by bisimilarity [15]. 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 $\sigma, \tau$ are bisimilar if there exists a bisimulation relation $R$ such that $(\sigma, \tau) \in R$, and we write $\sigma = \tau$.

## 3.2 Logical Formalism

We now consider a many-sorted first-order logic with equality. The general structure of this section follows much from [35]:

1. We define syntax: what we mean by signature, terms, and formulas.
2. We define semantics: what we mean by standard interpretation.
3. We define the notions of assignment, solution, satisfiability and validity.

What is different than usual is our treatment of streams, and quantification over streams. The intention of our logical formalism is to be able to use tools such as interactive theorem provers (e.g. Coq, Isabelle, or Lean) to implement our semantics. Further implementation within these tools is out of the scope of this thesis.

We first start with the definition of signatures. Signatures may contain more non-logical symbols than those given here: we specify only the minimal requirements. This treatment conveys openness and extensibility of our formalism.

**Definition 12.** $\Sigma = (S, F, P)$ is a *signature* consisting of the following data:

- $S$ is a set of *sorts*, such that:
  - for each data type $\alpha$ there is a distinct sort $\alpha \in S$,
  - for each data type $\alpha$ there is a distinct sort $(\mathbb{N} \to \alpha) \in S$.
- $F$ is a set of *function symbols*, such that:
  - for each data type $\alpha$, $*_\alpha \in F$ with arity $\langle \alpha \rangle$,
  - for each data value $d \in \alpha$, $d_\alpha \in F$ with arity $\langle \alpha \rangle$,
  - $+ \in F$, $- \in F$, $\times \in F$ with arities $\langle \mathbb{N} \rangle$, $\langle \mathbb{N}, \mathbb{N} \rangle$, and $\langle \mathbb{N}, \mathbb{N}, \mathbb{N} \rangle$,
  - for each data type $\alpha$, $\mathrm{at}_\alpha \in F$ with arity $\langle (\mathbb{N} \to \alpha), \mathbb{N}, \alpha \rangle$,
  - for each data type $\alpha$, $\mathrm{skip}_\alpha \in F$ with arity $\langle (\mathbb{N} \to \alpha), \mathbb{N}, (\mathbb{N} \to \alpha) \rangle$.
- $P$ is a set of *predicate symbols*, such that:
  - $\bot, \top \in P$ with arity $\langle \rangle$,
  - for each data type $\alpha$, $=_\alpha \in P$ with arity $\langle \alpha, \alpha \rangle$,
  - $\leq \in P$ with arity $\langle \mathbb{N}, \mathbb{N} \rangle$.

An *arity* is a list of sorts $\langle s_1, \ldots, s_n \rangle$ where $s_1, \ldots, s_n \in S$. Each function symbol has an associated non-empty arity. Each predicate symbol has an associated arity.

Since $\mathbb{N}$ is a data type, we also have a distinct sort $\mathbb{N} \in S$. Any natural number $0, 1, 2, 3, \ldots$ can be taken as a constant. We now fix some signature $\Sigma = (S, F, P)$. The definitions for terms and formulas are not surprising. Each term is assigned to a sort. We define terms inductively over all sorts simultaneously. Formulas are also defined inductively.

**Definition 13.** Let $s \in S$ be a sort. A *term of sort* $s$ is formed by:

- a variable $x$ of sort $s$ is an atomic term of sort $s$ denoted $x^s$,
- if $c \in F$ is a function symbol of arity $\langle s \rangle$, then $c$ is an atomic term of sort $s$,
- if $t_1, \ldots, t_n$ are terms of sorts $s_1, \ldots, s_n$ and $f \in F$ is a function symbol of arity $\langle s_1, \ldots, s_n, s_{n+1} \rangle$, then $f(t_1, \ldots, t_n)$ is a term of sort $s_{n+1}$.

**Definition 14.** A *first-order formula* is:

- if $p \in P$ is a predicate symbol of arity $\langle \rangle$, then $p$ is an atomic formula,
- if $t_1, \ldots, t_n$ are terms of sort $s_1, \ldots, s_n$ and $p \in P$ is a predicate symbol of arity $\langle s_1, \ldots, s_n \rangle$, then $p(t_1, \ldots, t_n)$ is an atomic formula,
- non-atomic formulas are formed with connectives $\neg$, $\wedge$, $\vee$, $\to$,
- non-atomic formulas are formed by quantifiers $\exists x^s, \forall x^s$.

We define *standard interpretations* to fix the interpretation of the non-logical symbols we have introduced in our signature.

**Definition 15.** A *standard interpretation* $\mathcal{M}$ of signature $\Sigma = (S, F, P)$ consists of:

- a map of sorts to domains such that $s \in S$ maps to domain $s^{\mathcal{M}}$, such that:
  - $\alpha^{\mathcal{M}}$ is the carrier set of data type $\alpha$,
  - the sort $(\mathbb{N} \to \alpha) \in S$ is mapped to data streams $\mathbb{N} \to \alpha$ of data type $\alpha$.
- a map of function symbols to domain functions such that $f \in F$ with arity $\langle s_1, \ldots, s_n, s_{n+1} \rangle$ maps to a function $f^{\mathcal{M}} : s_1^{\mathcal{M}} \times \cdots \times s_n^{\mathcal{M}} \to s_{n+1}^{\mathcal{M}}$, and $c \in F$ with arity $\langle s \rangle$ maps to $s^{\mathcal{M}}$, such that:
  - $*_\alpha^{\mathcal{M}}$ is the null value $* \in \alpha$,

  - $d_\alpha^{\mathcal{M}}$ is the data value $d \in \alpha$,
  - $+, -, \times$ are interpreted as the arithmetical functions of plus, monus[1] and times,
  - $\text{at}_\alpha$ is interpreted as a function $\text{at}_\alpha^{\mathcal{M}} : (\mathbb{N} \to \alpha) \times \mathbb{N} \to \alpha$ such that $\text{at}_\alpha^{\mathcal{M}}(\sigma)(n) = \sigma(n)$,
  - $\text{skip}_\alpha$ is interpreted as a function $\text{skip}_\alpha^{\mathcal{M}} : (\mathbb{N} \to \alpha) \times \mathbb{N} \to (\mathbb{N} \to \alpha)$ such that $\text{skip}_\alpha^{\mathcal{M}}(\sigma)(n) = \sigma^{(n)}$.
- a map of predicate symbols to domain relations such that $p \in P$ with arity $\langle s_1, \ldots, s_n \rangle$ maps to a relation $p^{\mathcal{M}} : s_1^{\mathcal{M}} \times \cdots \times s_n^{\mathcal{M}}$, and $p \in P$ with arity $\langle\rangle$ maps to propositions, such that:
  - $\bot^{\mathcal{M}}$ never holds and $\top^{\mathcal{M}}$ always holds,
  - $=_\alpha^{\mathcal{M}}$ is equality of values of data type $\alpha$,
  - $\leq^{\mathcal{M}}$ is the relation of less than or equals between naturals.

Let $\mathcal{M}$ denote a fixed standard interpretation. Towards defining satisfiability and validity, we define the notion of assignment. This is necessary, since not every domain element has a corresponding term.

**Definition 16.** An *assignment* $\beta$ is a map from variables to domain elements, where variables $x^s$ are mapped to elements in $s^{\mathcal{M}}$.

Given the interpretation of function symbols in $\mathcal{M}$, an assignment can be extended to a map from terms to domain elements, defined inductively on the structure of terms. Similarly, given the interpretation of predicate symbols in $\mathcal{M}$, an assignment can be extended to a map from formulas to propositions, defined inductively on the structure of formulas, making use of the previously extended assignment to a map from terms to domain elements.

The *satisfiability* of a formula $\phi$ is denoted $\beta \vDash \phi$ and is defined to be equivalent to the truth of $\phi$ interpreted as a proposition given assignment $\beta$. The *validity* of a formula $\phi$ is denoted $\vDash \phi$ and holds if and only if $\beta \vDash \phi$ holds for all assignments $\beta$.

**Proposition 17.** *If $\beta_1, \beta_2$ are two assignments and $\beta_1(x^s) = \beta_2(x^s)$ for all free variables $x^s$ in $\phi$, then $\beta_1 \vDash \phi$ and $\beta_2 \vDash \phi$.*

An assignment that is restricted to map only the free variables of a formula $\phi$ is called a *solution* for $\phi$. We have a special class of formulas:

**Definition 18.** A *coordination protocol* is a first-order formula $\phi$, where all free variables $x^s$ of $\phi$ must be of sort $s = (\mathbb{N} \to \alpha)$ for some $\alpha$. These free variables are also called the *ports* (of data type $\alpha$) of $\phi$.

*Remark* 19. We treat sort annotations implicitly, to prevent clutter. We also use a more convenient notation for $\text{at}_\alpha$ and $\text{skip}_\alpha$: let $X$ be a port of sort $(\mathbb{N} \to \alpha)$ and $t$ be a variable of sort $\mathbb{N}$, then the term $\text{at}_\alpha(X, t)$ is written as $X(t)$ and the term $\text{skip}_\alpha(X, t)$ is written as $X^{(t)}$. We call such terms *applications* and *derivations,* respectively. Additionally, we use $t > s$ for $\neg(t \leq s)$, and $s < t$ for $t > s$, and $s = t$ for $s \leq t \wedge t \leq s$.

[1] Monus is minus for natural numbers by rounding negative numbers to 0.

## 3.3 Coordination Protocols

We employ coordination protocols to encode our intuition of the behavior of components. Formally, coordination protocols are formulas as defined in last section. The set of solutions of a coordination protocol is a set of tables of observations. It was found that this idea has an origin in Kleene's 1951 paper on representation of events [53]. Informally, observations represent a consistent snapshot of the data flowing through ports of a component, made by an independent observer. Tables of such observations capture behavior of a component over time. A set of tables of observations corresponds to accepting certain such behaviors and rejecting others.

**Definition 20.** The coordination protocol $\phi$ induces a set $\mathcal{L}(\phi) = \{\beta \mid \beta \vDash \phi\}$ of solutions of $\phi$.

A coordination protocol $\phi$ is called inconsistent it $\mathcal{L}(\phi)$ is empty. Intuitively, we consider the set $\mathcal{L}(\phi)$ of solutions as a set of tables of observations. Consider these examples:

**Example 21.** Let $X$ and $Y$ be ports of type $\text{Signal} = \{*, 0\}$. The coordination protocols $\forall t.X(t) = *$ and $\forall t.Y(t) = 0$ have one free variable: $X$ and $Y$, respectively. The sets of tables of observations are shown below. Both contain single solution.

$$\left\{ \begin{array}{|cc|} \hline t & X \\ \hline 0 & * \\ 1 & * \\ 2 & * \\ \vdots & \vdots \\ \hline \end{array} \right\} \qquad \left\{ \begin{array}{|cc|} \hline t & Y \\ \hline 0 & 0 \\ 1 & 0 \\ 2 & 0 \\ \vdots & \vdots \\ \hline \end{array} \right\}$$

Another example is the coordination protocol $\forall t.X(t) = * \vee Y(t) = *$ that has two free variables: $X$ and $Y$. Its set of tables of observations is shown below. This set contains any solution $X \mapsto \sigma, Y \mapsto \tau$ where $\sigma$ and $\tau$ are data streams such that $\sigma(t) = *$ or $\tau(t) = *$ for any $t \in \mathbb{N}$.

$$\left\{ \begin{array}{|ccc|} \hline t & X & Y \\ \hline 0 & * & * \\ 1 & * & * \\ 2 & * & * \\ \vdots & \vdots & \vdots \\ \hline \end{array} , \begin{array}{|ccc|} \hline t & X & Y \\ \hline 0 & 0 & * \\ 1 & * & * \\ 2 & * & * \\ \vdots & \vdots & \vdots \\ \hline \end{array} , \ldots, \begin{array}{|ccc|} \hline t & X & Y \\ \hline 0 & * & 0 \\ 1 & 0 & * \\ 2 & * & * \\ \vdots & \vdots & \vdots \\ \hline \end{array} , \ldots, \begin{array}{|ccc|} \hline t & X & Y \\ \hline 0 & 0 & * \\ 1 & * & 0 \\ 2 & * & 0 \\ \vdots & \vdots & \vdots \\ \hline \end{array} , \ldots \right\}$$

■

Consider two coordination protocols $\phi$ and $\psi$ that do not have any free variables in common. The protocol $\mathcal{L}(\phi)$ consists only of solutions of $\phi$, and similar for $\mathcal{L}(\psi)$ and solutions of $\psi$. The intersection of these two sets is empty, however $\mathcal{L}(\phi \wedge \psi)$ is not empty. In $\mathcal{L}(\phi \wedge \psi)$, every solution in $\mathcal{L}(\phi)$ is paired with every solution in $\mathcal{L}(\psi)$ and glued together.

**Example 22.** The solutions of $\forall t.X(t) = *$ and $\forall t.Y(t) = 0$ can be glued together to form the coordination protocol:

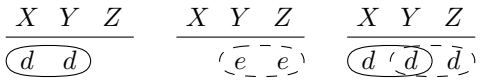



Figure 3.1: Intersection of frame conditions.

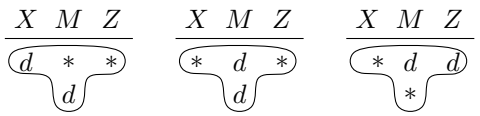



Figure 3.2: Three frame conditions that make up a BUFFER.

$$\begin{array}{ccc} t & X & Y \\ \hline \left\{ \begin{array}{ccc} 0 & * & 0 \\ 1 & * & 0 \\ 2 & * & 0 \end{array} \right\} & & \\ \vdots & \vdots & \vdots \end{array}$$

■

We illustrate our intuition using *frame conditions*.

**Example 23.** The frame conditions $X(0) = Y(0)$ and $Y(0) = Z(0)$. This condition applies to only the first row. These two frame conditions are overlapped, they restrict the allowed observations in the first element, as in Figure 3.1. 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. ■

If we want a frame condition to persists over time, then it must constrain every row. The intuition of universal quantification is to *slide* the frame condition over all rows.

**Example 24.** In Figure 3.1, the first frame condition then is $X(t) = Y(t)$ and we universally quantify over time $t$, to obtain $\forall t.X(t) = Y(t)$. The second has as frame condition $Y(t) = Z(t)$, and universally quantified it becomes $\forall t.Y(t) = Z(t)$. Hence, the first coordination protocol only has solutions for which at every time $X$ and $Y$ have the same value; it does not restrict $Z$ in any way. Similar for the second coordination protocol. Conjunction of the two protocols, $(\forall t.X(t) = Y(t)) \wedge (\forall t.Y(t) = Z(t))$, results in the constraint that all three ports, at all times, must have the same value. ■

**Example 25.** A frame condition that spans multiple rows is the example in Figure 3.2. The condition ranges over two rows. The first frame condition specifies that: if port $X$ has some value $d$ and $M$ has no value and $Z$ has no value, then $d$ must be in the next row of $M$. The intuition here is that $M$ acts as a sort of memory, that is updated by constraining its value in the next row. The second frame condition specifies that memory is retained whenever there is no input or output. The third frame condition specifies that the contents of memory is the same as at port $Z$

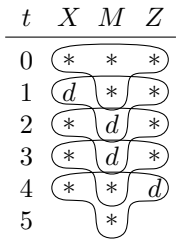



Figure 3.3: Example sequence of solutions, with constraints of a BUFFER.

and that the memory is cleared in the next row. We now obtain the coordination protocol:

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

We have the quantifier $\forall t.$ on the outer level to ensure that for each row, one of these three frame conditions applies. If we would take $(\forall t....) \vee (\forall t....) \vee (\forall t....)$ as coordination protocol, then we accept streams that have for all rows, either only the first, only the second, or only the third frame condition, and that is not our intended result.

A demonstration of how the three frame conditions apply to an arbitrary solution is given in Figure 3.3. Here, 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 3.4. These frame conditions apply for each row and are specified by:

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

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 $*$. 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 turn to primitive components and composite components. The main point is to associate each primitive component to a coordination protocol. We then map composite components to coordination protocols, by induction on the construction of compositions.

**Definition 26.** A *component specification* $\phi(U)$ consists of the following data:

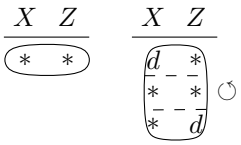



Figure 3.4: Alternative frame conditions that make up a BUFFER.

- an interface $U = \langle X_1, \ldots, X_n \mid Z_1, \ldots, Z_k \rangle$,
- a coordination protocol $\phi$.

The free variables of the coordination protocol $\phi$ are all ports assigned in solutions; the interface marks which ports are not hidden. Without loss of generality, we may assume that all free variables of $\phi$ are precisely the ports occurring in $U$: all hidden variables are existentially quantified, and for a non-occurring port $Y$ we add a trivial clause $\forall t.Y(t) = Y(t)$.

The semantics of a component specification is the semantics of the coordination protocol, $\mathcal{L}(\phi) = \{\beta \mid \beta \vDash \phi\}$ where $\beta$ are solutions that assign streams to the ports occurring in $U$. We use the notation $\mathcal{L}(\phi(U))$ to indicate that $\phi$ and thus $\mathcal{L}(\phi)$ depend on $U$. We introduce two abbreviations, useful for writing formulas that are indexed over interfaces:

Given an interface $U = \langle X_1 : \alpha_1, \ldots, X_n : \alpha_n \mid Z_1 : \alpha_{n+1}, \ldots, Z_k : \alpha_{n+k} \rangle$. We write $\forall \vec{Y} : U.\phi$ as an abbreviation for the indexed quantification $\forall Y_1^{\alpha_1}.\forall Y_2^{\alpha_2}.\ldots.\forall Y_{n+k}^{\alpha_{n+k}}.\phi$. Here $Y_i$ are the bound variables, that correspond for $1 \leq i \leq n$ to the input ports and for $n < i \leq n+k$ to the output ports. A similar abbreviation is used for $\exists \vec{Y} : U.\phi$. In the case $U$ is empty, then $\forall \vec{Y} : U.\phi$ is $\top$ and $\forall \vec{Y} : U.\phi$ is $\bot$. Similarly, we write $\bigvee_{i \in U} \phi$ for an indexed disjunction and $\bigwedge_{i \in U}$ for an indexed conjunction. Given a component specification $\phi(U)$ then by $\phi(\vec{Y})$ within an indexed quantification, we mean the simultaneous substitution of the free variables $X_1, \ldots X_n, Z_1, \ldots Z_k$ in $\phi$ by those of $\vec{Y}$: namely the coordination protocol $\phi[Y_1/X_1, \ldots, Y_n/X_n, Y_{n+1}/Z_1, \ldots Y_{n+k}/Z_k]$.

We lift the operations of compositions, namely parallel composition and identification, to component specifications. Given two components specifications $\phi(U)$ and $\psi(V)$ where $U$ and $V$ are disjoint. The parallel composition $\phi(U) \parallel \psi(V)$ is defined to be $(\phi \wedge \psi)(U \cup V)$. Identification on component specification is defined only for occurring ports. Let $U = \langle A : \alpha \mid B : \alpha \rangle \cup V$ be an interface with at least two ports, an input port $A$ and an output port $B$ of the same data type. Then $(\phi(U))_B^A$ is defined to be $(\phi \wedge \forall t.A(t) = B(t))(V)$.

The component specification for composite components can be constructed by induction on the structure of components as follows. For each primitive component, we assume its interface and coordination protocol are a priori given. We assume a composite component $C$ is normalized, meaning it is a sequence of let bindings to primitive components with a composition as deepest term. For each let binding **let** $x =$ **new** $R$ **in** $C$, we substitute the component specification that is known for $R$ for the instance variable $x$ in $C$. Ultimately we end up with a composition of component specifications and the previous definition applies.

**Example 27.** Suppose we have the following primitive component specifications:

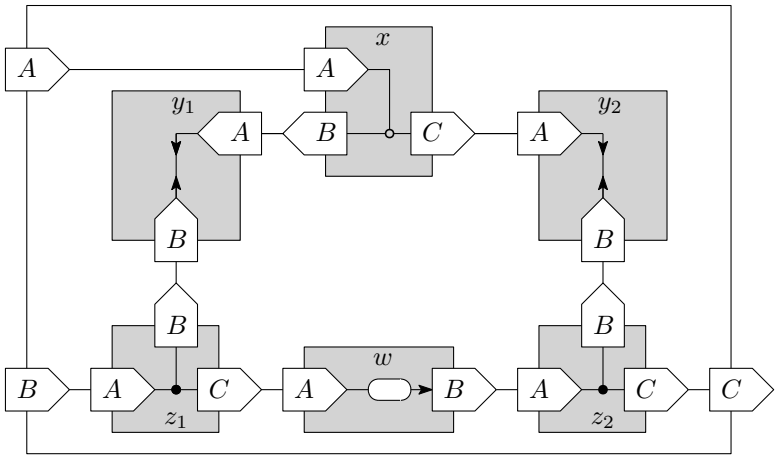
DRAIN $\quad \langle A, B \mid \rangle \forall t.(A(t) = * \leftrightarrow B(t) = *)$

REPLICATOR $\quad \langle A \mid B, C \rangle \forall t.(A(t) = B(t) \wedge A(t) = C(t))$

ROUTER $\quad \langle A \mid B, C \rangle \forall t.(A(t) = B(t) \wedge C(t) = * \vee A(t) = C(t) \wedge B(t) = *)$

VARIABLE $\langle A \mid B \rangle M(0) = * \wedge$
$\forall t.( \quad B(t) = * \quad \wedge M(t) = * \wedge M(t+1) = A(t) \ \vee$
$A(t) = * \wedge B(t) = * \quad \wedge M(t) \neq * \wedge M(t+1) = M(t) \vee$
$A(t) = * \wedge B(t) = M(t) \wedge M(t) \neq * \wedge M(t+1) = M(t) \vee$
$A(t) \neq * \wedge B(t) = * \quad \wedge M(t) \neq * \wedge M(t+1) = A(t)$
$A(t) \neq * \wedge B(t) = M(t) \wedge M(t) \neq * \wedge M(t+1) = A(t))$

We compose them into the following component:




Next, we qualify each reference by its instance variable, we take the conjunction of all formulas, and add identifications. This results in the following component:

$\langle A, B \mid C \rangle \forall t.(y_1.A(t) = * \leftrightarrow y_1.B(t) = *) \wedge$
$\forall t.(y_2.A(t) = * \leftrightarrow y_2.B(t) = *) \wedge$
$\forall t.(z_1.A(t) = z_1.B(t) \wedge z_1.A(t) = z_1.C(t)) \wedge$
$\forall t.(z_2.A(t) = z_2.B(t) \wedge z_2.A(t) = z_2.C(t)) \wedge$
$\forall t.(x.A(t) = x.B(t) \wedge x.C(t) = * \vee x.A(t) = x.C(t) \wedge x.B(t) = *) \wedge$
$\forall t.( \quad w.B(t) = * \quad \wedge w.M(t) = * \wedge w.M(t+1) = w.A(t) \ \vee$
$w.A(t) = * \wedge w.B(t) = * \quad \wedge w.M(t) \neq * \wedge w.M(t+1) = w.M(t) \vee$
$w.A(t) = * \wedge w.B(t) = w.M(t) \wedge w.M(t) \neq * \wedge w.M(t+1) = w.M(t) \vee$
$w.A(t) \neq * \wedge w.B(t) = * \quad \wedge w.M(t) \neq * \wedge w.M(t+1) = w.A(t)$
$w.A(t) \neq * \wedge w.B(t) = w.M(t) \wedge w.M(t) \neq * \wedge w.M(t+1) = w.A(t)) \wedge$
$w.M(0) = * \wedge$
$\forall t.(A(t) = x.A(t)) \wedge \forall t.(B(t) = z_1.A(t)) \wedge \forall t.(x.B(t) = y_1.A(t)) \wedge$
$\forall t.(x.C(t) = y_2.A(t)) \wedge \forall t.(y_1.B(t) = z_1.B(t)) \wedge \forall t.(y_2.B(t) = z_2.B(t)) \wedge$
$\forall t.(z_1.C(t) = w.A(t)) \wedge \forall t.(w.B(t) = z_2.A(t)) \wedge \forall t.(z_2.C(t) = C(t))$

Recall that all ports that do not occur in the interface $\langle A, B \mid C \rangle$ are implicitly existentially quantified.

An important aspect of reasoning about coordination protocols is the notion of ideal environment. Let $\phi(U)$ be a component specification. If $\phi(U)$ is satisfiable, it means there exists a non-empty $\mathcal{L}(\phi)$ that consists of assignments of the ports in $U$ to data streams. If $\phi(U)$ is unsatisfiable, it means that $\mathcal{L}(\phi)$ is empty and thus its coordination protocol is inconsistent. Any solution is called valid behavior of $\phi$; if no such valid behavior exists, then clearly $\mathcal{L}(\phi)$ is empty.

Typically, we reason about a component specification in isolation, and assume its environment is ideal. This means that whatever solution there exists,

the environment behaves accordingly. Over the course of composition with other components, the environment may be restricted. We say that a component specification becomes inconsistent, if the composition of a coordination protocol and some environment results in an inconsistent coordination protocol, that is, there is no solution that satisfies for both the constraints of the environment and the component specification.

We finally remark the difference between an inconsistent coordination protocol, e.g. $\perp$, and the coordination protocol which contains an assignment to ports consisting only of silent observations, e.g. $\forall t.X(t) = *$. The latter is valid behavior, since there exists a solution, while the former is not, since there is no solution that satisfies $\perp$.

## 3.4 Coordination Games

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. Information flow is controlled by playing a *coordination game*, that we define later. The coordination game resolves the constraints of components with an environment that is out of the control of a component. Suppose the environment initiates an inward flow, then the component may choose to, either, block the inward flow as if it applies back pressure, 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 applies for outward flow.

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 (at least) of a 'null' value $*$ that represents the absence of data. If at some time $*$ occurs in a data stream, we mean to indicate that no information flows at this time.

As a component has an interface, consisting of ports we intend to observe all simultaneously. Intuitively, we consider a snapshot of a component's ports, and call it an *observation*. If an observation contains only $*$, we call it a *silent observation*. Conversely, if an observation contains one value that is not $*$, we call it an *actual observation*. All information captured in the snapshots over time is assumed to be consistent and continuous. A consistent snapshot abstracts the ordering of information flow and is a faithful representation of what actually happened. A trace of snapshots is continuous if it excludes the existence of hidden information flows not captured in snapshots over time.

We say that a port *fires* if there is 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. For example, take any component with two ports, $X$ and $Y$. Here $X$ is an input port, and $Y$ is an output port. We observe it over some period of time. In Table 3.1 we see four observations: the first observation (at $t = 0$) has 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 a step of the coordination game and the result is that the data element $d$ is exchanged through

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

Table 3.1: Examples of observations of ports $X$ and $Y$

port $X$. The last observation (at $t = 3$) has that the data element $d$, which is the same element as previously, flows out of output port $Y$: again a step in the 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: e.g. 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 continuous here means that there is no hidden information flow. For example, in Table 3.1, all information that flows in or out the component between $t = 0$ and $t = 3$ is as shown. Between time $t > 0$ and $t < 3$, the table reports no activity at either port, and thus we may assume that there was no such activity even in between the time steps.

Component behavior is defined in terms of permissible traces of observations. We model an observation by an assignment of ports. We then consider a trace of observations, or a stream of observations. We define coordination games that gives rise to these traces, and give examples of a game play of the previously considered components.

Let $\beta$ be an observation. An observation is an assignment from ports to data values. Note that there is an essential difference between solutions and traces: a solution is an assignment from ports to data streams, whereas a trace is a stream of observations. Intuitively, from the perspective of infinite tables, a trace consists of a stream of rows, where at each row we assign a value for each port. A solution consists of multiple streams, one for each column, that are assigned to ports. Clearly, the two are different representations of an equivalent table.

**Definition 28.** A *game graph* is a graph with configurations as vertices and arcs as edges. A *configuration* is an arbitrary element of a set of configurations. There is a designated *initial configuration* $I$. An *arc* between configurations is a tuple $(C, \beta, C')$, where $C$ and $C'$ are configurations and $\beta$ an assignment. A configuration $C$ is *inconsistent* if there is no $C'$ such that there is an arc from $C$ to $C'$. A *path* is a finite sequence of alternating configurations and assignments. Arcs are denoted $C \xrightarrow{\beta} C'$, and paths are denoted $C_1 \xrightarrow{\beta_1} C_2 \xrightarrow{\beta_2} C_3 \cdots C_{n-1} \xrightarrow{\beta_n} C_n$. Two paths *follow* if the last configuration of the first path is the first configuration of the second path. A path starting with the initial configuration is a *prefix*. A *cycle* is a path such that the first and last configurations are the same. Given a cycle $C_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_n} C_1$, we may *unroll* it by walking the cycle multiple times, e.g. $C_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_n} C_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_n} C_1$. A *trace* is a path formed by a prefix followed by a

cycle, or an infinite sequence of alternating configurations and assignments. We have *inclusion* of a prefix in a trace: if the trace is finite, we must sufficiently unroll the cycle of the trace until the length is larger than the given prefix and then walking over both the given prefix and the trace with unrolled cycle; if the trace is infinite, we can take it immediately. A prefix then is included in a trace if, superimposing the prefix over the trace, each configuration and assignment matches. A *dead-end* is a prefix that is not included in a trace. ■

We assume our game graphs do not contain any junk, that is: every configuration is reachable from the initial configuration. The number of steps that a configuration is reachable from the initial configuration is the round number. A configuration may occur at multiple rounds.

**Definition 29.** A *coordination game* on a game graph is played by two players, an *environment* and an *machine*. In each step, the environment provides a partial assignment, by mapping all input ports. The machine must complete the partial assignment to a solution, by mapping all output ports. The *objective* of the machine is to avoid an inconsistent configuration. ■

The environment is taken as an adversary, in the sense that its behavior is unpredictable. A machine can avoid an inconsistent configuration by considering the dead-ends. If an environment never provides a partial assignment that inevitably leads to an inconsistent configuration, we say the environment is *ideal*.

# Chapter 4

# Components

Now that we have a solid logical foundation, we may define component specifications. These components are closely related to those in the Reo literature, e.g. as found in [10, 23, 33, 49]. The components PUSH, PULL, CONSENSUS, and PROPHET are not considered before in Reo. Moreover, the components we consider here are not exhaustive: others exists.

The format we employ to define components is the following:

| | |
|---|---|
| Component: | *name*(*parameters*) |
| Interface: | *interface* |
| Protocol: | *coordination protocol* |

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 interface $\langle In_1^{\alpha}, \ldots \mid Out_1^{\beta}, \ldots \rangle$ specifies the names of the stream variables and their type: all variables on the left act as inputs, variables on the right act as outputs. The protocol is given as a coordination protocol where free stream variables are as designated by the component definition (see Chapter 3). Free variables that are not in the interface are hidden ports, typically used as memory.

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.

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 absence data.

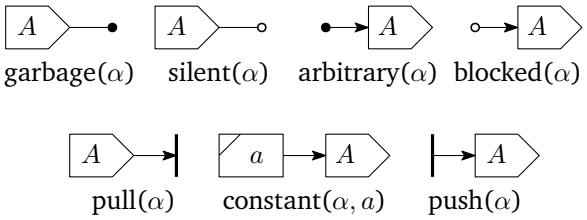



Figure 4.1: Endpoints

## 4.1 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 4.1.

| | |
|---|---|
| Component: | GARBAGE($\alpha$) |
| Interface: | $\langle A^{\alpha} \mid \rangle$ |
| Protocol: | $\forall t.(A(t) = * \vee A(t) \neq *)$ |

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. Its protocol is equivalent to $\top$.

A SILENT never produces any output. The component accepts only streams of observations when port $A$ 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($\alpha$) |
| Interface: | $\langle \mid A^{\alpha} \rangle$ |
| Protocol: | $\forall t.(A(t) = *)$ |

We have dual components, where input and output are swapped but the protocol the same. The dual of GARBAGE is ARBITRARY, and the dual of SILENT is BLOCKED.

| | |
|---|---|
| Component: | ARBITRARY($\alpha$) |
| Interface: | $\langle \mid A^{\alpha} \rangle$ |
| Protocol: | $\forall t.(A(t) = * \vee A(t) \neq *)$ |

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 $A$ 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.

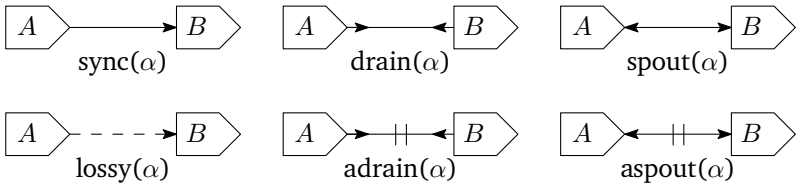



Figure 4.2: Channels

| | |
|---|---|
| Component: | BLOCKED($\alpha$) |
| Interface: | $\langle A^{\alpha} \mid \rangle$ |
| Protocol: | $\forall t.(A(t) = *)$ |

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.

| | |
|---|---|
| Component: | PUSH($\alpha$) |
| Interface: | $\langle \mid A^{\alpha} \rangle$ |
| Protocol: | $\forall t.(\exists s.(A(t+s) \neq *))$ |

| | |
|---|---|
| Component: | PULL($\alpha$) |
| Interface: | $\langle A^{\alpha} \mid \rangle$ |
| Protocol: | $\forall t.(\exists s.(A(t+s) \neq *))$ |

The two components that force progress for its outputs and inputs are PUSH and PULL, by always asserting that the port eventually fires. These components are similar to GARBAGE and ARBITRARY: PUSH generates an arbitrary output; PULL accepts arbitrary input. These component are useful to introduce an inconsistency in case of a deadlock.

Constants push a predefined value precisely once. These components are useful when reasoning about specific elements and constructing testing environments.

| | |
|---|---|
| Component: | constant($\alpha, a$) |
| Interface: | $\langle \mid A^{\alpha} \rangle$ |
| Protocol: | $\exists s.(\forall t.(t < s \vee t > s \rightarrow A(t) = *) \wedge (A(s) = a))$ |

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

## 4.2 Channels

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 two input ports, and channels with two output ports. The channels introduced here are depicted in Figure 4.2.

| $A$ | $B$ |
|---|---|
| $d$ | $d$ |

| $A$ | $B$ |
|---|---|
| $d$ | $*$ |

Table 4.1: Example observations of LOSSY

A SYNCHRONOUS CHANNEL (or SYNC) 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($\alpha$) |
| Interface: | $\langle A^{\alpha} \mid B^{\alpha} \rangle$ |
| Protocol: | $\forall t.(A(t) = B(t))$ |

A LOSSY SYNCHRONOUS CHANNEL (or LOSSY) 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($\alpha$) |
| Interface: | $\langle A^{\alpha} \mid B^{\alpha} \rangle$ |
| 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 4.1. 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 $d$. These two cases correspond to the disjunction in the protocol of LOSSY. Note that the LOSSY component defined here is non-deterministic but not context-sensitive, as described by Jongmans and others in [52].

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 (or DRAIN) loses all its input with the purpose of synchronization: either both input ports pass data instantaneously or both ports are absent of data. Passing data need not be related.

| | |
|---|---|
| Component: | DRAIN($\alpha, \beta$) |
| Interface: | $\langle A^{\alpha}, B^{\beta} \mid \rangle$ |
| Protocol: | $\forall t.((A(t) = * \wedge B(t) = *) \vee (A(t) \neq * \wedge B(t) \neq *))$ |

An ASYNCHRONOUS DRAIN (or ADRAIN) also loses all its input for the purpose of synchronization. At most one input port passes data instantaneously. Passing data is not related.

SYNCHRONOUS DRAINS and ASYNCHRONOUS 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$(\alpha,\beta)$ |
| Interface: | $\langle A^{\alpha}, B^{\beta} \mid \rangle$ |
| 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 (or 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$(\alpha,\beta)$ |
| Interface: | $\langle \mid A^{\alpha}, B^{\beta} \rangle$ |
| Protocol: | $\forall t.((A(t) = * \wedge B(t) = *) \vee (A(t) \neq * \wedge B(t) \neq *))$ |

An ASYNCHRONOUS SPOUT (or ASPOUT) is the dual of an ASYNCHRONOUS DRAIN: it also generates data, but never at the same time at both ports.

| | |
|---|---|
| Component: | ASPOUT$(\alpha,\beta)$ |
| Interface: | $\langle \mid A^{\alpha}, B^{\beta} \rangle$ |
| Protocol: | $\forall t.((A(t) \neq * \rightarrow B(t) = *) \wedge (B(t) \neq * \rightarrow A(t) = *))$ |

## 4.3 Buffers

A channel that deserves special attention is the BUFFER, which transports values non-instantaneously. We consider a BUFFER, and three variants. These components are depicted in Figure 4.3.

These components are stateful: the components have memory that is either empty or full. Memory is a hidden port, that is, it is part of the observation but it cannot 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.

| | |
|---|---|
| Component: | BUFFER$(\alpha)$ |
| Interface: | $\langle A^{\alpha} \mid B^{\alpha} \rangle$ |
| Protocol: | $M^{\alpha}(0) = * \wedge$ |
| $\forall t.(($ | $B(t) = * \quad \wedge M(t) = * \wedge M(t+1) = A(t)) \vee$ |
| | $(A(t) = * \wedge B(t) = * \quad \wedge M(t) \neq * \wedge M(t+1) = M(t)) \vee$ |
| | $(A(t) = * \wedge B(t) = M(t) \wedge M(t) \neq * \wedge M(t+1) = *))$ |

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 BUFFER 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 4.2a: first the input port fires with $d$, while the output port must block. The data element is stored in memory: as long as

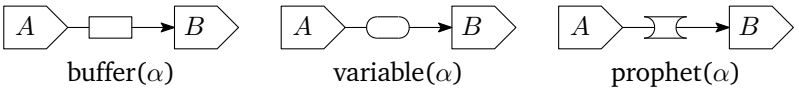



Figure 4.3: BUFFER, VARIABLE, PROPHET.

| $A$ | $M$ | $B$ |
|---|---|---|
| $d$ | $*$ | $*$ |
| $*$ | $d$ | $*$ |
| $*$ | $d$ | $d$ |
| $*$ | $*$ | $*$ |

(a) BUFFER

| $A$ | $B$ |
|---|---|
| $d$ | $*$ |
| $*$ | $d$ |
| $e$ | $*$ |
| $*$ | $e$ |

(b) BUFFER

| $A$ | $B$ |
|---|---|
| $e$ | $*$ |
| $d$ | $*$ |
| $*$ | $d$ |
| $*$ | $d$ |

(c) VARIABLE

| $A$ | $B$ |
|---|---|
| $*$ | $d$ |
| $*$ | $*$ |
| $d$ | $*$ |

(d) PROPHET

Table 4.2: Example observations of BUFFER, VARIABLE and PROPHET.

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 4.2b. Also see Table 3.1 for an earlier example.

**Lemma 30.** *Stream $M$ of a BUFFER is uniquely determined by the streams $A$ and $B$.*

*Proof.* Given two streams $A$ and $B$, and suppose there are two streams $M$ and $M'$ such that there is some $t$ where $M(t) \neq M'(t)$, such that the coordination protocol is satisfiable. We separate regions of streams $M$ and $M'$ into three types: initial, empty, full. The initial region is from $t = 0$ up to some time where the input fires fires. Both streams must coincide, because the second rule applies here. The full region is between the moment the input fires until the first moment the output fires: again, both streams must coincide because of the second rule. The empty region is between the moment the output fires until the first moment the input fires: again, both streams must coincide because of the second rule. Thus there does not exists any $t$ in which $M$ and $M'$ are different. □

The first variant of BUFFER is the VARIABLE. 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 4.2c: first the input fires with $e$. Then the input fires $d$, which overwrites the previous memory. Then the output fires $d$ but it does not erase the memory. Hence, the next turn, the output port fires again with $d$.

If a VARIABLE is full, any input element overwrites the existing value in memory. Additionally, if a VARIABLE is full, then it always remains full. If a VARIABLE is full, both input and output port may fire together, but input is never instantaneously transported to its output. A variable can be built out of other primitive components, as is done similar to [8]. The derivation given there is different, since it does not allow input and output to fire together. Here, we do allow that the input and output fire together.

| | | |
|---|---|---|
| Component: | VARIABLE($\alpha$) | |
| Interface: | $\langle A^\alpha \mid B^\alpha \rangle$ | |
| Protocol: | $M^\alpha(0) = * \wedge$ | |
| $\forall t.(($ | $B(t) = * \quad \wedge M(t) = * \wedge M(t+1) = A(t))$ | $\vee$ |
| | $(A(t) = * \wedge B(t) = * \quad \wedge M(t) \neq * \wedge M(t+1) = M(t))$ | $\vee$ |
| | $(A(t) = * \wedge B(t) = M(t) \wedge M(t) \neq * \wedge M(t+1) = M(t))$ | $\vee$ (!) |
| | $(A(t) \neq * \wedge B(t) = * \quad \wedge M(t) \neq * \wedge M(t+1) = A(t))$ | (!) |
| | $(A(t) \neq * \wedge B(t) = M(t) \wedge M(t) \neq * \wedge M(t+1) = A(t)))$ | (!) |

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($\alpha$) | |
| Interface: | $\langle A^\alpha \mid B^\alpha \rangle$ | |
| Protocol: | $M^\alpha(0) = * \wedge$ | |
| $\forall t.(($ | $A(t) = * \quad \wedge M(t) = * \wedge M(t+1) = B(t))$ | $\vee$ |
| | $(B(t) = * \wedge A(t) = * \quad \wedge M(t) \neq * \wedge M(t+1) = M(t))$ | $\vee$ |
| | $(B(t) = * \wedge A(t) = M(t) \wedge M(t) \neq * \wedge M(t+1) = *))$ | |

PROPHETS first fire their output port by speculatively generating 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 arguing 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 by speculative execution and backtracking if the wrong element was chosen.

BUFFER and PROPHET are asynchronous, but VARIABLE is not synchronous (input and output may not fire together) and not asynchronous (input and output may fire together). BUFFER 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 and VARIABLE are causal channels, and PROPHET is an acausal component.

## 4.4 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 4.4).

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.

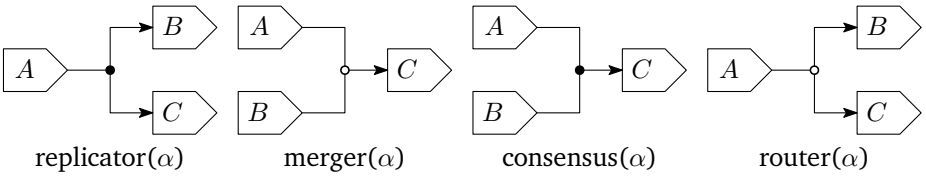



Figure 4.4: Nodes

| | |
|---|---|
| Component: | REPLICATOR($\alpha$) |
| Interface: | $\langle A^\alpha \mid B^\alpha, C^\alpha \rangle$ |
| 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($\alpha$) |
| Interface: | $\langle A^\alpha, B^\alpha \mid C^\alpha \rangle$ |
| 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($\alpha$) |
| Interface: | $\langle A^\alpha, B^\alpha \mid C^\alpha \rangle$ |
| 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($\alpha$) |
| Interface: | $\langle A^\alpha \mid B^\alpha, C^\alpha \rangle$ |
| 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

Table 4.3 is an overview of the components given here.

| | # | (I/O) | term | sync | inst | lin | caus |
|---|---|---|---|---|---|---|---|
| GARBAGE | 1 | (1/0) | | – | | | *c* |
| SILENT | 1 | (0/1) | *t* | – | | ✓ | *c* |
| ARBITRARY | 1 | (0/1) | | – | | | |
| BLOCKED | 1 | (1/0) | *t* | – | | ✓ | *c* |
| PULL | 1 | (1/0) | *p* | – | ✗ | | *c* |
| PUSH | 1 | (0/1) | *p* | – | ✗ | | |
| SYNC | 2 | (1/1) | | *s* | | ✓ | *c* |
| LOSSY | 2 | (1/1) | | | | | *c* |
| DRAIN | 2 | (2/0) | | *s* | | | *c* |
| ADRAIN | 2 | (2/0) | | *a* | | | *c* |
| SPOUT | 2 | (0/2) | | *s* | | | |
| ASPOUT | 2 | (0/2) | | *a* | | | |
| BUFFER | 2 | (1/1) | | *a* | ✗ | ✓ | *c* |
| VARIABLE | 2 | (1/1) | | | ✗ | | *c* |
| PROPHET | 2 | (1/1) | | *a* | ✗ | ✓ | *a* |
| REPLICATOR | 3 | (1/2) | | *s* | | | *c* |
| MERGER | 3 | (2/1) | | | | ✓ | *c* |
| CONSENSUS | 3 | (2/1) | | *s* | | | *c* |
| ROUTER | 3 | (1/2) | | | | ✓ | *c* |

Table 4.3: A non-exhaustive table of primitive components. The #-column shows the components arity (number of ports), and the **(I/O)**-column shows the numbero of input ports and output ports. The **term**-column shows progress (*p*) or termination (*t*). The **sync**-column shows synchronicity (*s*) or asynchronicity (*a*): one-port components are trivial (–). The **inst**-column displays which primitives are non-instantaneous (✗). The **lin**-column displays the property of linearity (✓). The **caus**-column display the property of causality (*c*) or acausality (*a*).

*This page is intentionally left blank.*

# Chapter 5

# Properties

In [7], Arbab begins with the claim that "the most challenging aspect of concurrency involves the study of interaction and its properties". What are these properties and why are these interesting in the first place? This deserves some discussion.

**Independence** or delay insensitivity or stuttering of a component indicates that it can cooperate with other components. A component is delay insensitive if it can be 'paused' and 'resumed'. A component is independent if, moreover, it has no internal delay. Independence and delay insensitivity are essential for composition: consider, for example, combining a slow-running component with a fast-running component. If these components need to communicate, then the fast-running component needs to slow down to match the slow-running component. Delay insensitivity captures the property that a component can be arbitrarily slowed down. Independence moreover captures the property that a component can be arbitrarily sped up. Delay insensitivity is preserved under composition.

The property of delay insensitivity or the stronger property of independence are also known as *stuttering*, cf. Chapter 7.8 of Baier and Katoen's *Principles of Model Checking* [12]. This property is important for defining behavioral equivalence of components, as understood by stuttering bisimulation relations. For example, Manolios employs this property to show the correctness of pipelined processors [62].

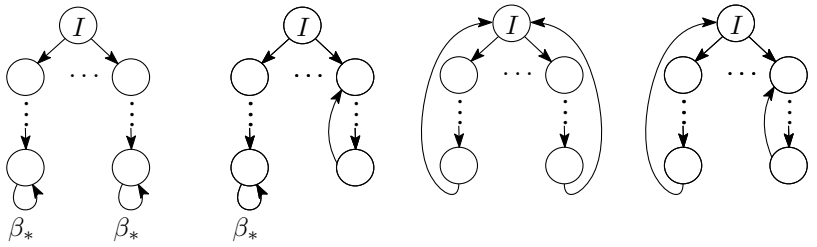
Another similar notion of equivalence is branching bisimulation with explicit divergences [36]. In our formalism stuttering is modeled by a row of stars, also called the silent observation. Independence can be understood by the closure condition that states that a coordination protocol allows the insertion and removal of such silent observations under certain conditions. Such conditions prevent the possibility to insert all silent transitions after arbitrary prefixes, which allows us to distinguish between acceptable behaviors which terminate from acceptable behaviors that do not: this makes divergence explicit.

**Synchronous** components model atomicity of their observations: either all ports fire or none of them fires. A non-synchronous component violates this constraint, e.g. allows two out of three ports firing at the same time. An asynchronous component has at most one port firing exclusively and other ports do not fire. These property are checked for a subset of ports of a component; the synchronous property on sets is downward closed, i.e. if any set of ports is synchronous, then so is any subset. Synchronicity is preserved under composition.

The distinction synchronous and asynchronous is useful to make in the context of asynchronous processor architectures [42]. A synchronous circuit requires a clock signal to propagate through the circuit, wasting energy and limiting the clock speed to the slowest component that receives the same clock signal. Even if a circuit consists of different synchronous regions, communicating asynchronously among such regions is highly efficient [74].

**Progress** can be understood more clearly by studying multiple properties. Productivity means that there is an eventual actual observation, and non-productive means there are always silent observations. Moreover, we distinguish four properties: termination and deadlock, livelock and perpetuity.

An execution is terminating if every computation reaches a point where it remains non-productive. An execution has a deadlock situation if there is a possibility of a computation that reaches a point where it remains non-productive. An execution is perpetual if all computations always eventually leads back to the initial configuration. An execution has a livelock situation if there exists a possibility in which the computation no longer leads back to the initial configuration, resulting in a defect. In terms of coordination game graphs, we can roughly understand these four properties as depicted below:




Progress is a useful property to study to understand the productivity of a system. Typically, we take deadlock and livelock as negative or useless properties we wish to avoid, although sometimes it is useful to detect [26]. Termination is useful to demonstrate that a machine always halts in a finite number of steps, after having performed a particular task. Perpetual computation[1] is useful to demonstrate that a machine always has potential to perform all its behavior, and thus does not contain any defects in which only limited parts of all its behavior are performed.

**Instantaneousness**: a component is instantaneous (or stateless) if its observations all happen in a single instant. An instantaneous component does not relate inputs and outputs over time. In terms of coordination game graphs, instantaneous components have a single configuration. A component that is non-instantaneous is called stateful. Instantaneousness is by definition preserved under parallel composition and identification.

Instantaneous components do not require memory. This is useful when composing instantaneous components: as an optimization, one could compress an instantaneous component into a single component accepting equivalent behavior. In addition, we make the case for modeling all components as instantaneous components that dynamically reconfigure after taking a step, similar to dynamic

[1] "Perpetual computation" is by some [78] understood similar to "perpetual motion" and deemed impossible. Here, by perpetual computation we mean the fact that a process can always reproduce all its behavior, even though that process has effects on its environment.

reconfiguration using rewriting techniques by Krause and others [58], and graph rewriting as done by Koehler and others [55]. It is also possible to consider a distributed network that dynamically reconfigures itself, and communicates behavior as instantaneous constraints, as done by Proença and others [69]. Bruni and others have focused on instantaneous components, and their algebraic formalism can model all instantaneous components [23].

**Linearity** means that input is never discarded or duplicated, and output never appears out of nowhere. Linearity holds if every output of a component can be traced back to precisely one of its inputs at a unique time, and every one of its inputs is uniquely related to an output. Intuitively, duplication in space or over time is prohibited.

Linearity is useful to consider, as it has a close connection to reversibility of computation [77, 78]. In distributed computing, taking a snapshot of a system is useful to detect a stable state, such as deadlock and termination [59]. Snapshots can be used to revert back to a previous state, to perform distributed error recovery [75]. However, algorithms for computing a snapshot incur space overhead in storing previous configurations. If a component is linear, it admits a reversed computation that allows for the recovery of an original configuration without storing it explicitly. This property is useful to understand when considering optimizations of component compilers.

**Causality**: a causal component relates each of its outputs to inputs that have happened in the past. Causality implies a flow from input to output. A non-causal component violates this constraint, e.g. by generating arbitrary output without any relation to its input. An acausal component has all its output in the past related to future input, as if its output is a true prediction of what input will happen in the future. Intuitively, acausality is related to speculative execution. Causality or acausality do not imply linearity: a component that duplicates elements in space or over time is not linear but may still be causal or acausal.

Causality allows one to "understand" a system, in terms of how input *leads to* an output, or how an output *requires* some input [22]. Bergner and others have studied causality of components, and their information flow, in a similar way as we have done here [21, 17]: introducing instances, components and connections between interfaces. There, components are modeled using *timed streams* over its instances, components, and connections: this is essentially different than our work, where in our semantics we assume component instances to be static. They define causality, or *time-guardedness*, in a similar way what we do here.

We thus now define and analyze these properties of coordination protocols. We give examples and intuition for these properties based on examples that are related to the running example introduced in Section 1.2.

The general approach is that of thesis, antithesis and synthesis. We study the property positively and we study it negatively. This leads to a number of interesting varieties of (opposing) properties. Properties could be combined to form more complex properties. The properties listed here are by no means exhaustive.

## 5.1 Independence

Independence of a component is intuitively the possibility to stretch and shrink observations over time. Delay insensitivity of a component is the possibility to stretch observations over time, but not shrinking. Let $\phi(U)$ be a component specification where $\phi$ is a coordination protocol and $U$ an interface. Without loss of generality, we assume all ports of $U$ occur precisely as the free variables of $\phi$. From the semantic viewpoint, $\mathcal{L}(\phi(U))$ is the set of assignments of streams. We can understand independence as a closure condition on $\mathcal{L}(\phi(U))$.

Let $\beta \in \mathcal{L}(\phi(U))$ be a solution. Recall that a solution is a set of (infinite) tables of observations, and if a solution is in $\mathcal{L}(\phi(U))$ we call it acceptable. We define two operations on such tables: adding a row consisting of $*$ only, and removing a row consisting of $*$ only. We restrict these operations to apply only to certain rows: adding may happen only before a row that actually contains data, and rows that contain nothing but $*$ may be removed only.

Let $t$ be a row containing at least one non-$*$ value. We insert a new row of all $*$ symbols before $t$. The result of adding this row in $\beta$ is denoted $\mathrm{add}(\beta,t)$, which itself is a solution. Adding a row $\mathrm{add}(\beta,t)$ is defined only if $\beta(Y)(t) \neq *$ for some $Y$:

$$\mathrm{add}(\beta,t)(X)(s) = \begin{cases} \beta(X)(s) & \text{if } s < t \\ * & \text{if } s = t \\ \beta(X)(s-1) & \text{if } s > t \end{cases}$$

Let $t$ be a row, for which all values are $*$. The result of removing a row in $\beta$ is denoted $\mathrm{remove}(\beta,t)$, defined only if $\beta(Y)(t) = *$ for all $Y$:

$$\mathrm{remove}(\beta,t)(X)(s) = \begin{cases} \beta(X)(s) & \text{if } s < t \\ \beta(X)(s+1) & \text{if } s \geq t \end{cases}$$

Independence is the closure of insertion and removal of rows that consist only of $*$. Delay-insensitivity is the closure of insertion of rows that consist only of $*$. More precisely,

**Definition 31.** A component specification $\phi(U)$ is *delay insensitive* if for each solution $\beta \in \mathcal{L}(\phi(U))$ and for each time $t$, we have that $\mathrm{add}(\beta,t) \in \mathcal{L}(\phi(U))$ if $\mathrm{add}(\beta,t)$ is defined. Moreover, a component specification $\phi(U)$ is *independent* if $\phi(U)$ is delay insensitive and for each time $t$, $\mathrm{remove}(\beta,t) \in \mathcal{L}(\phi)$ if $\mathrm{remove}(\beta,t)$ is defined.

Clearly, independence of a component specification implies delay insensitivity. The converse need not hold.

**Lemma 32.** *Delay insensitivity is preserved by composition: if $\phi(U)$ is delay insensitive and $\psi(V)$ is delay insensitive then $\phi(U) \parallel \psi(V)$ is delay insensitive; if $\phi(U)$ is delay insensitive and $A \in U$ is an input port and $B \in U$ is an output port then $(\phi(U))^A_B$ is delay insensitive.*

*Proof.* The first implication holds. Suppose we have two delay insensitive components. The resulting behavior is delay insensitive: we may insert rows with all $*$ before a row with data, by doing so in the two underlying tables, by considering two cases:

In the first case, the row below which we insert consists of data at ports at both $U$ and $V$. In that case, we insert the row with $*$ in both composed components, which is possible by assumption.

In the second case, the row below which we insert consists of data at ports at only one of $U$ or $V$. Suppose that data is observed at a port in $U$, and all ports in $V$ are silent. There are two possibilities: $V$ has terminated and never fires again. In that case, we can simply apply the assumption and insert a $*$ row for ports at $U$. If $V$ has not terminated, there exists some first row where there is data. We apply the assumption for $U$ at the current row; we apply the assumption for $V$ at the first row in the future where there is data.

The second implication holds. Suppose we have a delay insensitive component with two ports $A$ and $B$. Identification of ports removes solutions where $A$ and $B$ are not equal; and identification removes both ports from the interface. Hence any solution only ranges over the other ports. The set of solutions is closed under addition; given a row where there is some data at a port, by assumption, we can insert an all $*$ row before it. Since this assigns both $A$ and $B$ the same value, this solution is not removed by identification. □

**Proposition 33.** *Independence is not preserved by composition.*

*Proof sketch.* Consider a counter-example: the composition of two BUFFERS, where the output of one BUFFER is identified with the input of the other. Suppose the the input port of the first BUFFER fires; it transports the data into the next row and the output of the first BUFFER fires. Then the input of the second BUFFER fires in the next row; it transports the data into the second next row. Thus, in the composition, between the input event and a corresponding output event, there is at least one row consisting only of $*$ that is necessary and cannot be removed. □

This proposition only holds due to the implicit hiding of identification. Independence is preserved when identified ports are not hidden. To identify two ports $A$ and $B$ without hiding, one can replicate the input port $A$ using a REPLICATOR, where one of its outputs is identified to $B$ and the other output is exposed as a non-hidden port. However, by our choice of composition to be either parallel composition or identification, we do not have compositionality of independence.

**Definition 34.** Given a component specification $\phi(U)$, we define the properties:

$$
\begin{aligned}
\mathbf{delay}(\phi) \coloneqq \forall \vec{X} : U.(\phi(\vec{X}) \rightarrow \\
\forall t.(\bigvee_{i \in U} X_i(t) \neq * \rightarrow \\
\exists \vec{Y} : U.(\phi(\vec{Y}) \wedge \bigwedge_{i \in U} Y_i(t) = * \wedge \\
\forall s.(s < t \rightarrow \bigwedge_{i \in U} Y_i(s) = X_i(s)) \wedge \\
\forall s.(s > t \rightarrow \bigwedge_{i \in U} Y_i(s) = X_i(s-1)) \\
) \quad ) \qquad ) \\
\mathbf{indep}(\phi) \coloneqq \mathbf{delay}(\phi) \wedge \forall \vec{X} : U.(\phi(\vec{X}) \rightarrow \\
\forall t.(\bigvee_{i \in U} X_i(t) = * \rightarrow \\
\exists \vec{Y} : U.(\phi(\vec{Y}) \wedge \\
\forall s.(s < t \rightarrow \bigwedge_{i \in U} Y_i(s) = X_i(s)) \wedge \\
\forall s.(s \geq t \rightarrow \bigwedge_{i \in U} Y_i(s) = X_i(s+1)) \\
) \quad ) \qquad )
\end{aligned}
$$

**Lemma 35.** *(Correctness) Let $\phi(U)$ be a component specification. $\phi(U)$ is delay insensitive if and only if $\vDash \mathbf{delay}(\phi)$; $\phi(U)$ is independent if and only if $\vDash \mathbf{indep}(\phi)$.*

*Proof.* Trivial and follows from definition. The proof below highlights the connection between the formalism and the semantic definition.

(Delay insensitivity $\Rightarrow$) Assume $\phi(U)$ is delay insensitive. Then for each $\beta \in \mathcal{L}(\phi)$ and time $t$, $\mathrm{add}(\beta,t) \in \mathcal{L}(\phi)$ if $\mathrm{add}(\beta,t)$ is defined. We show that $\mathbf{delay}(\phi)$ is valid: let $\vec{X} : U$ denote an arbitrary stream such that $\phi(\vec{X})$, then $\vec{X}$ and $\beta$ correspond. Let $t$ be a time such that there is some $i \in U$ such that $X_i$ fires. Now $\mathrm{add}(\beta,t)$ is defined; take this stream as witness for $\exists \vec{Y}$. It is easily verified that the three conditions correspond to the definition of add.

(Delay insensitivity $\Leftarrow$) Assume that $\mathbf{delay}(\phi)$ is valid. We show that $\phi(U)$ is delay insensitive: let $\beta \in \mathcal{L}(\phi)$ be an arbitrary solution, and $t$ an arbitrary time such that at least one port fires. To establish that $\mathrm{add}(\beta,t) \in \mathcal{L}(\phi)$, let $\vec{X}$ correspond to the solution $\beta$, and $\vec{Y}$ be the witness. It is easily verified that the witness $\vec{Y}$ corresponds to the solution $\mathrm{add}(\beta,t)$.

Independence is proved in a similar way. □

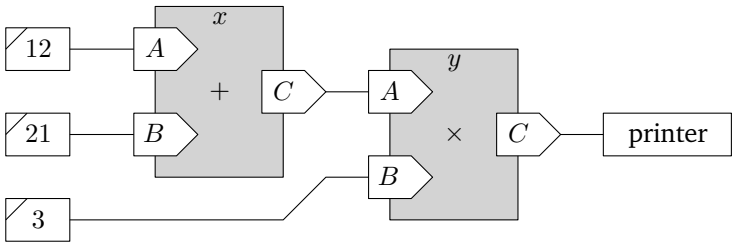
A real-time clock is a component on which the value at time $t$ depends on $t$.

**Proposition 36.** *Delay insensitivity prevents real-time clocks.*

*Proof sketch.* Suppose there exists a component that for each row $t$ outputs precisely the row index $t$. Then it cannot be delay insensitive since we can no longer stretch the table: the stretching implies that a clock value at row $t$ now potentially is observed at $t+s$ for some $s > 0$. That contradicts that each observation reflects the row index. □

**Corollary 37.** *Independence prevents real-time clocks.*

**Example 38.** An example of two independent components are ADDITION and MULTIPLICATION, as depicted below.




Delay insensitivity means that MULTIPLICATION can be delayed until ADDITION is done, and vice versa. Independence moreover means that we do not have necessary rows consisting only of $*$. As a concrete example, suppose that these tables are acceptable for ADDITION (on the left) and MULTIPLICATION (on the right):

| $t$ | $x.A$ | $x.B$ | $x.C$ |
|---|---|---|---|
| 0 | $*$ | $*$ | $*$ |
| 1 | 12 | 21 | $*$ |
| 2 | $*$ | $*$ | 33 |
| 3 | $*$ | $*$ | $*$ |
| ⋮ | ⋮ | ⋮ | ⋮ |

| $t$ | $y.A$ | $y.B$ | $y.C$ |
|---|---|---|---|
| 0 | 33 | 3 | $*$ |
| 1 | $*$ | $*$ | $*$ |
| 2 | $*$ | $*$ | 99 |
| 3 | $*$ | $*$ | $*$ |
| ⋮ | ⋮ | ⋮ | ⋮ |

For the first table, we know it also has another acceptable table because the component is independent: remove row 0. For the second table, we remove rows 1 and 3 and insert a row before 0. Since $x.C$ and $y.A$ are identified, the tables match up and we deduce the result of their composition:

| $t$ | $x.A$ | $x.B$ | $x.C$ |
|---|---|---|---|
| 0 | 12 | 21 | $*$ |
| 1 | $*$ | $*$ | 33 |
| 2 | $*$ | $*$ | $*$ |
| ⋮ | ⋮ | ⋮ | ⋮ |

| $t$ | $y.A$ | $y.B$ | $y.C$ |
|---|---|---|---|
| 0 | $*$ | $*$ | $*$ |
| 1 | 33 | 3 | $*$ |
| 2 | $*$ | $*$ | 99 |
| ⋮ | ⋮ | ⋮ | ⋮ |

Note how it is not possible to remove a row with data, since remove is not defined for rows with data. Similarly, for the addition component, we cannot insert a row before row 2, because add is not defined for a row without data. ■

## 5.2 Synchronicity

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).

A synchronous component relates its ports such that either all port activity happens at the same time, or nothing happens at all. An asynchronous components also relates its ports by stating that port activity happens at only one port at a time excluding at the same time any other port activity.

Synchronicity is defined on a set of ports $P = \{X_1, \ldots, X_n\}$ and component specifications $\phi(U)$. $P$ is *synchronous* if always either all ports fire, or no port fires. This is expressed by the following formulas:

$$\mathbf{sync}(P) \coloneqq \forall t.(\bigwedge_{i \in P} X_i(t) = * \vee \bigwedge_{i \in P} X_i(t) \neq *)$$

$$\mathbf{sync}(\phi) \coloneqq \forall \vec{X} : U.(\phi(\vec{X}) \rightarrow \mathbf{sync}(\vec{X}))$$

A set of ports is non-synchronous if the synchronous property does not hold. For example, a component with four ports where two ports fire together. The property of being not synchronous and the property of asynchronous are different: we have that $P$ is *asynchronous* if always at most one port fires, or no ports fire. This is expressed by the following formulas:

$$\mathbf{async}(P) \coloneqq \forall t. \bigwedge_{i \in P} (X_i(t) \neq * \rightarrow \bigwedge_{j \in P \setminus \{i\}} X_j(t) = *)$$

$$\mathbf{async}(\phi) \coloneqq \forall \vec{X} : U.(\phi(\vec{X}) \rightarrow \mathbf{async}(\vec{X}))$$

Each row corresponds to a port that fires and implies that all other ports must be silent. We have that $X_1(t) = * \wedge \cdots \wedge X_n(t) = *$ is admitted, if all antecedents are false.

**Definition 39.** A component specification $\phi(U)$ is *synchronous* if $\vDash \mathbf{sync}(\phi)$; it is *asynchronous* if $\vDash \mathbf{async}(\phi)$.

**Proposition 40.** *The singleton port set is both synchronous and asynchronous.*

*Proof.* Let $X$ be the single port. The formulas $\forall t.(X(t) = * \vee X(t) \neq *)$ for synchronicity and $\forall t.(X(t) \neq * \rightarrow \top)$ for asynchronicity are tautological. □

**Proposition 41.** *If a set of ports P is synchronous, then any of its non-empty subsets is synchronous too.*

*Proof.* The singleton set is already covered in Proposition 40. Now remove some port $X \in P$, then we show that $P \setminus \{X\}$ is synchronous. Suppose a port in $P \setminus \{X\}$ fires, then by synchronicity of $P$, all ports must fire, hence all ports in $P \setminus \{X\}$ also fire. □

**Proposition 42.** *If a set of more than one port is both synchronous and asynchronous, then its ports never fire.*

*Proof.* Suppose one fires, then all others must fire (synchronous) and all others must not fire (asynchronous): since this is inconsistent, never any port fires. □

**Example 43.** By definition the SYNCHRONOUS DRAIN and the ASYNCHRONOUS DRAIN are synchronous and asynchronous components, respectively. The SYNCHRONOUS DRAIN either fires both ports at the same time, or fires no port. The ASYNCHRONOUS DRAIN fires at most one port at a time. ■

**Example 44.** An example of an asynchronous component is a BUFFER. A BUFFER never fires both its input and output ports. ■

**Proposition 45.** *Let $\phi(U)$ and $\psi(V)$ be two component specifications that are synchronous, and let $A \in U$ and $B \in V$. The component specification $(\phi(U) \parallel \psi(V))^A_B$ is synchronous.*

*Proof sketch.* By linking two components using an identification, the resulting coordination protocol shares at least one identical hidden port, since $A(t) = B(t)$ for all $t$, and $A$ and $B$ are no longer part of the interface after identification. If a shared port fires, then all other ports of the composition must fire since both components are synchronous. If another port of one of the components fires, then the shared port must also fire since both components are synchronous, which means all ports fire. Therefore, composition with identification preserves synchronization. □

Identification is essential, since otherwise the composed components may fire independently: two synchronous components that are delay insensitive, for example, are not synchronous if composed only in parallel. When one component fires, the other component may be delayed and the set of ports of both components are not all firing together.

A component can have different synchronization characteristics with respect to different subsets of its ports.

**Example 46.** The ROUTER component has three ports. We study the synchronization characteristics of each subset.

| | |
|---|---|
| Component: | ROUTER($\alpha$) |
| Interface: | $\langle A^\alpha \mid B^\alpha, C^\alpha \rangle$ |
| Protocol: | $\forall t.((A(t) = B(t) \wedge C(t) = *) \vee$ |
| | $(A(t) = C(t) \wedge B(t) = *))$ |

Intuitively, a ROUTER takes data from its input and brings it to exactly one output port; the other output port remains silent. If we study $\{B,C\}$, then we see that they never fire together; so $\{B,C\}$ is asynchronous. If we study $\{A,B\}$, then the firing of $A$ does not necessarily imply the firing of $B$, since either $B$ or $C$ fires. Hence $\{A,B\}$ is neither synchronous, nor asynchronous. ■

**Example 47.** The CONSENSUS and REPLICATOR components have three ports. They have the same coordination protocol:

| | |
|---|---|
| Component: | REPLICATOR($\alpha$) |
| Interface: | $\langle A^\alpha \mid B^\alpha, C^\alpha \rangle$ |
| Protocol: | $\forall t.(A(t) = B(t) \wedge A(t) = C(t))$ |

The set $\{A,B,C\}$ is synchronous: if one port fires, then all other ports must fire. The subsets $\{A,B\}$, $\{A,C\}$, and $\{B,C\}$ are also synchronous, for the same reason. The singleton sets are trivially synchronous. The REPLICATOR and CONSENSUS in addition require that an observation must have the same data element for all ports.

## 5.3 Deadlock and Livelock

We guide our definitions with terminology from concurrent and distributed systems [37]. Let $\phi(U)$ be a component specification. The set of solutions $\mathcal{L}(\phi)$ is called an *execution*. We call a single solution $\beta \in \mathcal{L}(\phi)$ a *computation*.

We first define properties on a set of ports $P = \{X_1, \ldots, X_n\}$, in a similar way as done for the (a)synchronous property. Productivity, or non-productivity, specifies that eventually an actual observation, or only silent observations, is accepted, respectively. That $P$ is *productive* or *non-productive* can be expressed by the following formulas:

$$\begin{aligned} \mathbf{prod}(P) &:= \exists s.\ \bigvee_{1 \le i \le n} X_i(s) \neq * \\ \neg\mathbf{prod}(P) &= \forall s.\ \bigwedge_{1 \le i \le n} X_i(s) = * \end{aligned}$$

A set of ports $P$ is *terminating* if there is some point where the ports are non-productive, i.e. there is some point from which onward the ports no longer fire. This property is expressed by the following equivalent formulas:

$$\begin{aligned} \mathbf{term}(P) &:= \exists t. \forall s.\ \bigwedge_{1 \le i \le n} X_i(t+s) = * \\ \mathbf{term}(P) &= \exists t. \neg\mathbf{prod}(P^{(t)}) \end{aligned}$$

where we understand $t$ to be the maximum number of steps the component can take, after which the component becomes non-productive. Here $P^{(t)}$ is the set $\{X_1^{(t)}, \ldots, X_n^{(t)}\}$.

Conversely, the set of ports $P$ has *progress* if always there is a point in which one of its ports fire, i.e. at any time the ports are productive. This property is expressed by the following equivalent formulas:

$$\begin{aligned} \neg\mathbf{term}(P) &= \forall t. \exists s.\ \bigvee_{1 \le i \le n} X_i(t+s) \neq * \\ \neg\mathbf{term}(P) &= \forall t. \mathbf{prod}(P^{(t)}) \end{aligned}$$

A deadlock situation occurs if a computation eventually reaches a point where it remains non-productive. Using our definition of terminating ports, we can formulate the following more complex properties.

**Definition 48.** Let $\phi(U)$ be a component specification. We define the following:

$$\begin{aligned} \mathbf{deadlock}(\phi) &:= \exists \vec{X} : U.(\phi(\vec{X}) \wedge \mathbf{term}(\vec{X})) \\ \mathbf{term}(\phi) &:= \forall \vec{X} : U.(\phi(\vec{X}) \rightarrow \mathbf{term}(\vec{X})) \\ \neg\mathbf{term}(\phi) &= \exists \vec{X} : U.(\phi(\vec{X}) \wedge \neg\mathbf{term}(\vec{X})) \\ \neg\mathbf{deadlock}(\phi) &= \forall \vec{X} : U.(\phi(\vec{X}) \rightarrow \neg\mathbf{term}(\vec{X})) \end{aligned}$$

We may understand these properties as follows. An execution:

- *contains a deadlock* if at least one computation is terminating,
- *terminates* if all computations are terminating,

- is *non-terminating* if at least one computation is not terminating,
- is *deadlock-free* if all computations are not terminating.

In this definition, we have implicitly converted the vector of variables $\vec{X}$ corresponding to the interface $U$ to a set of ports $\{X_1, \ldots, X_n\}$.

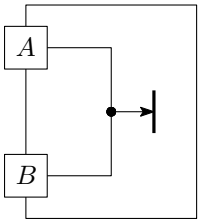
**Example 49.** A component that always progresses by definition is PULL.

| | |
|---|---|
| Component: | $\textsc{pull}(\alpha)$ |
| Interface: | $\langle A^{\alpha} \mid \rangle$ |
| Protocol: | $\forall t.(\exists s.(A(t+s) \neq *))$ |

■

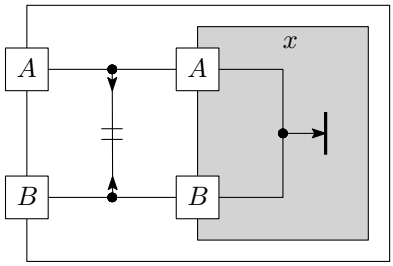
The PULL of a port has effect throughout a composition, similar to synchronization. For example, pulling from a CONSENSUS output means pulling from both its input ports, and pulling from a BUFFER output means pulling from its input. However, progress is not necessarily compositional.

**Example 50.** We consider the following construction:




Here ports $A$ and $B$ are linked together using a CONSENSUS, which is linked to a PULL. Suppose that at some point $A$ no longer can fire with data, i.e. $\exists t.\forall s.A(t+s) = *$. Then the output of the CONSENSUS also can no longer fire, since it is synchronous with $A$. This results in a contradiction when considering also the protocol of the PULL, which states that always eventually it must fire. Hence, we must have that both $A$ and $B$ are productive.

Now consider a more complicated construction, where $x$ is our previous one:




We have two ports on the left; two REPLICATORS that combine with an ASYNCHRONOUS DRAIN; and then the previous construction of a CONSENSUS and a PULL. We already argued before that $x.A$ and $x.B$ of the previous construction have to be productive. By the CONSENSUS $x.A$ and $x.B$ must fire together. However, we must also respect the property of ASYNCHRONOUS DRAIN: either the top fires, or the bottom fires, or both are silent. Thus, $x.A$ and $x.B$ are always silent.

This is in contradiction with PULL, which prohibits such termination. Hence, this component is inconsistent: it cannot exists. It does not have any valid behavior at all, not even always silent behavior.

Observe that replacing the PULL component above with a GARBAGE results in a component which has valid behavior: in this case, the component is always silent (ports $A$ and $B$ never fire). This is different than not having any behavior at all. ■

Progress on a set of ports is different than progress on those ports individually. Consider that progress on a set of ports allows one of those ports to always eventually fire, while the rest remain silent. This is not allowed when each port has progress individually. However, progress on ports individually implies progress on the set of those ports. If one port always eventually fires, then it is also the case that a larger set containing that port still has progress.

We finally consider an alternative specification of the BUFFER by this coordination protocol as we have seen in Example 25:

$$\begin{aligned}
&\forall t.((B(t) = * \wedge A(t) = *) \vee \\
&\quad (B(t) = * \wedge \exists j.(t < j \wedge A(j) = * \wedge B(j) = A(t) \wedge \\
&\qquad\qquad \forall i.(t < i \wedge i < j \rightarrow A(i) = * \wedge B(i) = *))) \vee \\
&\quad (A(t) = * \wedge \exists j.(j < t \wedge A(j) = B(t) \wedge B(j) = * \wedge \\
&\qquad\qquad \forall i.(j < i \wedge i < t \rightarrow A(i) = * \wedge B(i) = *))))
\end{aligned} \tag{5.1}$$

In here, we no longer have an occurrence of $M$, as in the standard definition in Section 4.3, repeated below:

$$\begin{aligned}
M(0) = * \wedge \forall t.(( \quad & B(t) = * \quad \wedge M(t) = * \wedge M(t+1) = A(t)) \vee \\
(A(t) = * \wedge B(t) = * \quad & \wedge M(t) \neq * \wedge M(t+1) = M(t)) \vee \\
(A(t) = * \wedge B(t) = M(t) \wedge & M(t) \neq * \wedge M(t+1) = *))
\end{aligned} \tag{5.2}$$

The alternative specification (5.1) and the standard specification (5.2) are different: this is demonstrated by the following counter-example.

**Proposition 51.** *The two specifications of* BUFFER *are not equivalent.*

*Proof.* We work towards contradiction. Assume the two specifications are equivalent, then for all solutions, the original specification must be true if and only if the alternative specification is true. We construct a scenario in which only a single input event happens, but never any output event. Let $A \mapsto \sigma, B \mapsto \tau, M \mapsto \gamma$ be a solution, where $\sigma(0) \neq *$ and $\sigma(x) = *$ for all $x > 0$, and $\tau(x) = *$ for all $x$. The stream $\gamma$ is uniquely determined by $\sigma$ and $\tau$ by Lemma 30, and is the stream $\gamma(0) = *$ and $\gamma(x) = \sigma(0)$ for all $x > 0$. We now have that the original specification is true (the first row is verified by the first clause, all consecutive rows are verified by the second clause). However, the alternative specification is false: at the first row, we do have $\tau(0) = *$, but there does not exists some $j$ such that $0 < j$ and $\tau(j) = \sigma(0)$, since there never is any output event. Contradiction. □

The essential difference between the two specifications is that the alternative specification (5.1) has progress, whereas the standard specification (5.2) contains deadlocks.

For defining perpetuity and livelock, we encode these as relations between computations. However, it is still an open question what these characterizations entail. Let $P = \{X_1, \ldots, X_n\}$ and $Q = \{Y_1, \ldots, Y_n\}$ be two equally-sized sets of ports. We define *congruence*:

$$\begin{aligned} \mathbf{cong}(P,Q) &:= \forall t.(\bigwedge_{1 \leq i \leq n} X_i(t) = Y_i(t)) \\ \neg\mathbf{cong}(P,Q) &= \exists t.(\bigvee_{1 \leq i \leq n} X_i(t) \neq Y_i(t)) \end{aligned}$$

to mean that ports $P$ behave like ports $Q$. Conversely, *non-congruence* means that ports $P$ and ports $Q$ may disagree at some point. It seems now possible to formulate the proprety of a *fully deterministic* component:

$$\mathbf{det}(\phi) := \exists \vec{X} : U.(\phi(\vec{X}) \wedge \forall \vec{Y} : U.(\phi(\vec{Y}) \rightarrow \mathbf{cong}(\vec{X}, \vec{Y})))$$

that is, given some computation, if every computation is congruent with it, then there can only be a single computation in the execution. And we define *convergence* of $P$ to $Q$:

$$\begin{aligned} \mathbf{conv}(P,Q) &:= \exists s.(s > 0 \wedge \mathbf{cong}(P^{(s)}, Q)) \\ \neg\mathbf{conv}(P,Q) &= \forall s.(s > 0 \rightarrow \neg\mathbf{cong}(P^{(s)}, Q)) \end{aligned}$$

to mean that eventually ports $P$ behave like ports $Q$ do initially. Intuitively, $P$ converges to $Q$ if it is possible to shift $P$ in time to obtain congruence with $Q$. Non-convergence means that $P$ cannot be shifted to obtain congruence with $Q$. Here $P^{(s)}$ denotes the set $P^{(s)} = \{X_1^{(s)}, \ldots, X_n^{(s)}\}$. Now let $\phi(U)$ be a component specification. We define the following:

$$\begin{aligned} \mathbf{perp}(\phi) &:= \forall \vec{X} : U.(\phi(\vec{X}) \rightarrow \forall \vec{Y} : U.(\phi(\vec{Y}) \rightarrow \mathbf{conv}(\vec{X}, \vec{Y}))) \\ \mathbf{cycl}(\phi) &:= \forall \vec{X} : U.(\phi(\vec{X}) \rightarrow \mathbf{conv}(\vec{X}, \vec{X})) \\ \mathbf{acycl}(\phi) &:= \forall \vec{X} : U.(\phi(\vec{X}) \rightarrow \neg\mathbf{conv}(\vec{X}, \vec{X})) \\ \neg\mathbf{acycl}(\phi) &= \exists \vec{X} : U.(\phi(\vec{X}) \wedge \mathbf{conv}(\vec{X}, \vec{X})) \\ \neg\mathbf{cycl}(\phi) &= \exists \vec{X} : U.(\phi(\vec{X}) \wedge \neg\mathbf{conv}(\vec{X}, \vec{X})) \\ \neg\mathbf{perp}(\phi) &= \exists \vec{X} : U.(\phi(\vec{X}) \wedge \exists \vec{Y} : U.(\phi(\vec{Y}) \wedge \neg\mathbf{conv}(\vec{X}, \vec{Y}))) \end{aligned}$$

Intuitively, an execution:

- is *perpetual,* or *livelock-free*, if every computation eventually behaves like every other computation,
- is *cyclic* if every computation eventually repeats itself from the start,
- is *acyclic* if every computation never repeats itself from the start,
- is *non-acyclic* if some computation eventually repeats itself from the start,
- is *non-cyclic* if some computation never repeats itself from the start,
- contains a *livelock* if some computation cannot eventually behave like some other computation.

## 5.4 Instantaneousness

When data flow in and out of components, we intuitively transport such data. If this movement happens without progression of time, then we say data is transported instantaneously. We also consider non-instantaneous components, which may move data elements in time.

**Definition 52.** A component specification is *instantaneous* if its coordination protocol is instantaneous. A coordination protocol is instantaneous if:

- it has the shape $\forall t.\phi$ such that $\phi$ is quantifier free and each occurrence of a port $X$ is in an application $X(t)$,
- it is an existential quantification of some port $X$ of an instantaneous coordination protocol,
- it is a conjunction of two instantaneous coordination protocols.

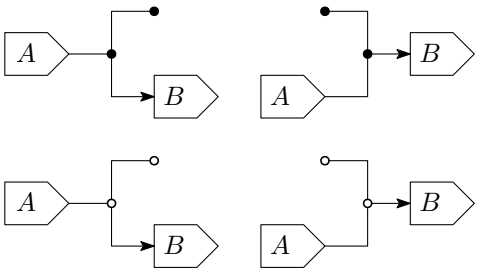
A composition of two instantaneous components is also instantaneous. This follows from parallel composition as a conjunction of the coordination protocols of the two respective components. Identification is also instantaneous, since two ports $X$ and $Y$ that are identified has $\exists X.\exists Y.(\phi \wedge \forall t.(X(t) = Y(t)))$ as protocol, where $\phi$ is the coordination protocol of the underlying composition.

*Remark* 53. The name *stateless* can be alternatively used for *instantaneous*. However, the name stateless is counter-intuitive from the perspective of coordination protocols.

We have that there are certain equivalences of instantaneous components. Two instantaneous coordination protocols $\phi(U)$ and $\psi(U)$ are equivalent whenever they are logically equivalent:

$$\mathbf{equiv}(\phi,\psi) := \forall \vec{X} : U.(\phi(\vec{X}) \rightarrow \psi(\vec{X})) \wedge (\psi(\vec{X}) \rightarrow \phi(\vec{X}))$$

**Example 54.** With this notion of equivalence, we can demonstrate some basic identities between instantaneous components:




All these composite components are equivalent to the SYNCHRONOUS CHANNEL between $A$ and $B$. The equivalence is easily shown by composing the coordination protocols, and simplification of the formula. We do it for the two cases on the left, the others are analogous:

Let $x$ be a REPLICATOR, $y$ be a GARBAGE, the composition has the protocol:

$$\begin{gathered}\exists x.A.\exists x.B.\exists x.C.\exists y.A.\\ \forall t.(x.A(t) = x.B(t) \wedge x.B(t) = x.C(t)) \wedge\\ \forall t.(y.A(t) = * \vee y.A(t) \neq *) \wedge\\ \forall t.(x.A(t) = A(t)) \wedge \forall t.(x.B(t) = B(t)) \wedge \forall t.(x.C(t) = y.A(t))\end{gathered}$$

that can be simplified to $\forall t.(A(t) = B(t) \wedge (B(t) = * \vee B(t) \neq *))$ which is equivalent to $\forall t.(A(t) = B(t))$.

Let $x$ be a ROUTER and $y$ be BLOCKED, the composition has the protocol:

$$\exists x.A.\exists x.B.\exists x.C.\exists y.A.$$
$$\forall t.((x.A(t) = x.B(t) \wedge x.C(t) = *) \vee (x.A(t) = x.C(t) \wedge x.B(t) = *)) \wedge$$
$$\forall t.(y.A(t) = *) \wedge$$
$$\forall t.(x.A(t) = A(t)) \wedge \forall t.(x.B(t) = B(t)) \wedge \forall t.(x.C(t) = y.A(t))$$

that can be simplified to $\forall t.((A(t) = B(t) \wedge * = *) \vee (A(t) = * \wedge B(t) = *))$ which is equivalent to $\forall t.(A(t) = B(t))$. ■

It is still an open question how to formulate instantaneousness as a logical property.

## 5.5 Linearity

A component is linear if all input data and output data are in a 1-to-1 correspondence. Let $\phi(U)$ be a component specification, where $U$ is an interface (of input and output ports) and $\phi$ is a coordination protocol. From the semantic viewpoint, $\mathcal{L}(\phi(U))$ is the set of assignments of streams. Given a solution $\beta \in \mathcal{L}(\phi(U))$, we call the tuple of port, time and data element $(X,t,d)$ an *event* at some port $X$ if $\beta(X)(t) = d$. An event $(X,t,d)$ is an *input event* if $X$ is an input port, and an *output event* if $X$ is an output port.

**Definition 55.** A component specification is linear if for each solution, a bijection $f$ between its input events and output events exists, such that $f(X,t,d) = (Y,s,d)$, that is, each mapped event has equal data.

**Example 56.** Examples of linear components with such interfaces are SILENT and BLOCKED. These components never fire and thus are instantaneous. ■

Any component with interfaces such as $\langle A,B,\ldots \mid\rangle$ ($\langle\mid A,B,\ldots\rangle$) cannot be both linear and have progress. Progress guarantees that one of its ports eventually fires. Since there are no output events (respectively input events), a bijection cannot exist.

Linearity and progress are closely related. If a linear component has an input event, there must exists an output event. However, linearity does not imply progress, since an input event is not necessary and a solution without any events is still linear.

**Example 57.** An example of a non-instantaneous linear component is a BUFFER with input $A$ and output $B$. We assume that the output port has progress, i.e. always eventually fires. This can be demonstrated by the following table:

| $t$ | $A$ | $B$ |
|---|---|---|
| 0 | $*$ | $*$ |
| 1 | $d$ | $*$ |
| 2 | $*$ | $*$ |
| 3 | $*$ | $d$ |
| 4 | $e$ | $e$ |
| 5 | $f$ | $*$ |
| $\vdots$ | $\vdots$ | $\vdots$ |

Here, we have input event $(A,1,d)$ corresponding to the output event $(B,3,d)$. The input event $(A,4,e)$ corresponds to the output event $(B,4,e)$. The input event $(A,5,f)$ corresponds to some output event. This output event must exists, because of the assumption that our output port has progress. ■

Linearity does not imply that related input events and output events are ordered. For example, the following table is still considered linear, since $(A,0,e) \mapsto (B,2,e)$, $(A,1,d) \mapsto (B,0,d)$, and $(A,2,f) \mapsto (B,1,f)$:

| $t$ | $A$ | $B$ |
|---|---|---|
| 0 | $e$ | $d$ |
| 1 | $d$ | $f$ |
| 2 | $f$ | $e$ |
| $\vdots$ | $\vdots$ | $\vdots$ |

Non-linearity is not compositional. The non-linear ARBITRARY component (that generates arbitrary output) composed with one of the inputs of the non-linear CONSENSUS component is equivalent to a SYNCHRONOUS CHANNEL, which is linear.

It is still an open question how to formulate linearity as a logical property.

## 5.6 Causality

Causality is defined by the same technique as how we defined linearity. Intuitively, a component specification is causal if all output data is related to some input data that occurs in the past.

**Definition 58.** A solution is *causal* if for every output event $(X,t,d)$, an input event $(Y,s,d)$ exists such that $s \leq t$. A component specification is *causal* if every solution is causal.

In contrast to linearity, a future output event can be related to multiple past input events. A linear component is not necessarily causal; in case of a linear component, an output event may be related to a future input event.

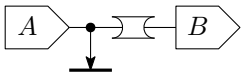
A component $\langle | A,B,\ldots \rangle$ cannot be both causal and have progress. By progress, eventually an output event must occur. However since there are no input ports, there does not exist an input event that can be related to it. Moreover, a component $\langle A,B,\ldots | \rangle$ is trivially causal since there are no output events.

**Definition 59.** A solution is *acausal* if for every output event $(X,t,d)$, an input event $(Y,s,d)$ exists such that $s \geq t$. A component specification is *acausal* if every solution is acausal.

Non-causality and acausality are different, similar to how non-synchronicity and asynchronicity are different.

Finally, we present an example of how acausality captures our understanding of speculative execution.

**Example 60.** Consider the essential part of our running example on page 1.2, depicted here below:




We have two ports: $A$ and $B$, with a PROPHET and a REPLICATOR to a PULL in between. The intention of this part of the protocol is as follows. Before it is known what the actual value of $A$ will be, we can already speculate on it. Thus, for $B$ there is an output event $(B,t,d)$. Then, suppose the actual value for $A$ is known. If this value is different from the speculated value, then $A$ will not fire and remain silent instead. However, the PULL requires $A$ to eventually fire: it is inconsistent if it never fires. Thus, the only consistent behavior that remains is when the output event $(B,t,d)$ corresponds to an input event $(A,s,d)$, where $s \geq t$. By the PULL component, there is always progress on both $A$ and $B$. ■

The intuition behind an acausal component such as the PROPHET is that it specifies prediction of future events. In reality, where we may not know the future events yet. We could implement prophecies using speculative execution.

In a speculative execution, the predicted value can either be a true speculation or a false speculation. If it is a true speculation, it means the future is correctly predicted: the future input event $(Y,s,d)$ corresponds to the past output event $(X,t,d)$. If the predicted value is a false speculation, meaning there is no future event $(Y,s,d)$ that corresponds to the past output event $(X,t,d)$, then we are in an inconsistent configuration due to the PULL component.

An implementation of a PROPHET needs in case of a false speculation to backtrack, undoing the current computation until a previous consistent configuration is reached. With the knowledge of the future from the inconsistent branch, the branch prediction function can now choose to avoid the branch which is known to be inconsistent.

If we do not force an eventual input of the PROPHET, then there is always a consistent behavior by blocking. Even in the case of a false speculation, the input never fires any other data that does not correspond with the (arbitrary) prediction. Intuitively, this amounts to a *self-fulfilling prophecy*: the component holds on to the false speculation and blocks any future attempt to input the actual value. This leads to a non-productive outcome.

It is still an open question how to formulate causality and acausality as a logical property.

*This page is intentionally left blank.*

# Chapter 6

# Conclusion

In this thesis, we have seen the establishment of a logical formalism for defining specifications of the behavior of components. We have studied some important properties for understanding speculative execution, in particular concerning prophecies. The logical formalism allows for formulation of the specification of primitive components, and by the syntax of our typed coordination language we can derive specifications of composite components.

We have seen how coordination games can be used to understand how coordination protocols model an interaction between an environment and a component. We gave an overview of components and we have formalized the properties of *delay insensitivity*, *independence*, *synchronicity* and *asynchronicity*, *progress* and *termination*, *deadlock-freedom* and *livelock-freedom*, *instantaneousness, linearity*, and *causality*. Most properties are formulated using our logical formalism.

This thesis is a contribution to the logical foundations of concurrent and distributed systems, and the logical foundations of Reo in particular, for the following reason: the logical formalism presented in this thesis intuitively captures what we mean by executions and computations. Components specify an execution by constraining the permissible computations. A coordination protocol denotes an execution; its solutions correspond to computations. Properties of coordination protocols can be determined by considering their formalism in which a placeholder is substituted for the coordination protocol for which the property should hold. This allows one to reason about the validity of such properties.

We have seen that some properties are compositional, such as synchronicity. Other properties are not compositional, such as independence. We have made precise what we mean by compositionality, namely, the property is preserved by *parallel composition* and by *identification*. In addition, we have recovered the possibility to derive algebraic identities between instantaneous components.

## 6.1 Summary

This thesis can be summarized in the following key points.

Atomic actions describe the fact that a single event happens at a particular instant. A sequence of actions forms a computation, and interleaving is typically used to understand concurrent executions. We have studied a better model, where we take observations instead of actions as the basis of forming computations. Observations are modeled by a set of (synchronous) actions that happen simultaneously. By formulating constraints, we can precisely describe which observations are allowed. These constraints can be composed more easily than the interleaving of atomic actions.

We have seen components, as they are formed from *primitives* to *composites*. We understand components as binding instance variables; these bindings can be flattened by a normalization procedure. By a well-formedness condition and employing a type system, we ensure that we only work with compositions and components in which ports are linked correctly.

We studied progress, and made a clear distinction between *termination* and *deadlock* by means of *silent* and *actual observations*. We have also formulated properties for *deadlock-freedom* and *livelock-freedom*. The validity of these properties are with respect to a *coordination protocol*, that specifies the behavior of a component. A *coordination game* allows us to see coordination protocols as an interaction between environment and component.

We have studied and explained *speculations* and how we understand *false speculations* and *true speculations*. False speculations occur in speculative computations, that eventually lead to an inconsistent configuration. We have seen how a PROPHET can be used to model a speculative execution. We have also seen that it is necessary to impose a progress condition, using a PULL component, to prevent PROPHETS to deadlock in case of false speculations.

## 6.2 Related Work

In [60], Li and Sun formalize Reo in the Coq theorem prover. They formalize circuits using data packets, which are either pending or delivered. They employ a similar technique in modeling nodes as data streams, where data is either a natural number or empty. Implicitly, they prove independence of an alternator component, by showing that appending empty data packets does not violate the specified behavior.

The graphical notation introduced here is similar to, and inspired by, string diagrams, cf. Selinger's survey on string diagrams [72]. Bruni and others have developed algebraic techniques for stateless Reo components also based on string diagrams [23]. The design of the type system for checking components and compositions is based on the $\overline{\lambda}\mu\tilde{\mu}$-calculus by Curien and Herbelin [29]. The notion of duality of components and swapping input and output ports, is based on the work by Downen and Ariola [34].

In our thesis, we have left out any discussion or reference to constraint automata, to avoid any confusion that might arise between coordination games and constraint automata. Constraint automata also give a semantics to Reo [10]. Jongmans has extended this semantics to constraint automata with memory [49]. These constraint automata with memory still separate data constraints

from synchronization constraints. Data and synchronization are combined, by introducing $*$, and can captured by a single constraint. Automata could subsequently be simplified to a single state, by capturing its state as a memory value. The next insight was to model memory variables by streams [32]. A first-order logic in which constraints on streams can be directly expressed was found, and presented in this thesis.

Reo is very closely related to process algebra (see e.g. [38] for an introduction), with an important and profound difference: process algebras are action-based, Reo is interaction-based. The reasons why process algebra needs proof methodology [39], may also apply to Reo: realistic systems exhibit models that are too large to deal with state-based techniques. Additionally, there are many impossibility results in distributed and concurrent systems, but there is not a single general theory in which we can proof such results.

Reo is related to component-based modeling. Numerous models are proposed for component-based modeling, e.g. separating component from connector [3], a calculus based on asynchronous $\pi$-calculus [65]. However, as Gössler mentions, the question of properties of component-based systems have not been studied systematically [43]. In this thesis, we have set out to define such a set of properties, thereby contributing to this question.

Vereofy is a model checking tool for analyzing Reo circuits [11]. It accepts two input languages: one for construction of compositions, and one for defining constraint automata. The model checker then allows verification of temporal properties expressed in linear temporal logic (LTL) and computational tree logic (CTL).

The idea of formalizing the behavior of a component in a first-order logic is not new [21]. Broy gives a formalism that is similar to the one presented in this thesis [20], that allows for the equational specification of components by using predicates. The behavior of components are represented by stream processing functions, that map a tuple of streams correspond to input ports to a tuple of streams corresponding to output ports. A component is defined by the set of acceptable such behaviors; similar to our interpretation of coordination protocols. Broy defines an algebra for composition, in a similar way as we have done: $C \parallel C$ for parallel composition, $\mu_x^y$ for so-called feedback of channels that is comparable to our identification of ports, and $\rho_y^x$ for renaming channel names. He defines the predicate interpretation of a composition in a very similar way as what we have done: a parallel composition takes the conjunction of the specifications of two components, and specifies that the shared channels must be identical.

The notion of a prophecy is well known. In 1986, Dijkstra explicitly forbids "clairvoyance" [31], being the property of being able to receive messages before they are sent, when discussing the distributed snapshot algorithm of Chandy and Lamport [25]. Prophecy is also known as *possibility*, and used in the proof of Herlihy and Wing for techniques showing linearizability of concurrent objects [45]. They refer to the paper of Abadi and Lamport of 1988 [1], whom claim "as far as we know, prophecy variables are new".

# Bibliography


[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

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

[3] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[4] Anonymous. Performance considerations for L1 terminal fault. `https://access.redhat.com/security/vulnerabilities/L1TF-perf`. (Accessed 24-Aug-2018).

[5] Farhad Arbab. What do you mean, coordination? *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, 19, 1998.

[6] 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.

[7] Farhad Arbab. Elements of Interaction. In *Complex Systems Design & Management*, pages 1–28. Springer, 2010.

[8] Farhad Arbab. Puff, the magic protocol. In *Formal Modeling: Actors, Open Systems, Biological Systems*, pages 169–206. Springer, 2011.

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

[10] Christel Baier et al. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.

[11] Christel Baier et al. Formal verification for components and connectors. In *International Symposium on Formal Methods for Components and Objects*, pages 82–101. Springer, 2008.

[12] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT press, 2008.

[13] Rachel Bailey. *A Comparative Study of Algorithms for Solving Büchi Games*. MSc dissertation, University of Oxford, 2010.

[14] Bruno Barras et al. *The Coq proof assistant reference manual*. PhD thesis, INRIA, 1997.

[15] Henning Basold et al. Newton Series, Coinductively. In *International Colloquium on Theoretical Aspects of Computing*, pages 91–109. Springer, 2015.

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

[17] Klaus Bergner et al. A Formal Model for Componentware. In *Formale Beschreibungstechniken für verteilte Systeme*, pages 17–26, 1999.

[18] Anasua Bhowmik and Manoj Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 99–108. ACM, 2002.

[19] Ana Bove et al. 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.

[20] Manfred Broy. Algebraic specification of reactive systems. *Theoretical Computer Science*, 239(1):3 – 40, 2000.

[21] Manfred Broy. Time, abstraction, causality and modularity in interactive systems. *Electronic Notes in Theoretical Computer Science*, 108:3–9, 2004.

[22] Manfred Broy. Relating time and causality in interactive distributed systems. *Engineering Methods and Tools for Software Safety and Security*, 22:75, 2009.

[23] Roberto Bruni, Ivan Lanese, and Ugo Montanari. A basic algebra of stateless connectors. *Theoretical Computer Science*, 366(1):98 – 120, 2006.

[24] Roberto Bruni and Ugo Montanari. Zero-safe nets, or transition synchronization made simple. *Electronic Notes in Theoretical Computer Science*, 7:55–74, 1997.

[25] K.M. Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[26] K.M. Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, 1983.

[27] Dave Clarke. Coordination: Reo, Nets, and Logic. In *Formal Methods for Components and Objects*, pages 226–256. Springer, 2008.

[28] Ernie Cohen. Separation and reduction. In *International Conference on Mathematics of Program Construction*, pages 45–59. Springer, 2000.

[29] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of the 5th ACM International Conference on Functional Programming*, volume 35, pages 233–243. ACM, 2000.

[30] Leonardo de Moura et al. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[31] Edsger W. Dijkstra. The distributed snapshot of K.M. Chandy and L. Lamport. In *Control Flow and Data Flow: Concepts of Distributed Programming*, pages 513–517. Springer, 1986.

[32] Kasper Dokter and Farhad Arbab. Rule-based form for stream constraints. In *International Conference on Coordination Languages and Models*, pages 142–161. Springer, 2018.

[33] Kasper Dokter and Farhad Arbab. Treo: textual syntax of Reo connectors. *Proceedings of 1st International Workshop on Methods and Tools for Rigorous System Design*, 2018.

[34] Paul Downen and Zena M. Ariola. The duality of construction. In *European Symposium on Programming Languages and Systems*, pages 249–269. Springer, 2014.

[35] Herbert Enderton. *A Mathematical Introduction to Logic*. Elsevier, 2001.

[36] 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.

[37] Wan Fokkink. *Distributed Algorithms: An Intuitive Approach*. MIT Press, 2013.

[38] Wan Fokkink. *Introduction to Process Algebra*. Springer, 2013.

[39] Wan Fokkink, Jan Friso Groote, and Michel Adriaan Reniers. Process algebra needs proof methodology. *EATCS Bulletin 82*, pages 109–205, 2004.

[40] Michael Frank. Introduction to reversible computing: motivation, progress, and challenges. In *Proceedings of the 2nd Conference on Computing Frontiers*, pages 385–390. ACM, 2005.

[41] M.R. Frias et al. Dynalloy: upgrading Alloy with actions. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 442–450. IEEE, 2005.

[42] David Geer. Is it time for clockless chips? (Asynchronous processor chips). *Computer*, 38(3):18–21, 2005.

[43] Gregor Gössler et al. An approach to modelling and verification of component based systems. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 295–308. Springer, 2007.

[44] Ed Grochowski et al. Best of both latency and throughput. In *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pages 236–243. IEEE, 2004.

[45] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[46] Peter Höfner, Bernhard Möller, and Kim Solin. Omega algebra, demonic refinement algebra and commands. In *International Conference on Relational Methods in Computer Science*, pages 222–234. Springer, 2006.

[47] Jann Horn et al. Reading privileged memory with a side-channel. `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`, 2018. (Accessed 24-Aug-2018).

[48] Mohammad Izadi and Marcello M. Bonsangue. Recasting constraint automata into Büchi automata. In *International Colloquium on Theoretical Aspects of Computing*, pages 156–170. Springer, 2008.

[49] Sung-Shik Jongmans. *Automata-theoretic protocol programming*. PhD thesis, Centrum Wiskunde & Informatica (CWI), Leiden University, 2016.

[50] Sung-Shik Jongmans and Farhad Arbab. Overview of Thirty Semantic Formalisms for Reo. *Scientific Annals of Computer Science*, 22(1), 2012.

[51] Sung-Shik Jongmans and Farhad Arbab. Global consensus through local synchronization: A formal basis for partially-distributed coordination. *Science of Computer Programming*, 115-116:199 – 224, 2016.

[52] Sung-Shik Jongmans, Christian Krause, and Farhad Arbab. Encoding context-sensitivity in Reo into non-context-sensitive semantic models. In *International Conference on Coordination Languages and Models*, pages 31–48. Springer, 2011.

[53] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. Technical report, Project RAND, Santa Monica, 1951.

[54] Paul Kocher et al. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[55] Christian Koehler et al. Reconfiguration of Reo connectors triggered by dataflow. *Electronic Communications of the EASST*, 10, 2008.

[56] Dexter Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and computation*, 110(2):366–390, 1994.

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

[58] Christian Krause et al. Modeling dynamic reconfigurations in Reo using high-level replacement systems. *Science of Computer Programming*, 76(1):23 – 36, 2011.

[59] Ten H. Lai and Tao H. Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[60] Yi Li and Meng Sun. Modeling and verification of component connectors in Coq. *Science of Computer Programming*, 113:285–301, 2015.

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

[62] Panagiotis Manolios. Correctness of pipelined machines. In *International Conference on Formal Methods in Computer-Aided Design*, pages 181–198. Springer, 2000.

[63] Jacques Mattheij. The several million dollar bug. `https://jacquesmattheij.com/the-several-million-dollar-bug/`, 2014. (Accessed 29-Aug-2018).

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

[65] Oscar Nierstrasz and Franz Achermann. A calculus for modeling software components. In *Formal Methods for Components and Objects*, pages 339–360. Springer, 2003.

[66] David Park. Concurrency and automata on infinite sequences. In *Theoretical computer science*, pages 167–183. Springer, 1981.

[67] Bahman Pourvatan, Marjan Sirjani, Hossein Hojjat, and Farhad Arbab. Symbolic execution of Reo circuits using constraint automata. *Science of Computer Programming*, 77(7):848 – 869, 2012.

[68] Cristian Prisacariu. Synchronous kleene algebra. *The Journal of Logic and Algebraic Programming*, 79(7):608–635, 2010.

[69] José Proença et al. Dreams: a framework for distributed synchronous coordination. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1510–1515. ACM, 2012.

[70] J.J.M.M. Rutten. Elements of stream calculus: (an extensive exercise in coinduction). *Electronic Notes in Theoretical Computer Science*, 45:358 – 423, 2001.

[71] J.J.M.M. Rutten. *On Streams and Coinduction*. Unpublished manuscript, 2002.

[72] Peter Selinger. A survey of graphical languages for monoidal categories. In *New structures for physics*, pages 289–355. Springer, 2010.

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

[74] Kenneth S. Stevens. Energy and performance models for clocked and asynchronous communication. In *Ninth International Symposium on Asynchronous Circuits and Systems*, pages 56–66. IEEE, 2003.

[75] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(3):204–226, 1985.

[76] Andrew Tanenbaum. *Structured Computer Organization (6th edition)*. Prentice Hall, 2013.

[77] Tommaso Toffoli. Reversible computing. In *International Colloquium on Automata, Languages, and Programming*, pages 632–644. Springer, 1980.

[78] Tommaso Toffoli and Norman H. Margolus. Invertible cellular automata: A review. *Physica D: Nonlinear Phenomena*, 45(1-3):229–253, 1990.

[79] Augustus Uht and Vijay Sindagi. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 313–325. IEEE/ACM, 1995.

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

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