A Logical Formalism for Coordination Protocols

Hans-Dieter Hiep and Benjamin Lion

Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
hdh@cwi.nl, b.lion@cwi.nl

Abstract. A logical foundation for concurrent and distributed systems consists of a model-theoretic (semantics) and proof-theoretic approach (syntax). Our approach seeks a model that suits 'computation', 'concurrency', and 'coordination'. We introduce a logic that captures the essence of such a model: an execution is a stream of observations of places; a computation is a set of potential executions of which only a single one actually happens [18], capturing inherent non-determinism of concurrency; and coordination [2] is modeled as a binary composition operation on computations. We explore high-level properties that classify computations.

1 Introduction

Current hardware and software design practices favor modular and reusable components. These are combined to form large-scale inter-networked complex distributed systems. The trustworthiness of these systems in turn relies on the trustworthiness of their comprised components: an error in any layer potentially has devastating societal impact by causing safety, privacy and security problems.

A promising direction for increasing such trust is the certification of essential components: e.g. operating systems drivers [11], and compilers [28]. However, as far as we know, there is not a unifying framework that allows the certification of software components: concurrent and distributed algorithms implemented in commonly-used programming languages. Certification typically requires the formulation of properties of low-level operational details (such as shared memory access or message passing).

An alternative approach is to raise the level of abstraction. A system comprises multiple independent components that coordinate storage and transmission of information. Higher-level properties of components should be verified regardless of individual implementations. We explore a number of higher-level properties important for concurrent and distributed systems: we describe stuttering and asynchronous components, and distinguish progress and stabilization properties. These properties are expressed in a logical formalism for components.

The main result of this paper is the establishment of such a logical formalism, to describe computations expressed as certain first-order formulas, called coordination protocols. The standard interpretation of a coordination protocol is a computation, and the only computations definable in our logic have a

corresponding coordination protocol. We allow reasoning about the validity of properties to classify definable computations.

This work is closely related to process algebra [19, 6]. The reasons why process algebra needs proof methodology [20], may also apply to our work: realistic systems exhibit models that are too large to deal with state-based techniques. We take as starting point research in compositional verification of properties. Additionally, there are many impossibility results in distributed and concurrent systems (see e.g. [18]), but there is not a single general theory in which we can derive such impossibility results. However, in the present paper we refrain from any proof-theoretical developments, leaving that for the future.

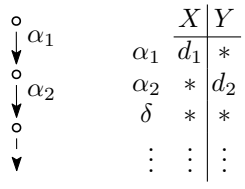
Our work originated from studying Reo, a coordination language for components [5, 4]. The behavior of components, intuitively, corresponds to computations. Reo is closely related to process algebra, but with an important and profound difference: process algebras are action-based, Reo is interaction-based. Over the years Reo developed over thirty formal semantics: these semantics can be roughly divided into three groups, co-algebraic models (e.g. streams), operational models (e.g. automata), and others [25]. We see our work as an other semantics for Reo.

Reo allows modeling based on components. Alongside Reo, numerous languages are proposed for component-based modeling, e.g. separating component from connector [1], and a calculus based on asynchronous π-calculus [30]. As Gössler mentions, the properties of component-based systems have not been studied systematically [23]. In this paper, we set out to define such properties, thereby contributing to this question.

1.1 Executions

An *execution* is modeled as a calculation performed step by step. We model a step by an *observation* that records observed values at some time for a given set of places. We represent an execution as an infinite sequence of observations, where each step is sequentially ordered with regard to the time of appearance.

For each place, either a data element or $*$ is recorded. If an observation contains only $*$, we call it a *silent* observation (denoted by δ). Conversely, if an observation contains at least one value that is not $*$, we call it an *actual* observation (denoted by $\alpha, \alpha_1, \alpha_2, \ldots$). We let $\gamma, \gamma_1, \gamma_2, \ldots$ denote arbitrary observations, being either silent or actual.



	X	Y
α_1	d_1	$*$
α_2	$*$	d_2
δ	$*$	$*$
$\vdots$	$\vdots$	$\vdots$

Fig. 2: Execution as an infinite sequence and as an infinite table

For example, we consider that α_1 is observed first, then α_2, followed by a δ, and so on, we can imagine an execution starting as depicted in Fig. 2 on the left. Take $\{X, Y\}$ to be the set of places. The places correspond to the columns of the table in Fig. 2 on the right. We have observed two data elements, d_1 and d_2, at different times at different places. One could represent an execution as a stream of observations (left), or as a mapping from places to streams of data values (right).

The *domain* of an observation is the set of places it records. Two observations are compatible whenever they have the same domain. Within an execution, every two observations are compatible. We speak of the domain of an execution, to mean the domain of any of its observations. Two executions are compatible if they have the same domain.

An execution as an infinite table of observations captures continuous behavior over time restricted to a set of places. Additionally, we take the following intuitive assumptions for granted:

Assumption 1. (Continuity) *The table of observations must contain all data elements ever observable at the given places, viz. no data elements are missing.*

Assumption 2. (Locality) *Observations are restricted to only its domain, and such observation never precludes at any time observation of data at other places.*

It was found that the idea of representing an execution as an infinite table, has an origin in Kleene's 1951 paper on representation of indefinite events [26].

1.2 Computations

A computation is modeled as a set of potential executions of which only a single execution actually happens. In principle, it is impossible to know what execution actually happens in the face of non-determinism. Therefore, we study the set of potential executions as a whole.

$$\left\{ \begin{array}{c|c|c} & X & Y \\ \hline \alpha_1 : & d_1 & d_1 \\ \alpha_4 : & d_3 & * \\ \alpha_3 : & d_4 & * \\ \vdots & \vdots & \end{array} , \begin{array}{c|c|c} & X & Y \\ \hline \alpha_2 : & d_2 & d_1 \\ \alpha_6 : & d_2 & * \\ \delta : & * & * \\ & \vdots & \vdots \end{array} , \ldots \right\}$$

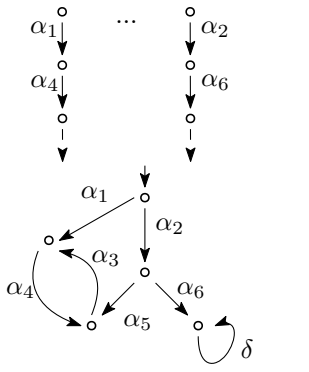


Fig. 4: Computation as a set, as a labeled transition system

We take a computation to be a set of compatible executions. The domain of a computation is the domain of any of its executions, or the domain must be explicitly given if the computation is empty.

There are two different representations of computations. First, we could consider computations as sets of executions, viz. sets of infinite tables or as sets of streams of observations. Second, we could consider a computation as a labeled transition system: a directed graph of configurations and labeled arcs between them, and an initial configuration. Both are depicted in Fig. 4. In the latter representation, arcs are labeled by observations. An execution is an infinite path in such a labeled transition system. The computation is then recovered by taking all infinite paths starting from the initial configuration.

Whereas the representation of computations as sets does not interrelate executions, a labeled transition system displays structural properties

of executions, such as sharing the same prefix or containing cycles. For example, the labeled transition system in Fig. 4 shows that there exist executions that contain infinite alternations of the observations α_4 and α_3.

A *labeled transition system* is a directed graph with configurations as vertices and arcs as edges. A *configuration* is an arbitrary element of a set of configurations. There is a designated *initial configuration* I. An *arc* between configurations is a tuple (C, γ, C'), where C and C' are configurations and γ an arbitrary observation. Arcs are denoted $C \xrightarrow{\gamma} C'$. A configuration C is *inconsistent* if there is no C' such that there is an arc from C to C'.

A *trace* is an infinite sequence of alternating configurations and observations. A *path* is a finite sequence of alternating configurations and observations. Traces (and paths) are denoted $C_1 \xrightarrow{\gamma_1} C_2 \xrightarrow{\gamma_2} C_3 \cdots (C_{n-1} \xrightarrow{\gamma_n} C_n)$.

Two paths *follow* if the last configuration of the first path is the first configuration of the second path. A path starting with the initial configuration is a *prefix*. A *cycle* is a path such that the first and last configurations are the same.

Given a cycle $C_1 \xrightarrow{\gamma_1} \cdots \xrightarrow{\gamma_n} C_1$, we may *unroll* it by walking the cycle multiple times, e.g. $C_1 \xrightarrow{\gamma_1} \cdots \xrightarrow{\gamma_n} C_1 \xrightarrow{\gamma_1} \cdots \xrightarrow{\gamma_n} C_1$.

A trace is *periodic* if it can be formed by a prefix followed by a cycle, such that the cycle is unrolled forever. A prefix then is included in a trace if, superimposing the prefix over the trace, each configuration and observation match up.

In a labeled transition system, each arc represents a step. The number of steps taken from the initial configuration indicates the current round. Each configuration reachable from the initial configuration is at some round. We assume our graphs do not contain any junk, that is: every configuration is reachable from the initial configuration.

1.3 Components

A *component* encapsulates a set of places, of which a subset is exposed as its *interface*. The places of a component's interface are called *ports*. Two components coordinate by composing them into a composite component. A component can be regarded to be in composition with any other component; the latter is called its *environment*. Two components interact with each other through their ports. A component constraints the possible values that can be exchanged at its port.

Intuitively, we refer to a port as the place where a value is exchanged between components. A port is then described by the stream of exchanged values between a component and its environment, viz. the flow of information is modeled by a data stream. Each port has an associated data type, which is the type of the observed values at that port. The type of data that may flow through a port is the data type of the stream. Each data type consists at least of a 'null' value $*$ that represents the absence of data. If at some time a port is *inhibited* or *blocked*, no data element is exchanged through the port: this is indicated by $*$ at that time in the data stream of that port. We say that a port *fires* if there is an actual data element, different than $*$, is exchanged through the port.

The behavior of a component is a computation, viz. a set of infinite tables such that each port corresponds to a column. The interface of a component is

the domain of a computation. To compose two components, we form a new component. The behavior of a composed component, has as domain the union of the domains of the behaviors of its constituent components. For a pair of executions that belong to two given computations, a composition of these executions is included in the newly formed computation. A composition of two executions is zipping the streams of observations: for each pair of observations at some time, a new observation is formed either by the values of places that are not in both domains, or by coinciding values of places that are common in both domains. If two executions disagree at some point in time on the value observed at the same place, then there is no composition of the two executions.

2 Foundation

In this section, we formalize components by so-called *coordination protocols*. Coordination protocols are formulas in a many-sorted logic. The free variables of a coordination protocol fix the ports of the defined component. We compose components by taking the conjunction of their coordination protocols.

Let's consider an example of a component with two ports X and Y, accepting the same type of values noted $d_1, d_2, ...$ We take the perspective of an external observer that records all values flowing at port X and Y. The observed values are collected in an infinite table. We depict each table as one element in the set, as seen in Fig. 4.

An observation can be easily understood as a row in a table, being the values observed simultaneously at port X and Y. An execution is a stream of observations, which corresponds to an infinite table. We consider computations definable by coordination protocols: all executions satisfying this formula comprise the defined computation.

Towards this aim, we define a particular many-sorted logic together with a standard interpretation that interprets certain kinds of sorts as data streams. Furthermore, we allow universal and existential quantification over these kinds of sorts, and introduce quantification indexed over sets of ports. This is useful for formulating properties of components.

Our logical formalism is inspired by Dokter's rule-based semantics [15]. He defines coordination protocols as relations of data streams and a modal logic equipped with stream constraints that are constructed using stream head equality, stream derivatives, and modal operators. In our formalization we employ standard first-order logic to define coordination protocols, freeing ourselves from the limited expressivity of modal logic.

2.1 Data streams

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

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

'null' value. Assume that equality of data types is decidable. We write $A, B, \ldots$ to denote arbitrary data types. As convention, we write $a \in A$ to mean some element a of the carrier set.

Let a stream be a function from natural numbers to some set. We denote streams by the Greek letters $\sigma, \tau, \ldots$. Intuitively, one thinks of streams as an enumeration. *Data streams* are functions from naturals to data types, e.g. $\sigma : \mathbb{N} \to A$ for some data type A. Alternatively, we may define streams by a stream differential equation. See [32, 31] for an elementary introduction. In short:

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, take $N = \{\mathbb{N}, *\}$ as a data type where $* = 0$. The enumeration $(0, 0, 0, \ldots)$ that repeats 0 forever is a stream. $\sigma(x) = 0$ defines this stream as a function. $\sigma(0) = 0$ and $\sigma' = \sigma$ defines this stream as a differential equation.

2.2 Logical formalism

We now consider a many-sorted first-order logic with equality. Our presentation is as usual [16]. We first define the syntax of terms and formulas of some signature. Then we define semantics and fix a standard interpretation. We have the useful notions of assignment, solution, satisfiability, and validity. What is different than standard first-order logics is our treatment of streams and quantification over streams.

Definition 1. *A* signature $\Sigma = (S, F, P)$ *consists of the following data: S is a set of* sorts*, F is a set of* function symbols*, P is a set of* predicate symbols*. An* arity *is a list of sorts $\langle s_1, \ldots, s_n \rangle$ where $s_1, \ldots, s_n \in S$. Each function symbol has an associated non-empty arity, and each predicate symbol has an associated (possibly empty) arity.*

Function symbols of arity length one are called *constant symbols*, and predicate symbols of empty arity are called *proposition symbols*. We only consider signatures for which all of the following conditions hold for each data type A:

- There are two sorts, $A \in S$ and $(N \to A) \in S$. We have the constant symbol $*_A \in F$ with arity $\langle A \rangle$ and for each data element $d \in A$ we have the constant symbol $d_A \in F$ with arity $\langle A \rangle$.
  In particular, we have $N \in S$, and so for any natural number $n \in \mathbb{N}$ we have the symbol $n_N \in F$ with arity $\langle N \rangle$. We also have the binary operations $+ \in F$, $- \in F$, $\times \in F$ with arity $\langle N, N, N \rangle$.
- We have $\mathrm{at}_A \in F$ with arity $\langle (N \to A), N, A \rangle$, and $\mathrm{skip}_A \in F$ with arity $\langle (N \to A), N, (N \to A) \rangle$.
- We have equality $=_A \in P$ with arity $\langle A, A \rangle$. Furthermore, we have the proposition symbols $\bot, \top \in P$, and $\leq \in P$ with arity $\langle N, N \rangle$.

Fix some signature $\Sigma = (S, F, P)$ for which these conditions hold. The definition for terms and formulas are not surprising. Let V be a set of variables.

Definition 2. *Let s be a sort. A* term of sort s *is defined inductively: if $x \in V$, then x^s is a term of sort s; if $c \in F$ is a constant symbol, then c is a 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 3. *A* formula *is defined inductively: if $p \in P$ is a proposition symbol, then p is a 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 a formula; other formulas are formed by the usual binary connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, and formed by quantifiers $\exists x^s, \forall x^s$ for any $x \in V$.*

We conveniently assume that each variable is associated to a single sort only. Given a formula ϕ, by the set of free variables $FV(\phi)$ we mean the variables occurring in ϕ that are not bound by any quantifier. A formula without any free variable is a *sentence*. Next, we define (standard) interpretations.

Definition 4. *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 a function symbol $f \in F$ with arity $\langle s_1, \ldots, s_n, s_{n+1} \rangle$ maps to a function $f^{\mathcal{M}} : s_1^{\mathcal{M}} \times \cdots \times s_n^{\mathcal{M}} \rightarrow s_{n+1}^{\mathcal{M}}$, and a constant symbol $c \in F$ with arity $\langle s \rangle$ maps to $c^{\mathcal{M}} \in s^{\mathcal{M}}$; 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 $p^{\mathcal{M}} \subseteq s_1^{\mathcal{M}} \times \cdots \times s_n^{\mathcal{M}}$, and proposition symbols $p \in P$ with arity $\langle \rangle$ maps to a proposition.*

An interpretations is *standard* if these conditions hold for each data type A:

- The sort A is mapped to the carrier set of data type A, and $N \in S$ is mapped to the set of natural numbers $\mathbb{N}$ in particular. The sort $(N \rightarrow A) \in S$ is mapped to the set of data streams $\mathbb{N} \rightarrow A$.
- Constants $*_A^{\mathcal{M}}$ and $d_A^{\mathcal{M}}$ are the null values $* \in A$ and data elements $d \in A$.
- $+, -, \times$ are interpreted as the usual arithmetical functions on $\mathbb{N}$.
- at_A is interpreted as a function such that $\mathrm{at}_A^{\mathcal{M}}(\sigma)(n) = \sigma(n)$, and skip_A is interpreted as a function such that $\mathrm{skip}_A^{\mathcal{M}}(\sigma)(n) = \sigma^{(n)}$.
- $=_A^{\mathcal{M}}$ is equality of values of data type A, $\bot^{\mathcal{M}}$ never holds and $\top^{\mathcal{M}}$ always holds, and $\leq^{\mathcal{M}}$ is the relation of less than or equals between naturals.

Now, let $\mathcal{M}$ denote a fixed standard interpretation. We define the notion of assignment. Assignments are necessary, since not every domain element has a corresponding term. 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 $\beta \vDash \phi$ 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 $\vDash \phi$ and holds if and only if $\beta \vDash \phi$ holds for all assignments β.

If β_1, β_2 are two assignments and $\beta_1(x^s) = \beta_2(x^s)$ for all free variables x^s in ϕ, then $\beta_1 \vDash \phi$ and $\beta_2 \vDash \phi$. An assignment β that is restricted to map only the free variables of a formula ϕ such that $\beta \vDash \phi$ is called a *solution* of ϕ.

We shall treat sort annotations implicitly to prevent clutter. We also use a more convenient notation for at_A and skip_A: let X be a port of sort $(N \to A)$ and t be a variable of sort N, then the term $\mathrm{at}_A(X, t)$ is written as $X(t)$ and the term $\mathrm{skip}_A(X, t)$ is written as $X^{(t)}$. We call such terms *applications* and *derivations,* respectively. We also use $t > s$ for $\neg(t \leq s)$, and $s < t$ for $t > s$.

2.3 Coordination protocols

In this section, we make good use of our logic to encode the behavior of components. Recall that we intuitively understand executions as a stream of observations that can be tabulated, and a computation is a set of such potential executions. We consider coordination protocols as a special class of formulas, that allows us to define computations.

Definition 5. *A* coordination protocol *is a formula ϕ, where all free variables x^s of ϕ must be of sort $s = (N \to A)$ for some data type A. These free variables $FV(\phi)$ are also called the* ports *of ϕ.*

Definition 6. *The coordination protocol ϕ induces the set $\mathcal{L}(\phi) = \{\beta \mid \beta \vDash \phi\}$ consisting of all solutions of ϕ.*

Each coordination protocol ϕ induces a computation $\mathcal{L}(\phi)$. Every solution $\beta \in \mathcal{L}(\phi)$ corresponds to an execution: the columns of the infinite table are the ports $FV(\phi)$, and for each column $X \in FV(\phi)$ there is a data stream $\beta(X)$. This gives us the table representation of an execution, where each row corresponds to an observation. Each round of a computation precisely corresponds to all the observations at a particular index of such a table, often called time variable t.

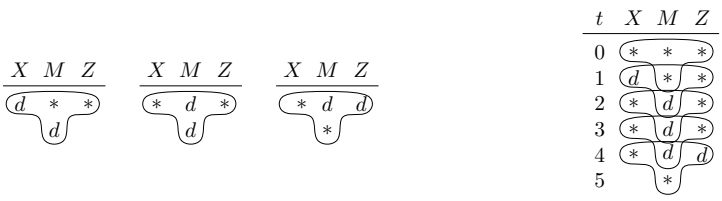


Fig. 5: On the left, frame conditions on observations with domain X, M, Z, where all ports have the same sort of data stream of data type A, and $d \in A$ is some constant. On the right, an example prefix of an execution.

We illustrate our intuition using *frame conditions*. If we want a frame condition to persists over time, then it must constrain every row. Consider Fig. 5, where on the left we have three different frame conditions that relate the observation of ports X, M, Z to an observation of port M one step ahead:

$$\exists d.(X(t) = d \wedge M(t) = * \wedge Z(t) = * \wedge M(t+1) = d) \qquad (\psi_1)$$
$$\exists d.(X(t) = * \wedge M(t) = d \wedge Z(t) = * \wedge M(t+1) = d) \qquad (\psi_2)$$
$$\exists d.(X(t) = * \wedge M(t) = d \wedge Z(t) = d \wedge M(t+1) = *) \qquad (\psi_3)$$

To formalize that the frame condition slides over each observation, we universally quantify over the time variable t:

$$\forall t.(\psi_1 \vee \psi_2 \vee \psi_3) \qquad (\psi)$$

These sliding frame conditions defines a set of executions: ψ is a coordination protocol with $FV(\psi) = \{X, M, Z\}$. See an example in Fig. 5 on the right.

3 Properties

Arbab claims in [3] that "the most challenging aspect of concurrency involves the study of interaction and its properties." We understand properties as classes of computations. We gloss over some properties and discuss their relevance.

(Time) independence is also known as stuttering [9]. Independence is a closure condition on a computation of insertion and removal of silent observations under certain conditions. These conditions prevent the possibility to insert an infinite number of silent transitions after arbitrary prefixes, which allows us to distinguish between acceptable behaviors which terminate from acceptable behaviors that do not, this makes divergence explicit. A similar notion is branching bisimulation with explicit divergences [17].

Synchronicity and *asynchronicity* are properties that relate components by their firing patterns. The distinction synchronous and asynchronous is useful to make in the context of asynchronous processor architectures [22]. A synchronous circuit requires a clock signal to propagate through the circuit, wasting energy and limiting the clock speed to the slowest component that receives the same clock signal. Even if a circuit consists of different synchronous regions, communicating asynchronously among such regions is highly efficient [33].

Termination is useful to demonstrate that a machine always halts in a finite number of steps, after having performed a particular task. *Perpetual* computation is useful to guarantee productivity. Moreover, we take deadlock as a property that lies in between of termination and perpetual computations. It is useful to detect that a computation contains deadlock [14, 23].

Congruence, *joinability* and especially *confluence* are properties typically studied in the setting of abstract reduction systems [27]. These properties are useful when studying self-stabilizing algorithms [10, 21].

3.1 Schematic sentences

Properties are schematically defined sentences in our logic. Typically, properties quantify over executions of some computation defined by a coordination protocol. To do so, we introduce port-indexed quantification and port-bound conjunction and disjunction. Such a quantification can be thought of as a special abbreviation that expands into a formula.

Given a finite set $P \subseteq V$ of ports such that $P = \{X_1, X_2, \ldots, X_n\}$, we introduce the following abbreviations:

$$\forall \vec{Z} : P.\phi \quad \coloneqq \quad \forall Y_1.\forall Y_2.\ldots\forall Y_n.\phi$$

Exists is defined similarly. We expect the variables $Y_1, \ldots, Y_n$ to be variables, freshly introduced by a quantification, such that the sorts match up with the sorts of $X_1, \ldots, X_n$ in P. We regard $\vec{Z}$ as a map, that takes $X_i \in P$ to a corresponding variable Y_i. We lift derivation $\vec{Z}^{(t)}$ to mean that port variables $X_i \in P$ are mapped to terms $Y_i^{(t)}$.

To use such port-indexed quantification, we employ big conjunction (and disjunction) that binds an index $i \in P$, together with an application of such index to an occurrence of $\vec{Z}$ that is bound by a surrounding $\forall \vec{Z} : P$ or $\exists \vec{Z} : P$. These are described as follows:

$$\bigwedge_{i \in P} \ldots(\vec{Z}_i)\ldots \quad \coloneqq \quad \ldots(Y_1)\ldots \wedge \ldots(Y_2)\ldots \wedge \cdots \wedge \ldots(Y_n)\ldots$$

Disjunction is defined similarly. We think of i as a port that is indexed over, and $\vec{Z}_i$ to apply such port to obtain the fresh variable that is bound by the quantifier of $\vec{Z}$. We could also consider the identity map of P to itself, denoted $\vec{P}$, and use it to quantify over a set of known ports.

Given a coordination protocol ϕ with free variables $P = FV(\phi)$. Let $\vec{Z} : P$ be a map from P to variables. By $\phi(\vec{Z})$ we mean the coordination protocol ϕ in which every of its free variables is substituted by the corresponding variable mapped by $\vec{Z}$.

For example, consider the following schematically defined property, where ϕ is a coordination protocol and $P = FV(\phi)$:

$$\mathbf{prod}(\phi) \coloneqq \forall \vec{X} : P.\ (\phi(\vec{X}) \to \exists t.(\bigvee_{i \in P} \vec{X}_i \neq *))$$

This property is read as follows: "$\forall \vec{X} : P$." is read as "Given an arbitrary infinite table $\vec{X}$ with columns P", "$(\phi(\vec{X}) \to \ldots$" is read as "such that $\vec{X}$ is an execution of the computation described by ϕ", "$\ldots \to \exists t.(\bigvee_{i \in P} \vec{X}_i(t) \neq *)$" is read as "then there exists some time (t), such that there is some port $(i \in P)$ that fires at that time $(\vec{X}_i(t) \neq *)$."

Let $P(\phi)$ be a property defined schematically over some ϕ. Once ϕ is known, we can substitute it in the schematically defined property and verify its validity. Thus, such properties describe classes of definable computations; all computations described by a coordination protocol ϕ such that $P(\phi)$ is valid, i.e. $P(\phi)$ denotes $\{\phi \mid \vDash P(\phi)\}$.

3.2 Independence and (a)synchronicity

Given an arbitrary infinite table $\vec{X}$ with columns P and some time variable t:

$$\textbf{all-silent}(\vec{X} : P, t) \coloneqq \bigwedge_{i \in P} \vec{X}_i(t) = * \qquad \textbf{some-firing}(\vec{X} : P, t) \coloneqq \bigvee_{i \in P} \vec{X}_i(t) \neq *$$
$$\textbf{some-silent}(\vec{X} : P, t) \coloneqq \bigvee_{i \in P} \vec{X}_i(t) = * \qquad \textbf{all-firing}(\vec{X} : P, t) \coloneqq \bigwedge_{i \in P} \vec{X}_i(t) \neq *$$

Intuitively, we can stretch an execution by the insertion of silent observations. An execution consisting of silent observations can be shrunk. Let ϕ be an arbitrary coordination protocol with $P = FV(\phi)$.

$$\begin{aligned}\textbf{stretch}(\phi) \coloneqq \forall \vec{X} : P.(\phi(\vec{X}) \rightarrow \forall t. \exists \vec{Y} : P.(\phi(\vec{Y}) \wedge \\ \textbf{all-silent}(\vec{Y}, t) \wedge \textbf{precong}(\vec{X}, \vec{Y}, t) \wedge \textbf{cong}(\vec{X}^{(t)}, \vec{Y}^{(t+1)})))\end{aligned}$$

$$\begin{aligned}\textbf{shrink}(\phi) \coloneqq \forall \vec{X} : P.(\phi(\vec{X}) \rightarrow \forall t. \exists \vec{Y} : P.(\phi(\vec{Y}) \wedge \\ (\textbf{all-silent}(\vec{X}, t) \ \rightarrow \ \textbf{precong}(\vec{X}, \vec{Y}, t) \wedge \textbf{cong}(\vec{X}^{(t+1)}, \vec{Y}^{(t)}))))\end{aligned}$$

A coordination protocol ϕ is time independent if $\textbf{stretch}(\phi) \wedge \textbf{shrink}(\phi)$ is valid. We define prefix congruence **precong** and congruence **cong** later.

A synchronous component either fires all its ports or none of them fire. A non-synchronous component violates this constraint, e.g. allows two out of three ports firing at the same time. An asynchronous component has that at most one port fires exclusively and other ports do not fire.

$$\textbf{sync}(\vec{X} : P) \coloneqq \forall t. (\textbf{all-silent}(\vec{X} : P, t) \vee \textbf{all-firing}(\vec{X} : P, t))$$
$$\textbf{async}(\vec{X} : P) \coloneqq \forall t. \bigwedge_{i \in P} (\vec{X}_i(t) \neq * \ \rightarrow \bigwedge_{j \in P \setminus \{i\}} \vec{X}_j(t) = *))$$

A property on an execution can be lifted to hold for every execution in a computation. A computation definable by coordination protocol ϕ is synchronous or asynchronous if one of these property holds, where $P = FV(\phi)$.

$$\textbf{sync}(\phi) \coloneqq \forall \vec{X} : P.(\phi(\vec{X}) \rightarrow \textbf{sync}(\vec{X} : P))$$
$$\textbf{async}(\phi) \coloneqq \forall \vec{X} : P.(\phi(\vec{X}) \rightarrow \textbf{async}(\vec{X} : P))$$

Non-synchronous $\neg\textbf{sync}(\phi)$ and asynchronous are different properties.

3.3 Progress and stabilization

Each computation can be divided into three classes: a computation is perpetual, or is terminating, or is neither. By a perpetual computation, we mean that every execution is always productive. By a terminating computation, we mean that every execution terminates. Productivity means that there is an eventual

actual observation, and non-productive means there are always silent observations. For a computation that is neither perpetual nor terminating, we classify computations containing a deadlock.

Productivity of an execution $\vec{X} : P$ is formalized as:

$$\mathbf{prod}(\vec{X} : P) \coloneqq \exists t. \bigvee_{i \in P} X_i(t) \neq *$$

Non-productivity is obtained by negation of productivity property. An execution terminates if there is a point where it becomes non-productive:

$$\mathbf{term}(\vec{X} : P) \coloneqq \exists t.(\neg \mathbf{prod}(\vec{X}^{(t)}))$$

where we understand t to be the maximum number of steps until completion. An execution $\vec{X}$ is always productive if $\neg\mathbf{term}(\vec{X})$.

A computation definable by coordination protocol ϕ is perpetual or terminating, where $P = FV(\phi)$, by considering all executions:

$$\mathbf{perp}(\phi) \coloneqq \forall \vec{X} : P.(\phi(\vec{X}) \rightarrow \neg\mathbf{term}(\vec{X}))$$
$$\mathbf{term}(\phi) \coloneqq \forall \vec{X} : P.(\phi(\vec{X}) \rightarrow \mathbf{term}(\vec{X}))$$

Now consider the gray area, $\neg(\mathbf{perp}(\phi) \vee \mathbf{term}(\phi))$, where there exists an execution that is productive, and there also exists an execution that terminates.

A deadlock situation occurs if an execution eventually reaches a point where it remains non-productive.

$$\mathbf{deadlock}(\phi, \vec{X} : P) \coloneqq \phi(\vec{X}) \wedge \mathbf{term}(\vec{X}) \wedge \neg(\mathbf{term}(\phi) \vee \mathbf{perp}(\phi))$$

A computation ϕ with $P = FV(\phi)$ contains a deadlock $\vec{X}$ if $\vec{X}$ terminates, and ϕ is non-terminating. Clearly, $\neg\mathbf{perp}(\phi)$ is implied by a computation containing a deadlock.

$$\begin{aligned}
\mathbf{cong}(\vec{X} : P, \vec{Y} : P) &\coloneqq \forall t.(\bigwedge_{i \in P} \vec{X}_i(t) = \vec{Y}_i(t)) \\
\mathbf{conv}(\vec{X} : P, \vec{Y} : P) &\coloneqq \exists s.(s > 0 \wedge \mathbf{cong}(\vec{X}^{(s)}, \vec{Y})) \\
\mathbf{joinable}(\vec{X} : P, \vec{Y} : P) &\coloneqq \exists t. \exists s.(\mathbf{cong}(\vec{X}^{(s)}, \vec{Y}^{(t)})) \\
\mathbf{wconfl}(\vec{X} : P, \vec{Y} : P) &\coloneqq \mathbf{joinable}(\vec{X}^{(1)}, \vec{Y}) \wedge \mathbf{joinable}(\vec{X}, \vec{Y}^{(1)}) \\
\mathbf{wconfl}(\vec{X} : P, \vec{Y} : P) &\coloneqq \exists t.(\mathbf{conv}(\vec{X}, \vec{Y}^{(t)})) \wedge \exists t.(\mathbf{conv}(\vec{Y}, \vec{X}^{(t)}) \\
\mathbf{confl}(\vec{X} : P, \vec{Y} : P) &\coloneqq \forall t. \forall s.(\mathbf{joinable}(\vec{X}^{(t)}, \vec{Y}^{(s)}))
\end{aligned}$$

Congruence of two executions means that their observations are forever the same. Convergence of an execution $\vec{X}$ to an execution $\vec{Y}$ means that at some point in the future of $\vec{X}$, we obtain congruence with $\vec{Y}$ from the beginning. Two executions are joinable whenever there exists two points in the executions from

which they are congruent. We have two formulations of weakly confluent, that two executions after one step are joinable. Given two executions, if any prefix of the two can be joined, the executions are confluent.

A computation ϕ is cyclic if every execution eventually repeats itself from the start, and is acyclic if every execution never repeats itself from the start. A computation is periodic if any execution can be joined with itself.

$$\begin{aligned}\mathbf{cycl}(\phi) &\coloneqq \forall \vec{X} : U.(\phi(\vec{X}) \to \mathbf{conv}(\vec{X}, \vec{X}))\\ \mathbf{acycl}(\phi) &\coloneqq \forall \vec{X} : U.(\phi(\vec{X}) \to \neg\mathbf{conv}(\vec{X}, \vec{X}))\\ \mathbf{periodic}(\phi) &\coloneqq \forall \vec{X} : U.(\phi(\vec{X}) \to \mathbf{join}(\vec{X}^{(1)}, \vec{X}))\end{aligned}$$

Additional properties on computations could be derived. We state two more properties that are not yet fully understood. Livelock is symmetric with deadlock: a computation contains a livelock $\vec{X}$ if $\vec{X}$ is always productive in a non-perpertual computation. Intuitively, this definition does not capture the idea of getting trapped from progressing towards a goal.

$$\mathbf{livelock}(\phi, \vec{X} : P) \coloneqq \phi(\vec{X}) \wedge \neg\mathbf{term}(\vec{X}) \wedge \neg(\mathbf{term}(\phi) \vee \mathbf{perp}(\phi))$$

Fairness ensures that every executions eventually behaves like every other execution, that in turn eventually behave like every other execution.

$$\mathbf{fair}(\phi) = \forall \vec{X} : U.(\phi(\vec{X}) \to \forall \vec{Y} : U.(\phi(\vec{Y}) \wedge \mathbf{conv}(\vec{X}, \vec{Y})))$$

4 Conclusion

In this paper, we have seen the establishment of a logical formalism for defining specifications of the behavior of components, and properties of such components. We have studied some important of properties such as synchronicity, perpetuity, termination, deadlock-freedom, and confluence.

The logical formalism presented in this paper intuitively captures what we mean by executions and computations. Components specify a computation by constraining the permissible executions. A coordination protocol denotes a computation; its solutions correspond to executions. Properties of coordination protocols can be determined by considering their formalization, in which a placeholder is substituted for the coordination protocol for which the property should hold. This allows one to reason about the validity of such properties.

4.1 Related work

In [29], Li and Sun formalize Reo in the Coq theorem prover. They formalize circuits using data packets, which are either pending or delivered. They employ a similar technique in modeling nodes as data streams, but take data either as natural number or empty. Implicitly, they prove delay insensitivity of an alternator component, by showing that appending empty data packets does not violate the specified behavior.

In our paper, we have left out any discussion or reference to constraint automata, to avoid any confusion that might arise between coordination games and constraint automata. Constraint automata also give a semantics to Reo [8]. Jongmans has extended

this semantics to constraint automata with memory [24]. These constraint automata with memory still separate data constraints from synchronization constraints. Data and synchronization are combined, by introducing $*$, and can captured by a single constraint [15]. Automata can subsequently be simplified to a single state, by capturing its state as a memory value. The next insight is to model memory variables by streams. A first-order logic in which constraints on streams can be directly expressed was found, and presented in this paper.

VeReofy is a model checking tool for analyzing Reo circuits [7]. It accepts two input languages: one for construction of compositions, and one for defining constraint automata. The model checker allows verification of temporal properties expressed in linear temporal logic (LTL) and computational tree logic (CTL). The formalism in this paper seems to subsume both logics in their expressivity.

The idea of formalizing the behavior of a component in a first-order logic is not new [13]. Broy gives a formalism that is similar to the one presented in this paper [12], that allows for the equational specification of components by using predicates. The behavior of components are represented by stream processing functions, that map a tuple of streams correspond to input ports to a tuple of streams corresponding to output ports. A component is defined by the set of acceptable such behaviors; similar to our interpretation of coordination protocols. Broy defines an algebra for composition: $C \parallel C$ for parallel composition, μ_x^y for so-called feedback of channels, and ρ_y^x for renaming channel names. He defines the predicate interpretation of a composition in a very similar way as what we have done: a parallel composition takes the conjunction of the specifications of two components, and specifies that the shared channels must be identical.

Acknowledgements

The authors wish to thank Farhad Arbab, Jasmin Blanchette, Kasper Dokter, Sun-Shik Jongmans, Femke van Raamsdonk, and Wan Fokkink.

References

1. Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
2. Farhad Arbab. What do you mean, coordination? *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, 19, 1998.
3. Farhad Arbab. Elements of Interaction. In *Complex Systems Design & Management*, pages 1–28. Springer, 2010.
4. Farhad Arbab. Puff, the magic protocol. In *Formal Modeling: Actors, Open Systems, Biological Systems*, pages 169–206. Springer, 2011.
5. Farhad Arbab. Proper protocol. In *Theory and Practice of Formal Methods*, pages 65–87. Springer, 2016.
6. Jos Baeten. *Applications of process algebra*, volume 17. Cambridge, 2004.
7. Christel Baier, Tobias Blechmann, et al. Formal verification for components and connectors. In *International Symposium on Formal Methods for Components and Objects*, pages 82–101. Springer, 2008.
8. Christel Baier et al. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.
9. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT, 2008.

10. Rena Bakhshi, Wan Fokkink, et al. Leader election in anonymous rings: Franklin goes probabilistic. In *Fifth Ifip International Conference On Theoretical Computer Science–Tcs 2008*, pages 57–72. Springer, 2008.
11. Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *International Conference on Integrated Formal Methods*, pages 1–20. Springer, 2004.
12. Manfred Broy. Algebraic specification of reactive systems. *Theoretical Computer Science*, 239(1):3 – 40, 2000.
13. Manfred Broy. Time, abstraction, causality and modularity in interactive systems. *Electronic Notes in Theoretical Computer Science*, 108:3–9, 2004.
14. K. M. Chandy, Jayadev Misra, et al. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, 1983.
15. Kasper Dokter and Farhad Arbab. Rule-based form for stream constraints. In *International Conference on Coordination Languages and Models*, pages 142–161. Springer, 2018.
16. Herbert Enderton. *A Mathematical Introduction to Logic*. Elsevier, 2001.
17. David De Frutos Escrig, Jeroen Keiren, and Tim Willemse. Games for bisimulations and abstraction. *Logical Methods in Computer Science*, 13(4:15), 2017.
18. Wan Fokkink. *Distributed Algorithms: An Intuitive Approach*. MIT, 2013.
19. Wan Fokkink. *Introduction to process algebra*. Springer, 2013.
20. Wan Fokkink, Jan Friso Groote, et al. Process algebra needs proof methodology. *EATCS Bulletin 82*, pages 109–205, 2004.
21. Martin Gairing et al. Distance-two information in self-stabilizing algorithms. *Parallel Processing Letters*, 14(03n04):387–398, 2004.
22. David Geer. Is it time for clockless chips? (Asynchronous processor chips). *Computer*, 38(3):18–21, 2005.
23. Gregor Gössler, Sussane Graf, et al. An approach to modelling and verification of component based systems. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 295–308. Springer, 2007.
24. Sung-Shik Jongmans. *Automata-theoretic protocol programming*. PhD thesis, Centrum Wiskunde & Informatica (CWI), Leiden University, 2016.
25. Sung-Shik Jongmans and Farhad Arbab. Overview of Thirty Semantic Formalisms for Reo. *Scientific Annals of Computer Science*, 22(1), 2012.
26. Stephen Cole Kleene. Representation of events in nerve nets and finite automata. Technical report, Project RAND, Santa Monica, 1951.
27. Jan Willem Klop. *Term rewriting systems*. Centrum voor Wiskunde en Informatica, 1990.
28. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM SIGPLAN Notices*, volume 41, pages 42–54. ACM, 2006.
29. Yi Li and Meng Sun. Modeling and verification of component connectors in Coq. *Science of Computer Programming*, 113:285–301, 2015.
30. Oscar Nierstrasz and Franz Achermann. A calculus for modeling software components. In *Formal Methods for Components and Objects*, pages 339–360. Springer, 2003.
31. J.J.M.M. Rutten. Elements of stream calculus: (an extensive exercise in coinduction). *Electronic Notes in Theoretical Computer Science*, 45:358 – 423, 2001.
32. J.J.M.M. Rutten. *On Streams and Coinduction*. Unpublished manuscript, 2002.
33. Kenneth S. Stevens. Energy and performance models for clocked and asynchronous communication. In *Ninth International Symposium on Asynchronous Circuits and Systems*, pages 56–66. IEEE, 2003.