Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Yet Another Reo Semantics:

Reasoning about Speculative Execution

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

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

September 11, 2018

Abstract

Five sentences.

Contents

1 Introduction

A recent interest in fundamental security of microcode architectures of mainstream processors revealed issues (popularily known as Meltdown, Spectre, and Foreshadow; see Kocher et al. [11]). These architectural issues demonstrate side-channel attacks caused by a combination of branch prediction, cache hierarchy, simultaneous multithreading and speculative execution. In particular, it turns out that when computation is not reversible under certain conditions, this could lead to a systematic leakage of priviledged information. Mitigation of these issues have serious impact for large-scale computing service providers: under certain workloads perceived performance will be reduced by 50% [8].

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 might also negatively affect throughput: a false speculation incurs costs and 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 dissipiate power and thus is highly energy efficient [16]. Some claim that adoption of reversible computing is necessary to keep the rate of performance improvements of general-purpose computing as seen in past decades stable [7].

In this paper, we aim to gain a fundamental understandig of concurrency, speculative execution, and reversible computing. The vision is to work on logical foundations for concurrent and distributed systems. We shall employ a structured approach by modelling interactive computing systems as coordination protocols using Reo. Reo is a coordination language. Coordination is the study of dynamic topologies of interaction among computing systems [1].

From the practical side, Reo manifests as a high-level programming language for constructing concurrent and distributed systems, exemplified by recently developed compilers that produce concurrency glue code that is linked together with existing single-threaded code. This is beneficial to programmers whom no longer are required 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. This is achieved by means of raising the level of abstraction. This alleviates programmers from working with low-level concurrency primitives (e.g. semaphores, locks) and debugging concurrency errors (e.g. race conditions, deadlocks). Instead, programmers specify coordination protocols to define the permissible interactions between external single-threaded programs, thus separating their concerns for functionality and concurrency. Secondary, experiments show that concurrency code generated by Reo compilers has run-time performance similar to hand-crafted concurrency code [9]. This is achieved by performing program optimization on the level of coordination protocols.

Over the years Reo has developed a rich theory, as demonstrated by over thirty formal semantics [10]. These semantics can be roughly divided into three groups: coalgebraic models (say, streams), operational models (say, automata), and others. In this paper, we introduce yet another semantics and methods for logical analysis.

Our presentation of Reo is novel in a way: we focus on the capability to detect and handle inconsistencies. An inconsistency in a system is effectively

resolved by tracing back to a nearest branch point from which execution can be resumed safely. Our language is typed and graphical. We shed light on dualities present in our language, by the definition of so-called buffers and prophets, that respectively correspond to history variables and future variables. To demonstrate our semantics, we will study speculative execution.

The main results of this paper are the establishment of a formal semantics of components and some of its important properties: first, we introduce a big-step game semantics (Section 2). Second, we define the syntax of a typed coordination language (Section 3). Third, we interpret our game semantics in a first-order logic and compositionally interpret our coordination language (Section 4). Fourth, we develop methods for logically analysis, to establish the properties of independent progress, causality, and linearity (Section 5).

We argue that these properties are related to an intuitive understanding of concurrency, speculative execution, and reversibility, and that our paper contributes to logical foundations of concurrent and distributed systems, and logical foundations of Reo in particular.

2 Speculative Execution

In central processing units (CPUs), out-of-order execution is a technique with an instruction pipeline architecture to increase performance of single-threaded code [15]. The behavior of a single-threaded program is defined by the order of its instructions. With out-of-order execution, instructions could be performed 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 number of instructions that can be throughput per time unit, by efficiently planning the use of computational resources such as arithmetic and logic units (ALUs) and floating point units (FPUs), resulting in better run-time performance.

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

Speculative execution not only applies to hardware. For example, speculative multithreading implements speculative execution in software [4]. Speculative execution is also conceivable in distributed systems[1].

In general, it is well-known that speculative execution in concurrent systems can variably increase throughput. The rate of increased throughput depends on the particular implementation technique applied, of which there are multiple. We consider two extreme cases: embarrassingly parallel and backtracking.

An embarrassingly parallel implementation branches into multiple isolated systems and performs computation in all branches concurrently. Separate branches are isolated systems and thus may not communicate with each other. The num-

[1] It is an urban legend to increase throughput of webservers by sending a specific HTTP response before any HTTP request is received. This was supposedly used in early webcam software [12].

ber 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, since every possibility executes concurrently, thus this technique has the highest gain in throughput. Branches that compute 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 chooses which branch to perform first. If at some later time this branch turns out to be 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 needs to be reverted before the true computation may commence.

In summary, we think of these two extremes as a trade-off between time and space. Embarrassingly parallel has negligible time overhead, but has a space overhead linear in the number of branches to compute speculatively. Backtracking 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, eager execution executes all branches with negligible space overhead by preemtive scheduling, similar to an iterative deepening search strategy [17].

Over the course of this paper, we shall construct a (toy) processing unit. Our toy is used as the running example for this paper. Our construction of a processing unit is not necessarily centralized (like CPUs), but can also be considered a decentralized processing unit. We shall explore interesting features of speculative execution. However, it is not our goal to construct a fully functional CPU. This exercise is approached as follows:

1. We construct our processor in a modular fashion to allow ourselves to reason about smaller individual pieces. Compositionality of reasoning ensures that the end result of composing all pieces together will behave as designed. This forms the basics of how to use Reo to (de)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 understandig the formal theory, and helps us understand the nature of speculative execution.

The intended purpose of this exercise 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, which is a high-level description of the operations that components perform. The game avoids any inconsistent states.
3. Logical analysis. We reason about compositional properties, and the necessity of reversibility of computations based on speculation, to argue that backtracking is an effective implementation strategy.

2.1 Case Specification

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: the running-time complexity of these operations depend 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.

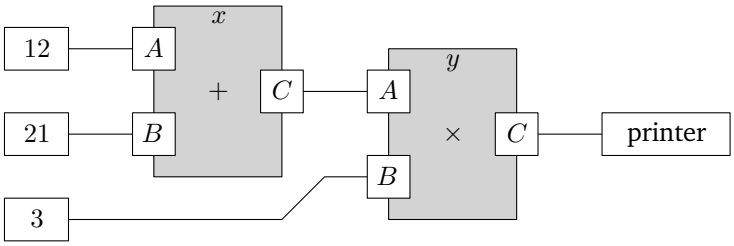


Figure 1

The addition component simulatenously 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. 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. Input ports are typically on the left, while output ports are typically on the right. In the circuit above we have ports *A*, *B*, and *C*.

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 has a number of associated ports, which we denote *x.A*, *x.B*, *x.C*, *y.A*, *y.B*, *y.C*. We haved link ports together to indicate that information flows between ports: here *x.C* and *y.A* are linked together.

The 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 inputs are available. It asychrnonously computes the outcome. Typically, one would abstract from this fact by assuming that arithmetic can be 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 some 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, say pseudo-Java, one would write:

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

This is given by following component circuit:

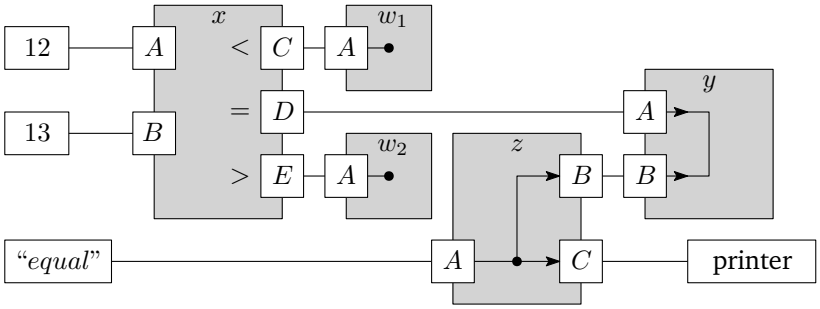


Figure 2

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 so-called garbages that dispose their input. The component y is a synchronous drain, that mediates synchronization in this circuit. Component z is a replicator that instanteneously 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 controlled valve. Only when $x.D$ fires a signal, will the synchronous drain also accept 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.

Note that 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 could be sent to the printer before the conditional check is performed. Intuitively, synchronous actions are an abstraction of the precise ordering of individual actions, as this ordering is irrelevant.

That the replicator and drain act as a controlled valve can be made explicit, by turning it into a composite component:

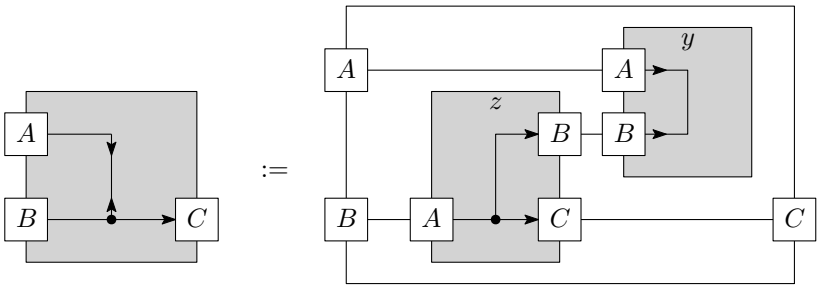


Figure 3

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. Each component has three ports on its exteriour boundary: *A*, *B*, *C*. Here we link the inputs *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 only link output ports to input ports. Hence, from the interior perspective, *A* and *B* act as output ports passing the supplied value along: these outputs are linked to the inputs of *y* and *z*. In general, the direction of a boundary port is opposite in the interior as compared to the exteror.

Another essential unit in processors are registers and memory banks. Typically, memory in modern CPUs are layed out using cache hierarchies. This means that accessing memory has variable and dynamic latencies. We specify a simple memory bank by first considering memory cells. Memory cells are modeled using controlled variables, for which we reuse our controlled valve of before:

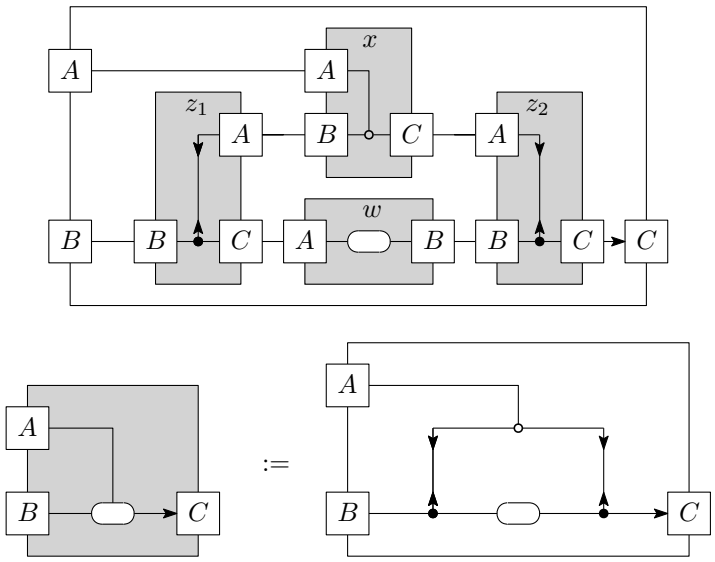


Figure 4

Note that we have two pictures: one where all components are explicit, and

the one below where we no longer drawn the component boundaries. Our composition consists of two controlled valves, a router, and a variable. Port A controls one of the two valves on B and C, with a variable in between. The variable component is a primitive component for which we will later define its semantics. For now, it is sufficient to think of variables as a component that continuously outputs its most recently supplied input value, if any. Variables may be overwritten and may output its value multiple times. The controlled variable either accepts input if port A fires, or provides output if port A fires. Never all ports fire simultenously, and B or C only fires if A fires.

It is important to understand that a controlled variable is not reversible, because a variable is not reversible. 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. The presence of A and the controlled valves do not change this fact.

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:

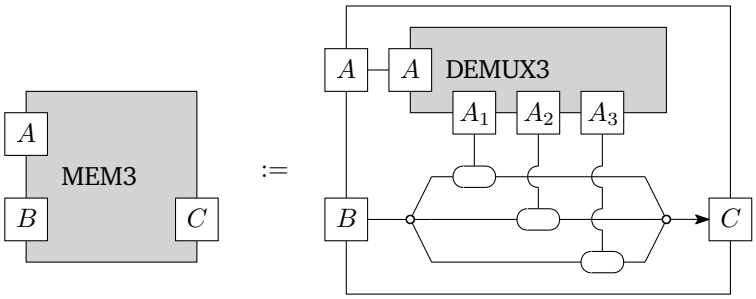


Figure 5

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 (say $1,2,3$) and translates it into a signal (on say A_1,A_2,A_3). We have three controlled variables that model three memory cells, connected to these signals: a so-called router takes the input at B and is connected to the input of each cell, and a so-called merger takes the output of each cell and merges into the output C.

We shall assume that the initial value of all memory cells is empty. An empty memory cell cannot be read, whereas a non-empty memory cell can be read with 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 $1, 2, 3$. We create the following components:

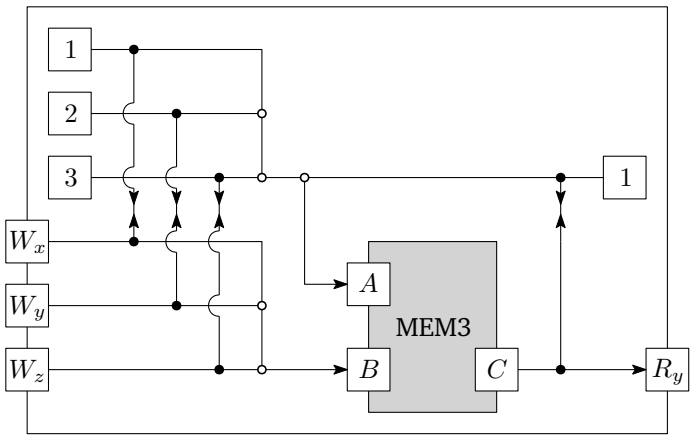


Figure 6

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

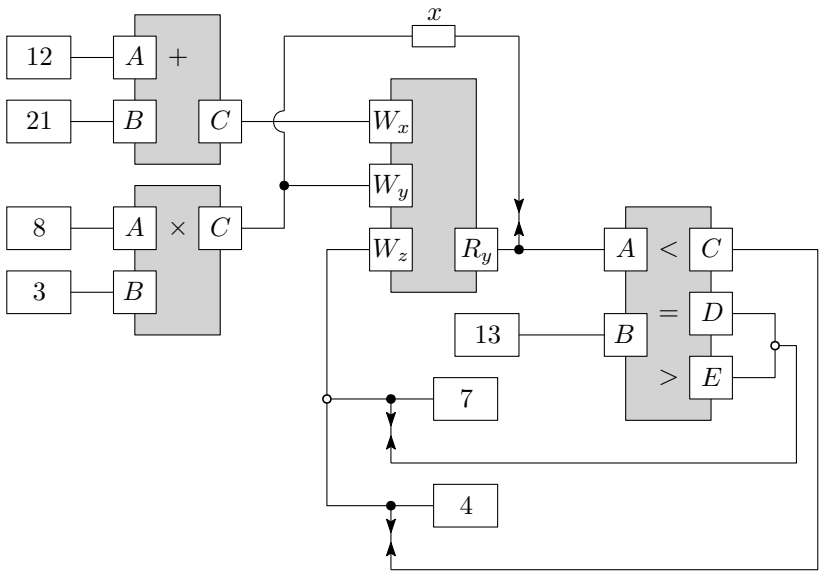


Figure 7

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 8×3, 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. Note that it is still possible in our circuit that W_x is delayed after W_z.

A buffer is a component that, intuitively, behaves so: 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.

Note that in our circuit, the buffer stores the computed value of y, but its buffered value is only used for a controlled valve: only the actual stored memory value is being compared, not the computed value. In this circuit these two values correspond. However, in a more complicated example where memory can also be affected by other computations, 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 awaits the arrival of the true value, and if that value equals the speculation the prophet will properly reset. However, if the arrived value is different than the speculation, it means we had done a false speculation that is inconsistent and we need to perform error recovery.

Consider inserting a prophet y between the replicator after R_y and before A of the comparator, as shown in the following circuit:

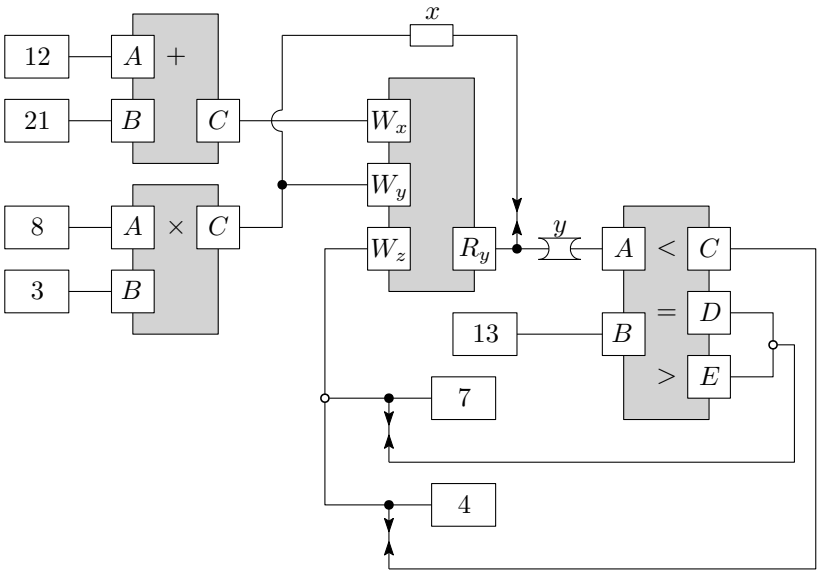


Figure 8

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 will fire with 4. Suppose that $y \geq 13$: then W_z will fire 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), we have a true speculation. Thus we have already performed computation (writing 7 to z), thereby gaining throughput. However, 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.

We shall now explore this example in more depth: by defining constraint automata and game semantics. We revisit our example of speculative execution in Section 5, where detailed properties of components are logically analyzed.

2.2 Constraint Automata

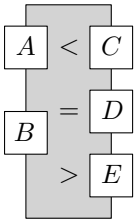
In this section we consider constraints and constraint automata. Constraints are formulas in a zeroth-order many-sorted logic with equality. Constraint automata are non-deterministic automata with constraints as transition labels.

The reason for considering constraints in the first place is to model synchronous behavior. We describe synchronous behavior by a constraint on the permissible values that can be observed at each port in a single instant.

Let $\mathbb{T}$ be a set of data types. Fix some signature Σ and interpretation $\mathcal{M}$, such that Σ consists (at least) of one sort for each data type in $\mathbb{T}$, and for each data type a constant $*$. Each data type is interpreted by $\mathcal{M}$ by some algebra that is closed under $*$, the interpretation of functions is required that $*$ as (one of its) argument maps to $*$, the interpretation of predicates is required that $*$ as (one of its) argument is false.

Let F^0_Σ be the set of quantifier free formulas, and $\phi, \psi, \ldots$ denote such formulas. Variables are ports $A, B, C, \ldots$ that have a sort that corresponds to a data type. These formulas are so-called constraints, and finding an assignment such that a constraint is satisfiable is also called constraint solving. We shall assume that constraint solving is decidable.

We assume constraints can be normalized in disjunctive normal form (DNF); implication is material, and conjunction and disjunction are distributive. Equalities and predicates are called literals. A conjunction of literals is called a clause.



Example 1. The comparator of previous section can be understood by the following constraint. This constraint is in DNF:

$$\begin{aligned}(A = * \wedge B = * \wedge C = * \wedge D = * \wedge E = *) \vee \\ (A < B \wedge C \neq * \wedge D = * \wedge E = *) \vee \\ (A \equiv B \wedge C = * \wedge D \neq * \wedge E = *) \vee \\ (A > B \wedge C = * \wedge D = * \wedge E \neq *)\end{aligned}$$

We take A and B to be of sort Nat, and C, D, E of sort Signal. The signature of Nat consists of the constant $*$, and the predicate symbols $<, \equiv, >$. The signature of Signal consists of the constant $*$. The Nat-algebra has $\{*, 0, 1, 2, \ldots\}$ as carrier, the Signal-algebra has $\{*, 0\}$ as carrier. The non-logical symbols are interpreted as standard, with the exception that $* < x$, $x < *$, $x \equiv *$, $* \equiv x$, $x > *$, $* > x$ are

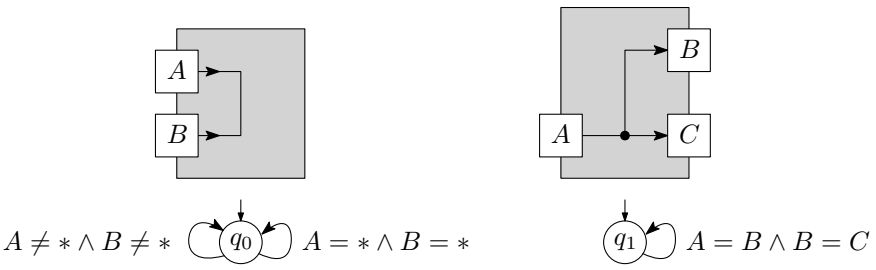
always false. We have that, for example, $A \mapsto 1, B \mapsto 3, C \mapsto 0, D \mapsto *, E \mapsto *$ is a valid assignment. ■

We then consider finite state automata that are labeled by constraints. Our definition of constraint automata is different than defined by Baier et al. [2], whom use a set of port names as synchronization constraint and a formula as data constraint. Since we require that each data type contains $*$, our synchronization and data constraint is expressed by a single formula. This simplifies the presentation of constraint automata considerably.

Definition 2. A constraint automaton (Q, I, P, R) consists of the following data:

1. A denumerable set of states Q, a non-empty subset $I \subseteq Q$ of initial states.
2. A finite set of ports P.
3. A transition relation $R \subseteq Q \times F_\Sigma^0 \times Q$.

We write $q \xrightarrow{\phi} q'$ to denote $(q, \phi, q') \in R$, or we say $q \to q'$ with constraint ϕ. The set P represents the interface of a constraint automaton. Ports that occur in constraints in R and are not in P, are said to be *hidden ports*.



Example 3. We define two constraint automata, depicted above: a synchronous drain and a a replicator. The synchronous drain is defined by $Q = \{q_0\}$, $I = \{q_0\}$, $P = \{A, B\}$, $R = \{(q_0, A \neq * \wedge B \neq *, q_0), (q_0, A = * \wedge B = *, q_0)\}$. The replicator is defined by $Q = \{q_1\}$, $I = \{q_1\}$, $P = \{A, B, C\}$, $R = \{(q_1, A = B \wedge B = C, q_1)\}$.

The automata corresponding to the synchronous drain has two constraints. At any state, either both ports must fire ($A \neq * \wedge B \neq *$) or both ports must block ($A = * \wedge B = *$). Note that the data of port A and B are not related when the ports fire: e.g. assignment $A \mapsto 0, B \mapsto 12$ is satisfiable.

Similarly, the automaton correspond to the replicator has a single constraint: all data must always be equal at all ports. This means that either all ports fire ($A \neq * \wedge B \neq * \wedge C \neq *$) and share the same data, or that no port fires at all and no data is exchanged ($A = * \wedge B = * \wedge C = *$). ■

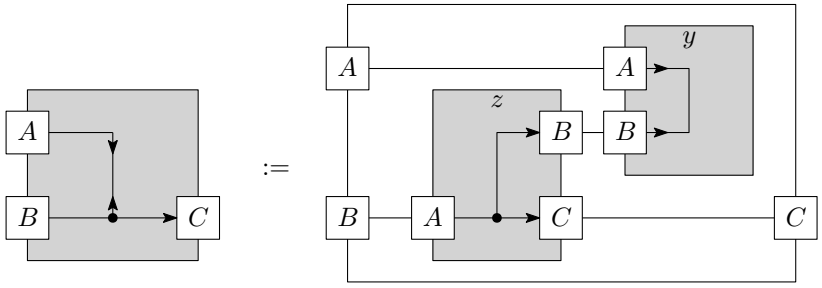
Composition, as seen before diagrammatically, can also be defined as an operation on constraint automata.

Definition 4. Let $C_1 = (Q_1, I_1, P_1, R_1)$ and $C_2 = (Q_2, I_2, P_2, R_2)$ be two constraint automata, such that:

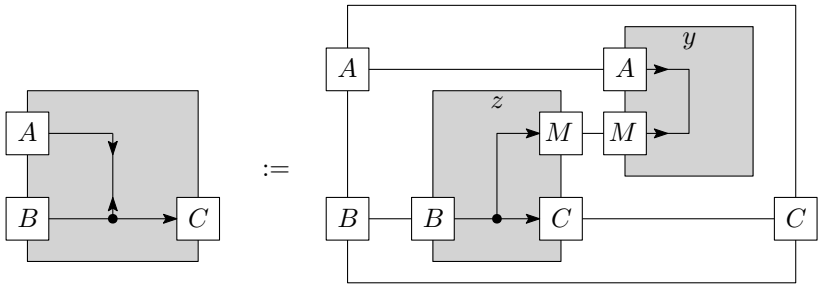
1. Q_1 and Q_2 are disjoint, and
2. all constraints of R_1 do not contain ports $P_2 \backslash P_1$, and
3. all constraints of R_2 do not contain ports $P_1 \backslash P_2$.

Then, the composition $C_1 \bowtie C_2$ is defined as $(Q_1 \cup Q_2, I_1 \cup I_2, P_1 \cup P_2, R_1 \cup R_2)$.

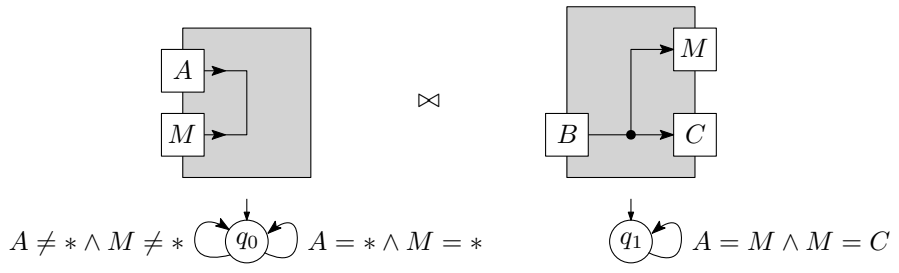
Constraints of two constraint automata may share ports. If a port is common, in both P_1 and P_2, then the composition intentionally identifies these ports. The last two conditions ensure that hidden ports of R_1 (resp. R_2) does not interfere with non-hidden ports of R_2 (resp. R_1). Thus when composing arbitrary constraint automata, we must take care of suitable renaming of ports. This is illustrated by the following example.



Example 5. The controlled valve is composed of a synchronous drain and a replicator, as depicted above. We perform the following steps: first, we rename the ports of the automata that we compose making sure that linked ports have the same name; second, we compose the two automata; third, we hide ports.

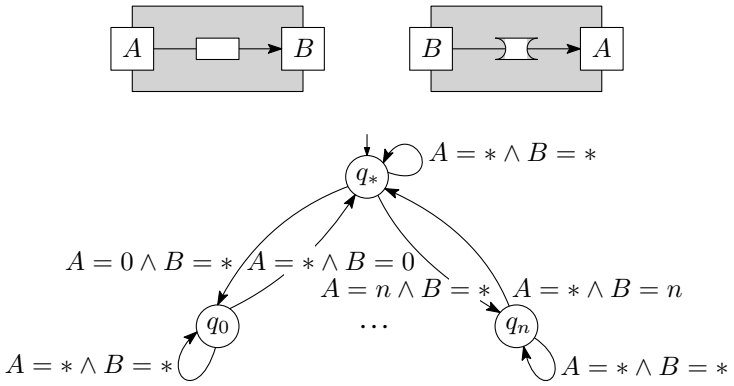


For y, we rename B to M; for z, rename A to B and B to M. We also do this for the corresponding constraint automata. Now M is intentionally identified between the two automata:



Finally, we take as interface $\{A,B,C\}$ and thereby hiding M. The resulting constraint automaton is: $Q = \{q_0,q_1\}$, $I = \{q_0,q_1\}$, $P = \{A,B,C\}$, $R = \{(q_0, A \neq * \wedge M \neq *, q_0), (q_0, A = * \wedge M = *, q_0), (q_1, A = M \wedge M = C, q_1)\}$. ■

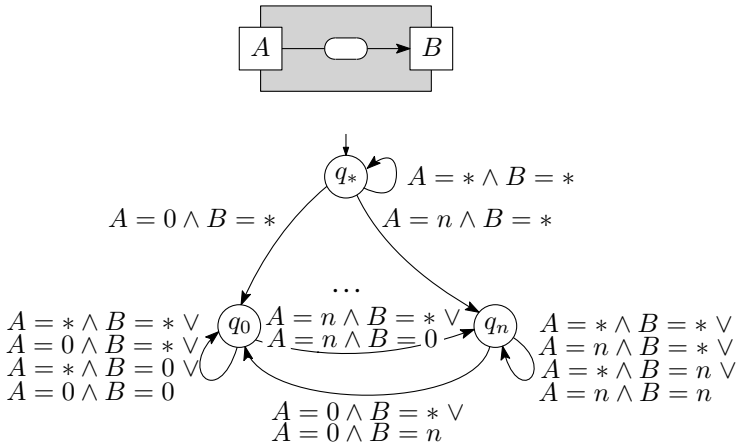
The last two examples we will treat are: buffers/prophets and variables. These primitive components have multiple states. We fix some data type, say Nat, to keep our presentation concrete: however, this semantics can easily be generalized to any data type.



Example 6. The constraint automaton for a buffer/prophet is depicted above. Both buffers and prophets have the same automaton. For each element $n \in \text{Nat}$ there exists a unique state q_n. Each state models the current value of the buffer. In particular q_* is the initial state for the empty buffer, since we have $* \in \text{Nat}$. The transition relation is defined as follows:

- Let q_k be an arbitrary state. There is a transition $q_k \to q_k$ with the constraint $(A = * \wedge B = *)$.
- Let q_n be an arbitrary state different from q_*. There is a transition $q_* \to q_n$ with the constraint $(A = n \wedge B = *)$, and a transition $q_n \to q_*$ with the constraint $(A = * \wedge B = n)$.

A buffer accepts input, and remains full after port A fires with some data. It then becomes empty again, by firing B with the same data, and resetting to the empty state. A prophet is similar, but with its inputs and outputs swapped: it outputs an arbitrary value on A, and remains in a speculation state until an input arrives at B that verifies the prediction. ■



Example 7. The constraint automaton for a variable is depicted above. For each element $n \in \text{Nat}$ there exists a unique state q_n. Each state models the current value of the variable, and q_* is the initial state for an empty variable. The transition relation is defined as follows:

- Let q_k be an arbitrary state. There is a transition $q_k \to q_k$ with the constraint $(A = * \wedge B = *) \vee (A = * \wedge B = k)$.

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

Table 1: Examples of observations

- Let q_k and q_n be two arbitrary states, where q_n is different from q_*. There is a transition $q_k \to q_n$ with constraint $(A = n \wedge B = *) \vee (A = n \wedge B = k)$.

Once a variable becomes full, it never becomes empty: q_* is only reachable from q_*. The constraint $A = * \wedge B = k$ corresponds to some output at port B: it does not change our state. The constraint $A = n \wedge B = *$ corresponds to some input at port A: we change to, or remain in, state q_n afterwards. A combination of these two is $A = n \vee B = k$: we have input, changing to state q_n, and output, from state q_k, combined. Finally, the constraint $A = * \wedge B = *$ corresponds to waiting without any input or output: we stay in the same state. ■

2.3 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 in this section. The coordination game resolves the constraints of components with an environment that is out of the control of a component. Say, the environment initiates an inward flow, then the component may choose to 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 of a special 'null' value $*$ that represents the absence of data, and within data streams it is used to indicate that no information is flowing at a moment.

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

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

A useful way of thinking of a component and its ports is by considering a tabulation of observations. Say, we have an arbitrary component with two ports, X and Y. Here X is an input port, and Y is an output port. We shall observe it over

some period of time. In Table 1 we see four observations: the first observation (at $t = 0$) shows some data element d flows through port X. Since X is an input port, the element d is supplied by the environment and deemed acceptable by the component: the environment and component have completed their coordination game and the result is that the data element d is exchanged through port X. The last observation (at $t = 4$) shows that the data element d, which is the same element as previously, flows out of output port Y: again a coordination game is completed.

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

The notion of completeness here means that there is no hidden information flow. For example, in Table 1, all information that flowed in or out the component between $t = 0$ and $t = 4$ is as shown. For example, this implies that between time $t > 0$ and $t < 4$, there was no activity at either port. It still happens that the component and environment are playing a coordination game; the result of that game is tabulated.

Component behavior is defined in terms of permissible traces of observations. We model an observation by an assignment of ports to values. We define coordination games that gives rise to these traces, and give examples how the previously considered components are played.

Let (Q, I, P, R) be a constraint automaton. By $\beta \vDash \phi$ we mean that assignment β of port variables to elements *solves* ϕ with respect to some (fixed) interpretation: substituting all variables assigned by β in ϕ is satisfiable. If $\beta \vDash \phi$, we say β is a *solution* for ϕ. Moreover, if no solution β exists we call ϕ *inconsistent*. A state q such that for each outgoing transition (q, ϕ, q') the constraint ϕ is inconsistent is called an *inconsistent state*.

A *configuration* is a non-empty subset of Q. The *initial configuration* is I. An *arc* between configurations is a tuple (C, β, C'), where C and C' are configurations and β an assignment. A *path* is a sequence of alternations of configurations and assignments.

Arcs are denoted $C \xrightarrow{\beta} C'$, and paths are denoted $C_0 \xrightarrow{\beta_1} C_1 \xrightarrow{\beta_2} C_2 \cdots C_{n-1} \xrightarrow{\beta_n} C_n$.

Two paths *follow* if the last configuration of the first path is the first configuration of the last path. A path starting with the initial configuration is a *prefix*. A *cycle* is a path of configurations such that the first and last configuration is the same. A *trace* is a path formed by a prefix followed by a cycle. A *dead-end* is a prefix that is not a trace. We define *inclusion* of a prefix in a trace, by checking whether there exists a trace unrolling such that the prefix followed by the path equals the unrolled trace.

A coordination game is modelled by a graph with configurations as vertices and arcs as edges. Each arc represents a step. The number of steps away from the initial state indicates the current round. Each configuration reachable from the initial configuration is at some round.

A configuration represents a set of states of a constraint automaton. The constraint automaton could be a composition of smaller constraint automata,

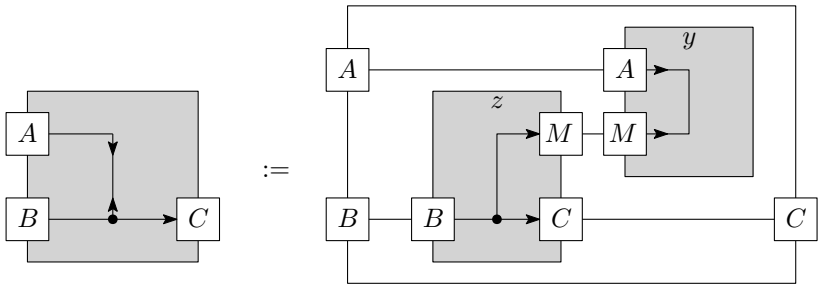
and the set of states represent the (con)current state in all automata.

Suppose we are in some round, and we are given an assignment. The configuration of the next round is found by taking transitions in the constraint automaton. In particular, for each state in our configuration, there must exists an outgoing transition labeled by a constraint that is solved by our given assignment. And, for each state in the configuration of the next round, there must exist an incoming transition labeled by a constraint that is solved by our given assignment. We call this *taking* a transition.

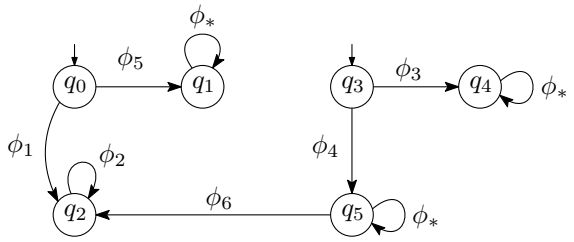
Definition 8. A *coordination game* is the largest graph of configurations and arcs between configurations such that:

- for each arc $C \xrightarrow{\beta} C'$ there is a collection of constraints $CC \subseteq F_\Sigma$ such that
  - for each $q \in C$ there exists a $q' \in C$ such that $(q, \phi, q') \in R$ and $\phi \in CC$,
  - for each $q' \in C'$ there exists a $q \in C$ such that $(q, \phi, q') \in R$ and $\phi \in CC$,
  - the assignment β is a solution for the conjunction of the collection of constraints, i.e. $\beta \vDash \bigwedge_{\phi \in CC} \phi$;
- there are no dead-ends that are not included in a trace.

We show some examples to give a better intuition for coordination games. The readers familiar with Petri nets may recognize similarities: a configuration is a set of places, and we shall move tokens around states. In particular, tokens may split and tokens may merge.



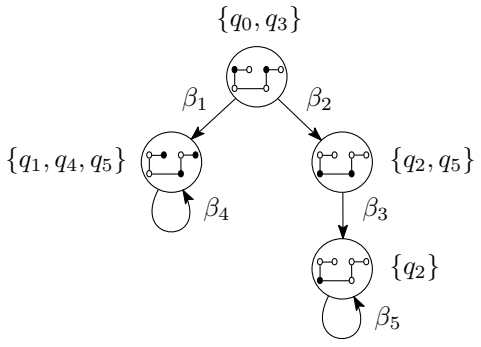
Example 9. We revisit the composition of Example 5, as depicted above. The point is that the composed constraint automaton has $\{A,B,C\}$ as ports and has a hidden port M. The resulting constraint automaton was: $Q = \{q_0, q_1\}$, $I = \{q_0, q_1\}$, $P = \{A,B,C\}$, $R = \{(q_0, A \neq * \wedge M \neq *, q_0), (q_0, A = * \wedge M = *, q_0), (q_1, A = M \wedge M = C, q_1)\}$. The game semantics consists of the initial state, and for each assignment an arc: $\{q_0, q_1\} \xrightarrow{\beta} \{q_0, q_1\}$. The assignment is a function from ports P to values. Suppose we work with the signature of $\text{Nat} = \{*, 0, 1, 2, \ldots\}$ for port B and C, and $\text{Signal} = \{*, 0\}$ for port A. Consider the example assignment $A \mapsto 0, B \mapsto 1, C \mapsto 1$. We have two possible transitions for q_0, but only one is satisfiable: $A \neq * \wedge M \neq *$ since $A \mapsto 0$. We have a single transition for q_0: $A = M \wedge M = C$. We collect both constraints in a conjunction (CC): $A \neq * \wedge M \neq * \wedge A = M \wedge M = C$. The assignment is a solution of our collected constraint, since $A \mapsto 0, B \mapsto 1, C \mapsto 1, M \mapsto 1$ is a model. ■



Example 10. Consider the constraint automaton given above. We shall draw a small part of the game graph, and reason about the constraints later. We start out in the initial configuration: $\{q_0, q_3\}$. We consider:

1. *Splitting*. There exists some β_1, and we have $\{q_0, q_3\} \xrightarrow{\beta_1} \{q_1, q_4, q_5\}$. Reasoning backwards, we have that $\beta_1 \vDash \phi_3$ (since q_3 split and moved to q_4) and $\beta_1 \vDash \phi_4$ (since q_3 split and moved to q_5) and $\beta_1 \vDash \phi_5$ (since q_0 moved to q_1). The point here is that q_3 splits into two states: q_4 and q_5. Reasoning forwards, we must have that $\beta_1 \nvDash \phi_1$ (since q_0 did not split and move to q_2).
2. *Moving*. There exists some β_2, and we have $\{q_0, q_3\} \xrightarrow{\beta_2} \{q_2, q_5\}$. Reasoning backwards, we have that $\beta_2 \vDash \phi_1$ (since q_0 moved to q_2) and $\beta_2 \vDash \phi_4$ (since q_3 moved to q_5). Reasoning forwards, we must have that $\beta_2 \nvDash \phi_3$ (since q_3 did not split and move to q_4) and $\beta_2 \nvDash \phi_5$ (since q_0 did not split and move to q_1).
3. *Merging*. There exists some β_3, and we have $\{q_2, q_5\} \xrightarrow{\beta_3} \{q_2\}$. Reasoning backwards, we have that $\beta_3 \vDash \phi_2$ (since q_2 stayed at q_2) and $\beta_3 \vDash \phi_6$ (since q_5 moved to q_2). Reasoning forwards, we must have that $\beta_3 \nvDash \phi_*$ (since q_5 did not split and stay at q_5).
4. *Looping*. Now there exists β_4 such that $\{q_1, q_4, q_5\} \xrightarrow{\beta_4} \{q_1, q_4, q_5\}$ by having $\beta_4 \vDash \phi_*$ and $\beta_4 \nvDash \phi_6$. There also exists β_5 such that $\{q_2\} \xrightarrow{\beta_5} \{q_2\}$ by having $\beta_5 \vDash \phi_2$. These form cycles.

These five arcs are shown in the game graph below. The small pictures within each configuration represent the set of states by marking them. ■



Example 11. We now consider how dead-ends are excluded. Suppose that ϕ_* and ϕ_2 are always exclusive: at any time, either ϕ_* holds or ϕ_2 holds, but never both. In the previous example, we figured that $\beta_1 \nvDash \phi_1$. Let's explore a similar assignment β_6 without that condition. Thus, suppose there is an arc $\{q_0, q_3\} \xrightarrow{\beta_6} \{q_1, q_2, q_4, q_5\}$. Then $\beta_6 \vDash \phi_3$ and $\beta_6 \vDash \phi_4$ and $\beta_6 \vDash \phi_5$ as before (this moves q_0 to

q_1, and q_3 splits and moves to q_4 and q_5). But now we also have $\beta_6 \vDash \phi_1$ (and q_0 splits and moves to q_2).

In the resulting configuration, $\{q_1, q_2, q_4, q_5\}$, we have that the assignment γ with both $\gamma \vDash \phi_*$ and $\gamma \vDash \phi_2$ is not possible, by our assumption that ϕ_* and ϕ_2 are exclusive. Note, it is not even possible to take $\{q_1, q_2, q_4, q_5\} \to \{q_1, q_2, q_4\}$: although q_5 can go to q_2 by taking ϕ_6, we must also argue why q_1 and q_4 remain in place (and thus that ϕ_* must hold) and that q_2 remains in place (and thus that ϕ_2 must hold). Since these two conditions are exclusive, it is not even possible to take this arc. Hence, $\{q_1, q_2, q_4, q_5\}$ has no outgoing arcs. By definition of coordination games there are no dead-ends. Hence, $\{q_0, q_3\} \xrightarrow{\beta_6} \{q_1, q_2, q_4, q_5\}$ is also not an arc. ■

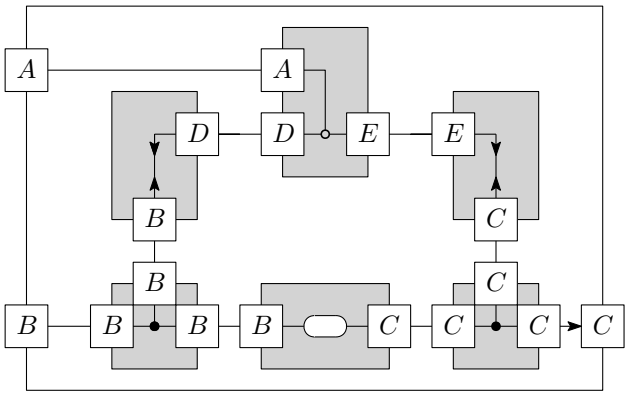
Our semantics is closely related to zero-safe Petri nets. A state is a place that holds at most one token. We synchronously move all tokens; if one of them is stuck, all others remain stuck too. Only a single transition may be taken within one round. For a detailed comparison of Reo and zero-safe Petri nets, see Clarke [5]. The precise relation between Clarke's definition and ours is out of scope of this paper.

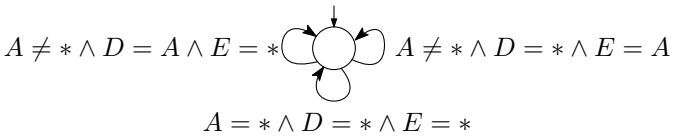
The relation between our semantics and speculative execution is that of refinement: speculative execution can be implemented by visiting multiples assignments non-determinisitically. Each branch might progress in rounds and sees multiple observations. Once we encounter a dead-end state, the branch is pruned: either we backtrack and search for the last consistent configuration, or we discard a parallel world. If the implementation implements a non-empty coordination game, there must exists a trace. Thus, at least one of the branches is guaranteed to be consistent.

There are two operations on constraint automata that simplify: a constraint automaton can be determinized, and a constraint automaton can be minimized.

The determinization of a constraint automaton follows quite simply from the coordination game graph. Each assignment corresponds to a unique clause. We take as states the configurations of the game graph, and for each arc there is a transition labeled by the clause corresponding to the assignment of the arc. In this way, we can also speak of an inconsistent state to mean the same thing as the last configuration of the largest dead-ends not included in a trace.

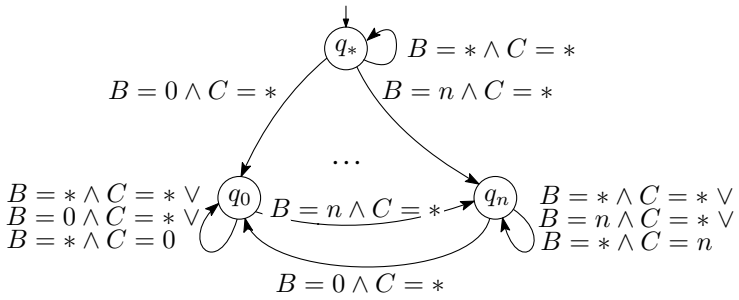
We also have the notion of minimization of constraint automata. Minimizing constraint automata is useful, when constructing larger systems. Two constraint automata are equivalent if they have the same coordination game. We do not formally define minimization, but give an example:





Example 12. We consider the minimization of a controlled memory cell. We have three boundary ports: A, B, C and six components: a router, two synchronous drains, a variable, and two replicators. Replicators are easily simplified: we simply name all output ports by the same input port. The constraint automaton for router is depicted. The constraint automata for synchronous drain and variable have been given before (but have to be renamed). The initial configuration is as follows: the router and synchronous drains have only one state, the variable is initially empty.

We argue that by the construction of the router, if A fires, then either B fires or C fires but never both. This insight can be applied to simplify the constraint automaton of the variable, any constraint that violates this condition is removed:



Additionally, for the overall constraint automaton for our composition, we add the constraint $(A \neq * \wedge (B \neq * \vee C \neq *)) \vee (A = * \wedge B = * \wedge C = *)$. ■

3 Syntax

Now that we have hopefully gained an intuitive understandig of components and their game semantics, we shall in this section 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 its ports.
2. Interfaces describes 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, and how to check whether a component is well-typed.

The result formalizes our earlier notation for diagrams of components.

3.1 Compositions

In this section we first introduce compositions and how to link components together. We shall then consider well-formedness and well-typedness of compositions.

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

Definition 13. 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 α a type annotation. The type annotation α 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, it is a qualified reference; if we point to a boundary port of a composite component, it is an unqualified references.

Definition 14. A *composition* is as given by the following grammar:

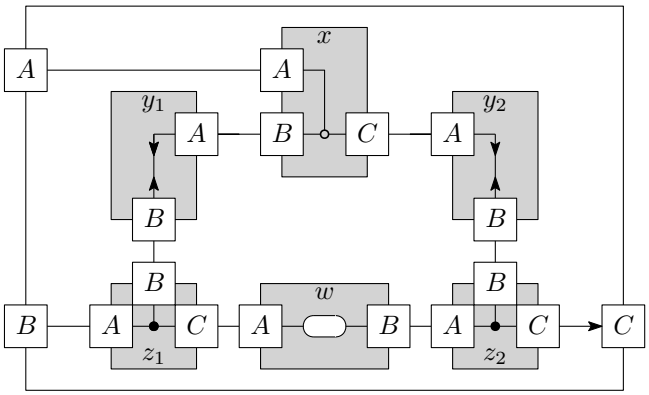
$$c, d, e ::= x \mid (c \parallel d) \mid (c)^p_q$$

such that the following laws hold:

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

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

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



Example 15. In the picture above 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 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: $(x \parallel y_1 \parallel y_2 \parallel z_1 \parallel z_2 \parallel w)$ forms all components without their identifications. Since parallel composition is associative we do not write parentheses. The composition with all identifications is:

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

■

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 the natural order of instance variables, removing duplicates. The surface identifications also form a list of pairs of references. We sort this list according to the lexicographic orders of pairs of references, removing duplicates. The result has the shape $((((x \parallel (\ldots \parallel z)))^{p}_{q})\ldots)^{r}_{s}$ such that instances are ordered, and references are lexicographically ordered.

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

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

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

Definition 16. A composition is *well-formed* if:

- every qualified reference that is qualified by an instance must have that instance occurring 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 a well-formed compositions are: $(x)_Y^{x.X}$ denotes the composition of instance variable x of which its external port $x.X$ is identified to the internal port Y at the boundary of our composition.

Here are some negative examples. $(y)_X^{x.Y}$ 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).

3.2 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 17. 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 left-hand side of an interface are called *input* port variables and the right-hand side are called *output* port variables. The dual of an interface swaps input and output.

We may also lift union to interfaces. Let $\Delta_1,\Delta_2,\Theta_1,\Theta_2$ be sets of references. Given two interfaces $U = \langle\Delta_1 \mid \Theta_1\rangle$ and $V = \langle\Delta_2 \mid \Theta_2\rangle$, then by $U \cup V$ we mean pairwise union of the two 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 interface U. Let a typing context be a set of typing judgments, as denoted by Γ. We define the relation $\vdash$ between typing contexts and typed compositions, as given by Figure 9.

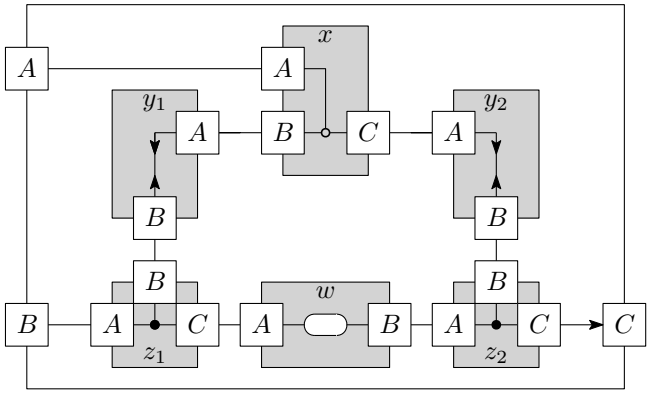
$$\frac{}{\Gamma, x :: U \vdash x : x.U} \qquad \frac{\Gamma \vdash c : \langle \Delta \mid \Theta \rangle}{\Gamma \vdash (c)^{X}_{Y} : \langle X^{\alpha}, \Delta \mid \Theta, Y^{\alpha} \rangle} \qquad \frac{\Gamma \vdash c : \langle x.X^{\alpha}, \Delta \mid \Theta, y.Y^{\alpha} \rangle}{\Gamma \vdash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

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

Figure 9: Typing rules for compositions.

Here, we take $U \cup V$ to mean the pairwise union of two interfaces. 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, Δ it holds that $p \notin \Delta$, and for Θ, q it holds that $q \notin \Theta$. We have not written type annotations in compositions for brevity: they are the same as the annotation given in the interface.

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



Example 18. In the above picture, consider that the interface of the router is $\langle B, C \mid A \rangle$, that of the drain is $\langle \mid A, B \rangle$, that of the replicator is $\langle B, C \mid A \rangle$ and that of the variable is $\langle B \mid A \rangle$. That these interfaces are reversed becomes obvious in the next section. 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 could 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 $\langle A \mid B, C \rangle$ for the composition with all identifications.

3.3 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 19. A *component* is as given by the following grammar:

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

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

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

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

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

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

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

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

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

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

We see an example of simplification and substitution: $\textbf{let}\ x = ((y)^{y.X}_A)^B_{y.Y}\ \textbf{in}\ ((x)^{x.A}_Z)^W_{x.B}$. 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

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

Figure 10: Typing rules for composite components.

first step results in the qualification of references: $((y)^{y.X}_{A})^{B}_{y.Y}$ becomes $((y)^{y.X}_{x.A})^{x.B}_{y.Y}$. The second step results in $((((y)^{y.X}_{x.A})^{x.B}_{y.Y})^{x.A}_{Z})^{W}_{x.B}$ where x is replaced by $((y)^{y.X}_{x.A})^{x.B}_{y.Y}$. The last step removes identifications by linking through: we link $y.X$ to Z via $x.A$ and remove $x.A$ to obtain $(((y)^{x.B}_{y.Y})^{y.X}_{Z})^{W}_{x.B}$, and we link $y.Y$ to W via $x.B$ and remove $x.B$ to obtain $((y)^{y.X}_{Z})^{W}_{y.Y}$.

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

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 Γ denote a set of typing judgments. We define a relation between typing contexts and a single typing judgment of a composite component, denoted $\Gamma \vdash C :: U$, as given by Figure 10. All the rules are remarkable:

- in the first rule, recall that the difference between $C :: U$ and $c : U$ is that the former judges only unqualified interfaces, whereas the latter judges arbitrary interfaces. Hence, for a composition to become a composite component, it is required to have an unqualified interface. This is possible by identifying all qualified references.
- The second rule is a family of rules, one rule for each primitive component R with unqualified interface U. We shall admit this rule for each primitive component together with U as its unqualified interface.
- The third 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 Γ, Δ we mean that Γ and Δ 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*.

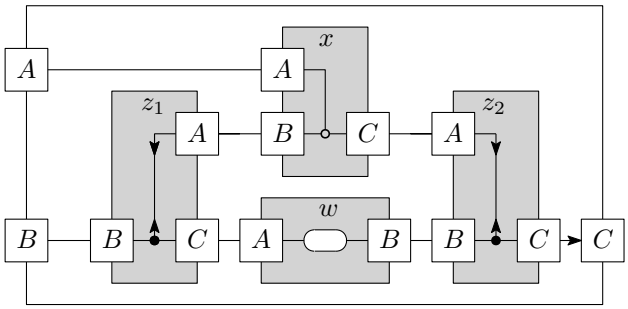
Example 20. We now take a component from before and show how it is normalized. The first thing to fix are the primitive components, of router $:: \langle A \mid B, C \rangle$, valve $:: \langle A \mid B, C \rangle$ and variable $:: \langle A \mid B \rangle$. The component valve is formed by composing replicator $:: \langle A \mid B, C \rangle$ and drain $:: \langle A, B \mid \rangle$ as:

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

this forms the following derivation:

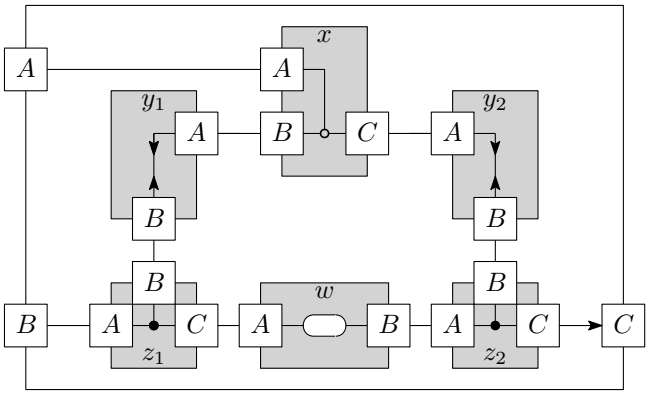
$$\dfrac{\vdash \textbf{new replicator} :: \langle A \mid B,C\rangle \qquad \dfrac{\vdash \textbf{new drain} :: \langle A,B \mid\rangle \qquad \dfrac{\vdots}{x :: \langle B,C \mid A\rangle, y :: \langle\mid A,B\rangle \vdash c :: \langle A \mid B,C\rangle}}{x :: \langle B,C \mid A\rangle \vdash \textbf{let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

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 to the unqualified interface $\langle A \mid B,C\rangle$. Let this closed component be called C. We can form the component of the controlled memory by reusing our earlier component, as follows:



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

After normalization and renaming we obtain the component shown below:



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

4 Semantics

This section describes a particular many-sorted logic and its standard interpretations, which can be employed to establish properties of components. We encode,

in our logic, the game semantics of components. Properties of the semantics of component can now be defined as classes of components: to establish that a component has a property, we prove that a component is member of the corresponding class.

1. We first lay down the mathematical preliminaries required in the rest of this section: 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 streams of assignments.
3. Next we show how to encode our game semantics of primitive components using frame conditions. This establishes the behavior of primitive components.
4. We lift our encoding to composite components that are constructed using the syntax of previous section.

4.1 Data Streams

Data types are algebraic structures with countable carrier sets. Streams are used to model the flow of data. We consider stream differential equations, stream equality, and data streams. We have a strict distinction between data types and (data) streams. All elements of a data type can be enumerated. Contrastingly, not all streams can be enumerated.

By $\mathbb{N}$ we denote the set of *natural numbers* $0,1,2,3,\ldots$ We use variables k,l,m,n to stand for arbitrary natural numbers, unless mentioned otherwise.

Data is algebraically structured. Let a *data type* be a structure $(D,*)$ where D is a countable carrier set and $* \in D$ is a designated constant. We shall speak of *data elements* to be 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. Null values are commonly used in object-oriented programming languages. The data types we define here are classes of immutable value objects in the object-orientation paradigm.

Definition 21. Data is encoded by natural numbers. Every data type $(D,*)$ is associated with an *encoding* function $e : D \to \mathbb{N}$ and *decoding* function $d : \mathbb{N} \to D$, such that:

- $e(*) = 0$ encodes the null value $*$,
- e is injective, i.e. every element has a unique encoding,
- $d(e(x)) = x$ for every $x \in D$ and $d(y) = *$ for every $y \notin \text{image}(e)$.

A data type is *finite* if there are more natural numbers than elements; or, equivalently, that e is not surjective. By definition, elements not in the image of e are mapped by d to $*$. A data type is *infinite* if e is surjective. Consequently for infinite data types, e is a bijection, d is the inverse of e, and d is bijective.

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

Let $\mathbb{T}$ be a countably infinite set of data types. We shall write $\alpha,\beta,\ldots$ to denote data types. As convention, we write $a \in \alpha$ to mean some element a of the carrier set. In the sequel, let $\alpha = (A,*)$ and $\beta = (B,*)$ be arbitrary data types with carriers A and B. The set $\mathbb{T}$ is closed under the operations described below.

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

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

Alternatively, one thinks of $\alpha + \beta$ as having as carrier the disjoint union $A \uplus B$ modulo the equivalence $\lfloor * \rfloor \equiv \lceil * \rceil$, setting $* := \lfloor * \rfloor$. One thinks of $\alpha \times \beta$ as having as carrier the Cartesian product $A \times B$ modulo the equivalences $(*,b) \equiv (a,*) \equiv (*,*)$, setting $* := (*,*)$. Similarly, one thinks of $\alpha \otimes \beta$ as having as carrier $A \times B$ modulo $(*,*) \equiv *$. The difference between $\alpha \times \beta$ and $\alpha \otimes \beta$ is most obvious here: $\alpha \times \beta$ does not contain tuples with $*$, wheras $\alpha \otimes \beta$ may contain tuples with $*$.

A function from α to β is a function between their carrier sets, $f : A \to B$ such that $f(*) = *$. In general, n-ary functions from $\alpha_1, \ldots, \alpha_n$ to β is a function between their carrier sets $f : A_1 \to \ldots \to A_n \to B$ such that $f(\ldots, *, \ldots) = *$, that is, if any argument is $*$ then the result is $*$. Two data types are equivalent if a bijection exists between the two data types, and by $\cong$ we denote the equivalence relation between data types. $(\mathbb{T}, +, \times, 0, 1)$ forms a semi-ring with respect to $\cong$.

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

Let a stream be a function from natural numbers to natural numbers. We denote streams by the Greek letters $\sigma, \tau, \ldots$. Intuitively, one thinks of streams as an enumeration. We may define streams directly as a function $\sigma : \mathbb{N} \to \mathbb{N}$. Alternatively, we may define streams as a stream differential equation. See [13, 14] for an elementary introduction.

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

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

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

There are more streams than natural numbers. The argument is a variation of Cantor's diagonalization argument that there exists more real numbers than natural numbers. There are at least as many streams as there are natural numbers;

for each natural number n we can construct the stream $[n]$.

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

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

Equality of streams is established by bisimilarity [3]. A relation R on two streams is called a (stream) bisimulation if for all $(\sigma, \tau) \in R$,

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

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

Data streams are functions from naturals to data types, say $\sigma : \mathbb{N} \to \alpha$ for some data type α. Data streams are streams. This is established by using the associated encoding and decoding function for data type α. A given data stream $\sigma : \mathbb{N} \to \alpha$ is a stream $\tau : \mathbb{N} \to \mathbb{N}$ such that $\tau(n) = e(\sigma(n))$ and $\sigma(n) = d(\tau(n))$.

4.2 Logical Framework

We now consider the many-sorted first-order logic with equality which we use to encode coordination game semantics we have seen before in Section 2.3. Additionally, a restriction of our language is a many-sorted zeroth-order logic with equality, that is used to form constraints in constraint automata of Section 2.2. The general structure of this section is, and follows much from [6]:

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 and satisfiability.

The intention of our 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 in these tools is out of scope of this paper.

We start with signatures, and we define two subclasses of signatures: basic signatures and stream signatures. Stream signatures are a subclass of a basic signatures. Our basic and stream signatures may contain more non-logical symbols than those given here: we only specify the minimal requirements. This intends to capture openness and extensibility of our framework.

Definition 22. $\Sigma = (S, F, P)$ is a *signature* consisting of the following data:

- S is a set of *sorts*,
- F is a set of function symbols,
- P is a set of predicate symbols,
- each function symbol has an associated non-empty arity,
- each predicate symbol has an associated arity.

An arity is a list of sorts $\langle s_1, \ldots, s_n \rangle$.

Definition 23. A signature $\Sigma = (S, F, P)$ is a *basic signature* if:

- for each data type α there is a distinct sort $\alpha \in S$,

- for each data type α, $*_\alpha \in F$ with arity $\langle\alpha\rangle$,
- for each data element $d \in \alpha$, $d_\alpha \in F$ with arity $\langle\alpha\rangle$,
- $\bot \in P$ with arity $\langle\rangle$,
- for each data type α, $=_\alpha \in P$ with arity $\langle\alpha,\alpha\rangle$.

Moreover, Σ is an *stream signature* if, additionally:

- there is a distinct sort $\mathbb{N} \in S$,
- for each data type α there is a distinct sort $(\mathbb{N} \to \alpha) \in S$,
- $0 \in F$, $S \in F$, and $+ \in F$, $- \in F$, $\times \in F$ 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 α, $\mathrm{at}_\alpha \in F$ with arity $\langle(\mathbb{N} \to \alpha),\mathbb{N},\alpha\rangle$,
- $\leq \in P$ with arity $\langle\mathbb{N},\mathbb{N}\rangle$.

We 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 24. 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 25. 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 $\exists x^s, \forall x^s$.

Moreover, a formula without quantifiers is called a *zeroth-order formula*.

Next, we define interpretations, and two subclasses of interpretations. Both are called standard interpretations: but one takes a basic signature, the other takes a stream signature.

Definition 26. An *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 $s^\mathcal{M}$,
- 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 $s^\mathcal{M}$,
- 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.

Definition 27. An interpretation $\mathcal{M}$ of a basic signature is a *standard interpretation* if:

- $\alpha^\mathcal{M}$ is the carrier set of data type α,
- $*_\alpha^\mathcal{M}$ is the null value $* \in \alpha$,
- $d_\alpha^\mathcal{M}$ is the data element $d \in \alpha$,
- $\bot^\mathcal{M}$ never holds,
- $=_\alpha^\mathcal{M}$ is equality of values of data type α.

Moreover, an interpretation $\mathcal{M}$ of a stream signature is a *standard interpretation* if furthermore:

- $\mathbb{N} \in S$ is mapped to the set of natural numbers,

- the sort $(\mathbb{N} \to \alpha) \in S$ is mapped to data streams $\mathbb{N} \to \alpha$ of data type α,
- $0, S, +, -, \times$ are interpreted as the natural number zero, the successor function, and the arithmetical functions of plus, monus[2] and times,
- at_α is interpreted as a lookup function $\text{at}_\alpha^{\mathcal{M}} : (\mathbb{N} \to \alpha) \times \mathbb{N} \to \alpha$ such that $\text{at}_\alpha^{\mathcal{M}}(\sigma)(n) = \sigma(n)$,
- $\leq^{\mathcal{M}}$ is the relation of less than or equals between naturals.

Next, we define the notion of assignment. This is necessary, since not every domain element has a corresponding term. We fix some (basic or extended) signature Σ, and let $\mathcal{M}$ denote a fixed standard interpretation.

Definition 28. An *assignment* β 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.

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

Proposition 29. *If β_1, β_2 are two assignments and $\mathcal{M}, \beta_1 \vDash \phi$ and $\mathcal{M}, \beta_2 \vDash \phi$ then $\beta_1(x^s) = \beta_2(x^s)$ for all free variables x^s in ϕ.*

An assignment that is restricted to map only the free variables of some formula ϕ is called a *solution* for ϕ. We have two classes of formulas and terms:

Definition 30. A *constraint* is a zeroth-order Σ-formula ϕ where Σ is a basic signature. The free variables x^α of sort α of ϕ are called *ports* of data type α.

Definition 31. A *coordination protocol* is a first-order Σ-formula ϕ where Σ is a stream signature, where all free variables x^s of ϕ must be of sort $s = (\mathbb{N} \to \alpha)$ for any α. These free variables are also called *ports* of data type α.

Remark 32. We shall use a more convenient notation for at_α: let X^α be a port of sort $(\mathbb{N} \to \alpha)$ and $t^{\mathbb{N}}$ be a variable of sort $\mathbb{N}$, then the term $\text{at}_\alpha(X^\alpha, t^{\mathbb{N}})$ is written as $X(t)$. We shall treat sort annotations implicitly to prevent clutter. Additionally, we use $s < t$ for $\neg(t \leq s)$.

4.3 Coordination Protocols

In this section, we provide more intuition of coordination protocols. Formally, coordination protocols are formulas as defined in last section. Thet set of solutions of a coordination protocol is a set of tables of observations. 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. We relate our intuition back to constraint automata and coordination games, by encoding constraint automata as coordination protocols.

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

Definition 33. Given a stream signature Σ and a standard interpretation $\mathcal{M}$, the coordination protocol ϕ induces a set $\mathcal{L}(\phi) = \{\beta \mid \mathcal{M}, \beta \vDash \phi\}$ of solutions of ϕ.

Intuitively, we consider the set $\mathcal{L}(\phi)$ of solutions as a set of tables of observations. Consider these examples:

Example 34. 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.

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

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 σ and τ are data streams such that $\sigma(t) = *$ or $\tau(t) = *$ for any $t \in \mathbb{N}$.

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

■

Consider two coordination protocols ϕ and ψ that do not have any free variables in common. The protocol $\mathcal{L}(\phi)$ consists only of solutions of ϕ, and similar for $\mathcal{L}(\psi)$ and solutions of ψ. 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 35. The solutions of $\forall t.X(t) = *$ and $\forall t.Y(t) = 0$ can be glued together to form the coordination protocol:

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

■

Interestingly, given two formulas ϕ and ψ that do not share any variables, we have that $\mathcal{L}(\phi \vee \psi)$ is equivalent to $\mathcal{L}(\phi \wedge \psi)$. Neither protocol affects the other, since they do not interact through a shared port variable. As soon as they

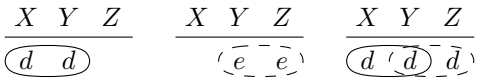


Figure 11: Intersection of frame conditions.

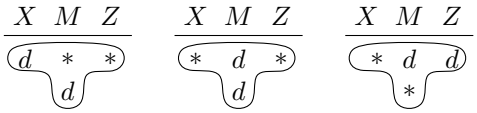


Figure 12: Three frame conditions that make up a buffer.

share a variable, the protocols might interact. Only solutions of the first protocol are retained that are consistent with the solutions of the second protocol. We illustrate our intuition using *frame conditions*.

Example 36. 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 11. 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 it must constrain every row. The intuition of universal quantification is to *slide* the frame condition over all rows.

Example 37. In Figure 11, 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 38. A frame condition that spans multiple rows is the example in Figure 12. 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 and that the memory is cleared in the next row. We now obtain the coordination protocol:

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

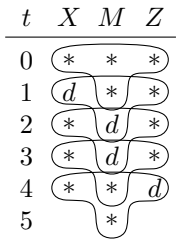


Figure 13: Example sequence of solutions, with constraints of a buffer.

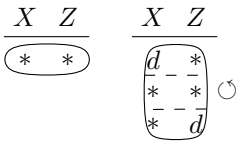


Figure 14: Alternative frame conditions that make up a buffer.

Note that we have the quantifier $\forall t.$ on the outer level to ensure that for each row, one of these three frame conditions apply. Otherwise, if we take $(\forall t \ldots.) \vee (\forall t \ldots.) \vee (\forall t \ldots.)$ as coordination protocol, we accepts streams that *only* have for all rows, the first, second, or third frame condition.

A demonstration how the three frame conditions apply to an arbitrary solution is given in Figure 13. In 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 14. 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 $*$. Note that our frame condition is still sliding here: the condition that $Z(t) = * \wedge X(t) = *$ is applied for all rows that are intermediate between the element accepted by X and then returned by Z. ■

The next step is to encode the coordination game semantics of Section 2.3 in our logical framework. Recall the definition of a constraint automaton:

Definition (See Definition 2)**.** A constraint automaton (Q, I, P, R) consists of:

1. A denumerable set of states Q, a non-empty subset $I \subseteq Q$ of initial states.
2. A finite set of ports P.
3. A transition relation $R \subseteq Q \times F_{\Sigma}^{0} \times Q$.

The main idea is to encode the set of states as a data type. We introduce a fresh hidden port variable that encodes the state of the automaton, say M^Q with a data type that corresponds to Q. For each transition $(q, \phi, q') \in R$ we add a clause that checks whether the value of M corresponds to the state q and that the constraint ϕ (lifted to ports for coordination protocols) holds, and updates the next value of M corresponding to the state q'. The example given above is the encoding of a buffer, as seen in Example 6. We refer the reader to Appendix A for some encodings of primitive components.

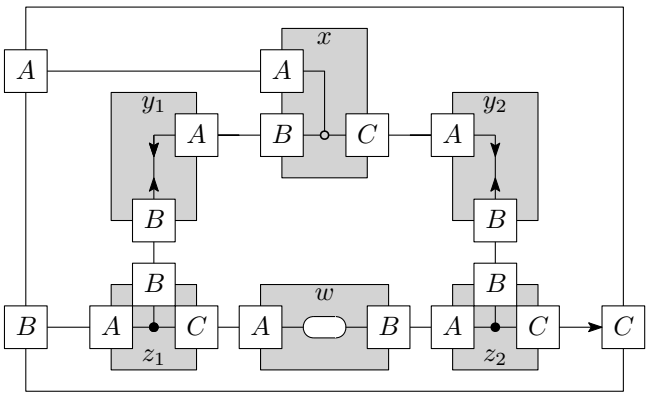
We remark that not all constraint automata can be mechanically encoded as a coordination protocol, and requires some intelligence: for example, the automaton that has an infinite number of states and a unique transition per state cannot be encoded, for it requires an infinite formula. Constraint automata with finite number of states can always be encoded, and the constraint automata with memory of Jongmans [9] seems to be encodeable without too much effort by encoding the possible values of memory cells as states. Doing so here is out of scope of this paper.

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 39. A *component* $U\phi$ consists of the following data:

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

The free variables of the coordination protocol ϕ are all ports assigned in solutions; the interface marks which ports are not hidden. The coordination protocol for composite components is constructed as follows. We assume the composite component is normalized. We work towards a conjunction of all coordination protocols of primitive components. To do so we qualify each reference by its instance variable, to ensure uniqueness of reference. We then take a conjunction of all qualified component formulas. For each identification of references p and q we add an equation $\forall t.p(t) = q(t)$ to the conjunction. The result is a coordination protocol for the composite component. All qualified references are hidden ports, unqualified references that occur as source are inputs, unqualified references that occur as sink are outputs.



Example 40. (See Examples 12, 15, 18, 20) We have the following primitive components, see Appendix A for a reference:

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

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

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

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

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:

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

5 Logical Analysis

Now that we have a solid logical foundation, we may define and analyze properties of coordination protocols. The purpose of this section is to give a broad set of properties that are interesting for analysing concurrent and distributed systems. We give an intuition for properties based on examples that are related to the running example of Section 2.1.

- **Independence** of a component indicates that it may cooperate with other components. A component is independent if it can be 'paused' and 'resumed' in any state. Independence is essential for composition: consider, for example, combining a slow-running component with a fast-running component. If these components need to communicate, the fast-running component needs to slow down to match the slow-running component. Independence captures the property that a component can be arbitrarily slowed down. Independence is preserved under composition.
- **Progress** of a component indicates that it is productive. A productive component always guarantees that eventually an actual observation will be made. The complement of progress is termination: a component is in deadlock if no further actual observations can be made, and a component terminates if eventually a deadlock is reached. Although a component with

progress can be independent, it cannot be terminating. Thus, a composition of a component that has the progress property with a component that terminates may be inconsistent.

- **Fairness** is a more refined notion of progress. Although a component can have progress, it does not mean that it is fair with respect to some of its outputs. For example a router may always progress by output to a single port, thereby always inhibiting the other port. This violates fairness between the two ports. Two ports are fair if the firing of one implies the eventual firing of the other. Fairness could be refined even further by balancing.
- **Synchronicity** of components model atomicity of its observations, and is related to the intuition of a transaction: either all ports fire (commit) or none of them fire (rollback). An asynchronous component fires one port exclusively and other other ports do not fire (critical section). This property can also be checked for a restriction of ports of a component. Sychronous behavior spreads through compositions: composing two synchronous components makes the composition also behave synchronous.
- **Instantaneous**: a component is instantaneous if its observations all happen in a single instant. An instantaneous component does not relate inputs and outputs over time. Instantaneous is preserved under composition.
- **Linearity** means that input is never descarded or duplicated, and output never appears out of nowhere. Every output of a component can be traced back to precisely one of its input at a unique time, and every of its input is uniquely related to an output. Duplication in space or over time is prohibited. Linearity is closely related to reversibility of a computation, and is preserved under composition.
- **Causality**: a causal component relates each of its outputs to inputs that have happened in the past. Causality does not imply linearity: a component that duplicates elements in space or over time is not linear but may still be causal. Causality implies a flow from input to output. An acausal component has all its output in the past related to future input, as if its output was a true prediction. Causality is closely related to speculative execution. Causality is preserved under composition.
- **Determinism** of a component is that inputs fully determines output behavior. In particular, determinism requires that at no point in time the same input eventually leads to different outputs. A deterministic component can determine its next state only by observing its input. Deterministic and instantaneous implies functionality.
- **Equivalence** of two components represent an equivalence in behavior. Equal components must react equally to all possible inputs and must produce the same data. Equivalence respects independence, in the sense that two independent components be equivalent even though one component takes more time than the other. However, equivalence discriminates the property of progress: two components that react equally and produce the same actual observations are different if one has the possibility to deadlock and the other not.
- **Abstraction** of a component relates it to another component with different synchronous behavior. A synchronous behavior abstracts the irrelevance of the order of observing ports. A component is an abstraction of another component if the data of inputs and outputs are equivalent, but the timing and synchronicity is not regarded. Abstraction does not preserve progress.

Some of these properties are formally defined in the following sections. Certain properties are expressed on a semantic level, others are definable in the framework given in Section 4.2. In Appendix A, a reference of standard components is given and an overview of components and their properties.

5.1 Independence

Independence of a component is intuitively the possibility to stretch observations over time. Formally, if $\mathcal{L}(\phi)$ is the set of assignments of streams, we can understand independence as a closure condition on $\mathcal{L}(\phi)$.

Let $\beta \in \mathcal{L}(\phi)$ be a solution. Recall that a solution is a set of (infinite) tables of observations. We define two operations on such tables: adding a row consisting only of $*$, and removing a row consisting only of $*$. We restrict these operations to apply only to certain rows: adding may only happen before a row that actually contains data, and only rows that do not contain any data may be removed.

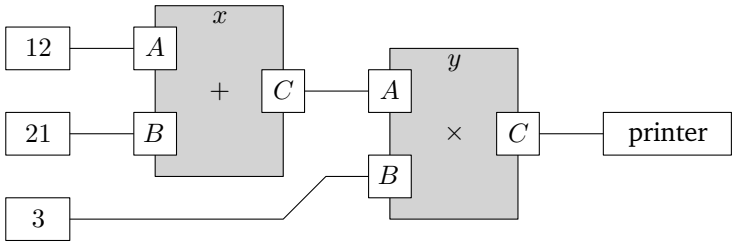
Let t be a row, we insert a new row in front. The result of adding a row in β is denoted $\text{add}(\beta,t)$, which itself is a solution. Adding a row $\text{add}(\beta,t)$ is only defined if $\beta(Y)(t) \neq *$ for some Y:

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

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

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

Our closure is under insertion and removal of rows that consist only of $*$. We call $\mathcal{L}(\phi)$ independent if for each solution $\beta \in \mathcal{L}(\phi)$, we have that, if defined, $\text{add}(\beta,t) \in \mathcal{L}(\phi)$ and $\text{remove}(\beta,t) \in \mathcal{L}(\phi)$ for all t.



Example 41. Consider our very first example. We have two independent components: say addition and multiplication. Independence means that multiplication can be delayed until addition is done. More precisely, suppose that these tables are acceptable for addition and multiplication, respectively:

t	$x.A$	$x.B$	$x.C$	t	$y.A$	$y.B$	$y.C$
0	$*$	$*$	$*$	0	33	3	$*$
1	12	21	$*$	1	$*$	$*$	$*$
2	$*$	$*$	33	2	$*$	$*$	99
3	$*$	$*$	$*$	3	$*$	$*$	$*$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

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

t	$x.A$	$x.B$	$x.C$	t	$y.A$	$y.B$	$y.C$
0	12	21	$*$	0	$*$	$*$	$*$
1	$*$	$*$	33	1	33	3	$*$
2	$*$	$*$	$*$	2	$*$	$*$	99
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Note how it is not possible to remove a row with data. Remark that for the addition component, we cannot insert a row before row 2, since row 2 does not contain any data. ■

Independence is preserved by composition. Suppose we have to independent components. Both may fire independently of eachother. The resulting behavior is independent: we can always remove rows with all $*$ and still obtain a valid table, by removing the corresponding rows of the composed components. We can also insert rows with all $*$, similarly by doing so in the two underlying tables.

We remark that independence prevents real-time clocks. Suppose there exists a component that for each row t outputs precisely the row index of t. Then it cannot be independent since we can no longer stretch the table, and still obtain an acceptable table.

5.2 Progress

TODO

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

Synchronicity is defined on a set of ports $P = \{X_1, \ldots, X_n\}$. A component is synchronous if all its input and output ports are synchronous. P is *synchronous* if always either all ports fire, or no ports fire. This is expressed by the following

formula:

$$\forall t.((X_1(t) = * \wedge \cdots \wedge X_n(t) = *) \vee \\ (X_1(t) \neq * \wedge \cdots \wedge X_n(t) \neq *))$$

We have that P is *asynchronous* if always at most one port fires, or no ports fire. This is expressed by the following formula:

$$\begin{aligned} \forall t.((X_1(t) \neq * \rightarrow \quad & X_2 = * \wedge \cdots \wedge X_{n-1}(t) = * \wedge X_n(t) = *) \wedge \\ (X_2(t) \neq * \rightarrow & X_1 = * \wedge \quad \cdots \wedge X_{n-1}(t) = * \wedge X_n(t) = *) \wedge \\ & \vdots \\ (X_{n-1}(t) \neq * \rightarrow & X_1 = * \wedge X_2 = * \wedge \cdots \quad \wedge X_n(t) = *) \wedge \\ (X_n(t) \neq * \rightarrow & X_1 = * \wedge X_2 = * \wedge \cdots \wedge X_{n-1}(t) = * \quad)) \end{aligned}$$

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.

If a set of ports is both synchronous and asynchronous, its ports never fire. Suppose one fires, then all others must fire (synchronous) and all others must not fire (asynchronous): since this is inconsistent, never any port fires.

We remark that non-synchronous and asynchronous are different. A set of ports is non-synchronous if the synchronous property does not hold. For example, a component with four ports where two ports can fire together. This component is not asynchronous, since its ports do not fire exclusively.

Example 42. The best examples of synchronous and asynchronous components are, by definition: the synchronous drain and the asynchronous drain.

Component:	drain(α, β)
Interface:	$\langle A^\alpha, B^\beta \mid \rangle$
Protocol:	$\forall t.((A(t) = * \wedge B(t) = *) \vee (A(t) \neq * \wedge B(t) \neq *))$

Component:	adrain(α, β)
Interface:	$\langle A^\alpha, B^\beta \mid \rangle$
Protocol:	$\forall t.((A(t) \neq * \rightarrow B(t) = *) \wedge (B(t) \neq * \rightarrow A(t) = *))$

These components have an input constraint. The synchronous drain only fires both ports at the same time. The asynchronous drain fires at most one port. ■

Example 43. An example of an asynchronous component is a buffer. A buffer never fires both its input and output ports. ■

A composition of two synchronous components, linking one to the other, is synchronous. By linking the components, they have a shared port. If that 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, resulting in all ports to fire. Therefore, composition with identification preserves synchronization. Identification is essential, since otherwise the composed components may fire independently.

5.4 Instantaneous

Transportation involves the movement of data elements. Data elements typically move from input ports to output ports, that is, in space. If this movement happens without progression of time, we say data is transported instantaneously. We also consider non-instantaneous components, that may move data elements in time.

5.5 Linearity

TODO

5.6 Causality

TODO

6 Discussion

Snapshots are expensive, synchronization is also expensive: a more logical approach allows us to identify which parts of a composition can be optimized to minimize such costs.

Related work (Manfred Broy, DClarke Int.LinearLogic)
Synthesis problem
Reconfiguration problem

7 Conclusion

TODO

Acknowledgements

References

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

[2] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in reo by constraint automata. *Science of computer programming*, 61(2):75–113, 2006.

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

[4] Anasua Bhowmik and Manoj Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 99–108. ACM, 2002.

[5] Dave Clarke. Coordination: Reo, nets, and logic. In *Formal Methods for Components and Objects*, pages 226–256, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[6] Herbert Enderton and Herbert B Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

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

[8] Red Hat. Performance considerations for l1 terminal fault. `https://access.redhat.com/security/vulnerabilities/L1TF-perf`. (Accessed 24-Aug-2018).

[9] Sung-Shik Theodorus Quirinus Jongmans. *Automata-theoretic protocol programming*. PhD thesis, Centrum Wiskunde & Informatica (CWI), Faculty of Science, Leiden University, 2016.

[10] Sung-Shik TQ Jongmans and Farhad Arbab. Overview of thirty semantic formalisms for reo. *Scientific Annals of Computer Science*, 22(1), 2012.

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

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

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

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

[15] Andrew S Tanenbaum. Structured computer organization (6th edition). 2013.

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

[17] Augustus K 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.

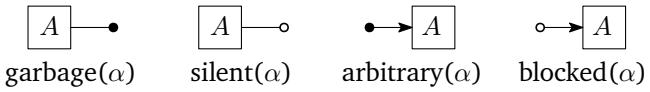


Figure 15: Endpoints

A Standard Components

In this section we will give a number of standard components. The components considered in this section are not exhaustive: others do exists.

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

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 Section 4). 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.

A.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 15.

Component:	garbage(α)
Interface:	$\langle A^\alpha \mid \rangle$
Protocol:	$\top$

A garbage can accept any data and throws it away. It is wasteful: any information it accepts is destroyed. Every stream of observations is acceptable. During the

coordination game, a garbage accepts any constraint by the environment on its input port.

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

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

Component:	arbitrary(α)
Interface:	$\langle \mid A^{\alpha} \rangle$
Protocol:	$\top$

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.

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

The force component guarantees progress for its input, by always asserting that the port eventually fires. This introduces inconsistency in case of a deadlock.

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

A.2 Channels

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

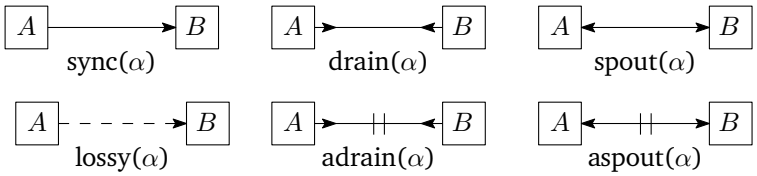


Figure 16: Channels

A	B
d	d

A	B
d	$*$

Table 2: Example observations of lossy

port and one output port, channels with two input ports, and channels with two output ports. The channels introduced here are depicted in Figure 16.

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

Component:	sync(α)
Interface:	$\langle A^\alpha \mid B^\alpha \rangle$
Protocol:	$\forall t.(A(t) = B(t))$

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

Component:	lossy(α)
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 2. The left table shows that a data element d from input A is transported instantaneously to output B. The right table shows that a data element d from input A is lost. Both of these subsequences of behavior are acceptable, for any data element d. Note that these two cases correspond to the disjunction in the protocol of lossy.

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

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

Component:	drain(α, β)
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 also loses all its input for the purpose of synchronization. At most one input port passes data instantaneously. Passing data is not related.

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

Component:	adrain(α,β)
Interface:	$\langle A^\alpha, B^\beta \mid \rangle$
Protocol:	$\forall t.((A(t) \neq * \to B(t) = *) \wedge (B(t) \neq * \to A(t) = *))$

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

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

Component:	spout(α,β)
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 is the dual of an asynchronous drain: it also generates data, but never at the same time at both ports.

Component:	aspout(α,β)
Interface:	$\langle \mid A^\alpha, B^\beta \rangle$
Protocol:	$\forall t.((A(t) \neq * \to B(t) = *) \wedge (B(t) \neq * \to A(t) = *))$

A.3 Buffers

A channel that deserves special attention is the buffer, which transports non-instantaneously. In this section we consider a buffer, and three variants. These components are depicted in Figure 17.

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

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

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

The output of a buffer must remain silent until some input element passes to memory. A buffers remembers its element indefinitely while the output remains

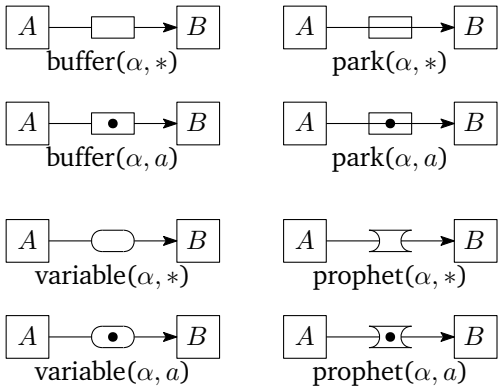


Figure 17: Buffers and three variants: the dot inside marks full memory.

A	M	B
d	$*$	$*$
$*$	d	$*$
$*$	d	d
$*$	$*$	$*$

(a) buffer

A	B
d	$*$
$*$	d
e	$*$
$*$	e

(b) buffer

A	B
e	$*$
d	$*$
$*$	d
$*$	d

(c) var

A	B
$*$	d
$*$	$*$
d	$*$

(d) prophet

Table 3: Example observations of buffer, variable and prophet.

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 3a: first the input port fires with d, while the output port must block. The data element is stored in memory: as long as the output does not fire, memory is retained. As long as the buffer is full, the next input remains blocked. Then the output fires with d, the element stored in memory, and the memory is cleared in the next observation. Looking just at the input and output ports; we have Table 3b. Also see Table 1 for an earlier example.

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

We illustrate the variable in Table 3c: first the input fires with e; the output port must block. Then the input fires d, which overwrites the previous memory, and the output must still block. When there is no input, the output may fire; it fires d but it does not erase the memory. Hence, the next round when there is no input, the output port may fire again with d.

If a variable is full, any input element overwrites the existing value in memory. Additionally, if a variable is full, it remains full. A variable may fire both input and output ports: its input is never instantaneously transported to its output.

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

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

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

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

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

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

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

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

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

We analyze two another properties of components. A component is linear if for every input port, every input element maps uniquely to the same output

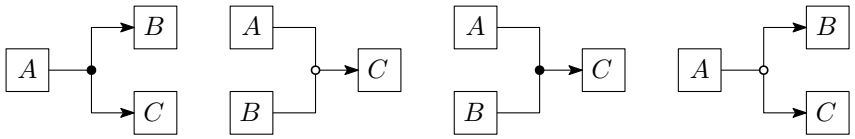


Figure 18: Nodes

element at some output port at some time. It is causal if for every output port and every output element, there exists an equal input element at some input port at some time in the past.

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

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

A.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 18).

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

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

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

Component:	replicator(α)
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(α)
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(α)
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(α)
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 is an overview of the primitive components in this section.

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

Table 4: A non-exhaustive table of primitive components. The first columns display #number of visible ports, (input ports/output ports). The third column shows that all components have indepencence, the forth column shows progress. The fifth column displays the synchronization property: synchronous (✓) or asynchronous (✗) or both. The sixth column displays the transportation property: instantaneous (✓) or delayed (✗) or both. The last columns display the property of linearity (✓) and the property of causality (✓).