An Algebra for Interaction of Cyber-Physical Components

Proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus prof. dr.ir. *H. Bijl*,
volgens besluit van het College voor Promoties
te verdedigen op *weekday day month year*
klokke *time* uur

door *Benjamin Lion*

geboren te *France*

in *1994*

Promotiecomissie

Promotors:	Prof. dr. F. Arbab	(Leiden University, Leiden, The Netherlands)
Co-promotor:	Dr. C. Talcott	(SRI International, CA, USA)
		(secretaris)
		(voorzitter)

The author's work on this thesis was partially supported by the U. S. Office of Naval Research under award number N00014-15-1-2202.

Backcover: *cover attribution*

Contents

Chapter 1

Introduction

1.1 Context

Today's technological and theoretical progress bring new challenges in the field of theoretical computer science. The emergence of digital systems that reliably measure and actuate physics reveals new kinds of interconnected systems that we call *cyber-physical systems*. A cyber-physical system typically refers to a system in which digital processes (e.g., controllers) interact (e.g., via sensing or actuating) with and through physical medium (e.g., space, time). The understanding of cyber-physical systems is branches into several lines of research [52]. We should mention first the field of cybernetics, whose appearance in the literature dates back to Wiener [81]. Cybernetics, and more largely control theory, aim at studying the feedback mechanisms taking place between a governor (kuberneties in Greek) and a physical system. Given a suitable model of how the physics operates, the governor can *steer* the physical system towards a desired objective and control its behavior [72, 25]. Also, and on top of cybernetics, the interaction among cyber-physical systems is a concern in the field of emergent behavior and swarm robotics [37]. The objective is to find suitable coordination patterns that enable a set of interacting cyber-physical systems to reach a collective objective. Still, challenges remain in the design and analysis for complex cyber-physical systems, and the concurrency that produces in the interaction among their cyber and physical parts. The specific challenge that we undertake in this thesis is that of a design framework that is *compositional* and *concurrent*. We describe in more details the essence and implications of two properties that we deem essential in formal models for cyber-physical systems.

Compositionality is, intuitively, the property that allows forming a *complex* system by assembling *simpler* systems together. Compositionality is used as a principle [77, 40], for instance, in order to assign meanings of natural language sentences given the meaning of its part. This principle has permeated many aspects of computer science, such as in program semantics, e.g., assigning meanings to programs, and in system design, e.g., defining a framework to construct systems. Given a set of small blocks, and a rule to assemble such blocks, one can quickly get a system whose behavior surpasses in complexity each of its parts. The emergence of new kinds of machines (e.g., interconnected networks, cyber-physical systems) requires a new stand on the question of compositional specification for such systems (see Chapter 2).

Concurrency is the field in theoretical computer science that studies the behavior of a set of communicating processes. The aim is to understand the behavior emerging from a set of interacting machines as yet another machine, similarly to how the behavior of a machine would be understood from its parts. One of the challenges, for instance, comes from the fact that, in a network, events may occur in parallel, independently, or *at the same time*. Therefore, the mode of interaction among machines is not necessarily derivable from the behavior specification of each machine. Several models of how machines interact through communication have been studied, from the original neural network model [48], or the network of communicating machines [65]. While still many paths explore ways to design and analyze concurrent systems, we observe yet another type of concurrent systems of interest: in cyber-physical systems, the physical medium in which machines evolve plays an active role in the interaction among the cyber parts.

We give an illustrative example to justify that compositionality and concurrency are two important features to include in a model for cyber-physical systems. For instance, consider a group of robots, each running a program that takes decision based on the sequence of sensor readings. The sensors that equip a robot return the current position of the robot and the position of any adjacent obstacle. The interaction occurring between robots in the group cannot be derived solely from the specification of individual robots. If the field on which the robots roam changes its property, the same group of robots might sense different values, and therefore take different actions. Also, the time at which a robot acts and senses will affect the decision of each controller and will change the resulting collective behavior.

Other instances of applications emerge from the challenges to architect a cyber-physical system, as in the trends of Industry 4.0 [75], digital twins [19], or the Internet of Things [10]. In Industry 4.0, the manufacturing process is equipped with sensors,

that observe physical variables of interest along the supply chain, and actuators that perform physical tasks. Such architecture aims at improving the automation of repetitive physical tasks, the allocation of resources, but also the detection and signaling of malfunctioning devices. The digital twin trend aims at creating models of physical phenomenon that gather information, update, and monitor in live physical objects to achieve some objectives. In such cases, having a formal design framework to model interaction between the physics and its discrete model is of key importance to minimize faults and increase accuracy. The Internet of Things captures the idea of connecting sensors and actuators over the internet. Each device therefore becomes a node that has cyber-physical capabilities that can be addressed remotely. As a result, the protocol that rules the interaction over the nodes in such a network is central to prove properties of, for instance, security or resilience.

Generally, a design framework gives some means to specify *what* behavior is desired and hides internal details that explain *how* a (network of) machines would construct such behavior. As pictured in Figure 1.1 the operational part of a system, which is represented by machines M_1 and M_2 interacting under a protocol Σ, is separated from the description of its behavior, which is given by the transformation $[\![\cdot]\!]$ as components C_1 and C_2. Ideally, operations on machine behaviors, such as the algebraic operations $\times_\Sigma$, should reflect practical interactions between machines. If such is the case, the behavior of the system consisting of machines M_1 and M_2 under the protocol Σ, is formally captured by the product of components C_1 and C_2 under the operation $\times_\Sigma$.

One benefit of having such component algebra is that it allows for reasoning about updates. Indeed, what qualifies as an update is any component for which its substitution with another component would not change the resulting behavior. As a consequence, the class of all M_2 that preserves the same collective behavior C_3, under protocol Σ, contains possible machine replacements.

We highlight some fundamental differences in the interaction occurring between purely cyber components (e.g., discrete programs interacting with other discrete programs), and cyber-physical components (e.g., discrete programs interacting with physics). We leave the definition of a component and an interaction signature abstract, and refer to Chapter 2 for a more precise description.

Cyber-cyber interaction The models that capture interactions between two cyber components follows, in general, several assumptions:

- *Reproducibility.* The time value at which two machines M_1 and M_2 initially start does not change the resulting behavior. This assumption also means that the

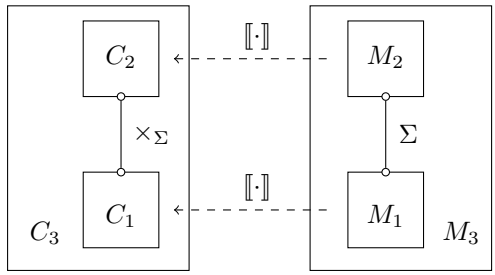


Figure 1.1: Two machines M_1 and M_2, whose respective behavior is captured by C_1 and C_2, interact through a protocol Σ. The product of the two components under the operation $\times_\Sigma$ reflects the behavior of the machines interacting with the protocol Σ.

same protocol between the two machines is independent of the initial time, and therefore reproducible at a later time.

- *Closed system.* Given a fixed set of machines and a fixed protocol among the machines, one can, in theory, reason about the whole system statically. Static analysis may therefore detect beforehand, for instance, some race conditions.

- *No event missed.* It is possible to coordinate each machine so that no interacting event is missed (i.e., no message exchanged is missed). For instance, the hardware connection between each machine assumes a frequency of communication that eliminates the possibility to miss an event on either end.

In Chapter 3, we explore the property of cyber-cyber interactions by studying the class of components for which only the ordering of events matters, and not the precise value of the observation's time stamp.

Cyber-physical interaction The assumptions underlying models for cyber-physical interactions are different in nature than assumptions for cyber-cyber interactions:

- *Variance under time shift.* Some physical systems have time dependent and chaotic responses while interacting with cyber systems. In such cases, the time at which the system is initialized changes the resulting behavior of the composite cyber-physical system. For instance, the program that controls the decision of an autonomous car will receive different responses from the physics if executed at different times (change of traffic, change of weather, change of landscape, etc.).

- *Open system.* Some physical systems are so complex that their analytical characterization is infeasible. As a consequence, some reasoning can happen only

dynamically as reaction, and the resulting specification must adapt to unexpected alternatives.

- *Approximation and missed events.* The sensing of physical quantities is inherently lossy as such quantities are continuously changing (i.e., function of time). An observation captures samples of those quantities at a time instant. As a result, observable properties on sensor readings are not sufficient to infer that the underlying physical quantity satisfies as well such property. Mechanisms for detection of deviations and diagnosis are therefore necessary to catch errors at runtime due to approximative measurements.

- *Non uniform description.* Dynamics in physics is usually described using differential equations, and leads to characterizing the evolution of physical quantities over real numbers. Alternatively, digital systems make use of discrete measures, and clocked processors that time their sensing and actuations. Thus, modelling concurrency between physical processes and cyber machines gives rise to the problem of uniformly describing their interactions.

1.2 Structure

This thesis lays a foundation to tackle the challenges stated above. We record in the following list the main points of each chapter.

In Chapter 2, we present an algebra of components that can model the four points presented above, making it suitable for modelling interaction in cyber-physical systems. More precisely, components are primitives in this algebra, and capture time-sensitive behavior of a part of a system, which includes both cyber and physical aspects. A component, in isolation, denotes all possible sequences of observations over time that a machine or physical process can exhibit. In composition with other components, only some of such sequences will remain possible. The same component, within different *contexts*, results in different behaviors as in an open system.

The relation between event occurrences in the behavior of two components is captured by algebraic operators. Each operation of the algebra is parametrized by an interaction signature, that specifies how two components interact. Such interaction may for instance allow or disallow events to occur independently (or simultaneously) between two components. The same two components under different interactions would expose different resulting behavior. The algebra therefore allows in some cases for decomposition, using a suitable division operation.

In Section 2.3, we define an operation of linearization that transforms a transactional component, i.e., observing multiple events at the same time, to a linear component, i.e., observing a single event at a time. We give conditions that a valid linearization must satisfy and present two instances of valid linearization: one lock-step procedure that linearizes all observations of a transactional component, and one multi-round procedure that allows for some interleaving. In both cases, the linearization can be performed in parts, i.e., linearization distributes over the product on transactional components. The material in this chapter is based on two journal publications and a paper to be submitted:

- Benjamin Lion, Farhad Arbab, Carolyn L. Talcott: *A semantic model for interacting cyber-physical systems.* J. Log. Algebraic Methods Program. 129: 100807 (2022)

- Benjamin Lion, Farhad Arbab, Carolyn L. Talcott: *A formal framework for distributed cyber-physical systems.* J. Log. Algebraic Methods Program. 128: 100795 (2022)

- Kasper Dokter, Benjamin Lion, Hans-Dieter A. Hiep: *Compositional Linearizations of Transactional Behaviors.* To be submitted (2022)

In Chapter 3, we study a class of components for which the precise time-stamp value of the observation does not matter, under a product that models synchronous interaction. We express, in Section 3.1, the semantics of the Reo coordination language as an algebra of order sensitive components. As an example, we express some well known connectors as a product of port components, under a suitable interaction signature. In Section 3.3, we study temporal properties of Reo connectors. We give a procedure to generate a specification in the Promela language from a logical specification of Reo connectors. We verify temporal properties of Reo connector by using its Promela translation and the Spin model checker. Through this work, we identify some key constructs to simplify the specification of a temporal property using Linear Temporal Logic (LTL) given the port and memory primitives in Reo. The material of the third chapter is based on a workshop paper and an implementation, respectively:

- Benjamin Lion, Samir Chouali, Farhad Arbab: *Compiling Protocols to Promela and Verifying their LTL Properties.* MoDELS (Workshops) 2018: 31-39

- Benjamin Lion, *Treo to Promela compiler*, 10.5281/zenodo.7393621 (2018)

In Chapter 4, we provide an operational and executable specification of components. We first introduce in Section 4.1 an intermediate state-based representation of a component behavior as a labeled transition system called a TES transition system. We define a wide class of products of two such transition systems, each parametrized by a composability relation on their observations. We show that the semantics of TES transition systems as components is compositional with respect to their parametrized product. Based on the intermediate TES transition system, we give in Section 4.2 an executable finite specification of components as agents specified in rewriting logic. The behavior resulting from a concurrent run of a set of agents depends on the composability relation that governs the interaction among the agent. We show that, for some composability relations, the behavior of the concurrent execution of agents coincides with the behavior of the product of the components representing those agents.

In the context of open systems, the behavior of an agent may be large in order to adapt to different contexts. Since not all of such behavior is equally desired, we give a graphical specification language using Reo to specify preference aware agents 4.3. The three sections are based on three publications:

- Benjamin Lion, Farhad Arbab, Carolyn L. Talcott: *Runtime Composition Of Systems of Interacting Cyber-Physical Components.* In proceeding WADT (2022)
- Benjamin Lion, Farhad Arbab, Carolyn L. Talcott: *A Rewriting Framework for Cyber-Physical Systems.* In proceeding Isola (2022)
- Tobias Kappé, Benjamin Lion, Farhad Arbab, Carolyn L. Talcott: *Soft component automata: Composition, compilation, logic, and verification.* Sci. Comput. Program. 183 (2019)

In Chapter 5, we give a concrete implementation of agents described in Chapter 4 in order to analyse collective behaviors resulting from their interaction. We implement in Section 5.1 the executable rewriting framework in Maude, and give a series of detailed applications to demonstrate the usefulness of the modeling framework and the scope of analysis that are possible. In Section 5.2, we implement a concurrent version of Reo and show that the framework is suitable for concurrent execution of Reo primitives. We analyze in Section 5.3 the protocol governing the interactions among a controller, a valve, and some reservoirs, by showing the safety of a controller's strategy. We verify in Section 5.4 some liveness, safety, and sorting properties for energy aware robots roaming on the same field. The material of this section is based on the following implementation:

- Benjamin Lion, *Cyber-physical agent framework in Maude*, Zenodo, 10.5281/zenodo.6592275 (2022)

1.3 Running example

We introduce, through an example, some intuitive concepts that we will formalize later. We consider a cyber-physical system as a set of interacting processes. Whether a process consists of a physical phenomenon (sun rising, electro-chemical reaction, etc.) or a cyber phenomenon (computation of a function, message exchanges, etc.), it exhibits an externally observable behavior resulting from some internal non-visible actions. Instead of a unified way to describe internals of cyber and physical processes, we propose in Section 2.1 a uniform description of what we can externally observe of their behavior and interactions.

An *event* may describe something like *the sun-rise* or *the temperature reading of* $5°C$. An event occurs at a point in time, yielding an event occurrence (e.g., the sun-rise event occurred at 6:28 am today), and the same event can occur repeatedly at different times (the sun-rise event occurs every day). Typically, multiple events may occur at "the same time" as measured within a measurement tolerance (e.g., the bird vacated the space at the same time as the bullet arrived there; the red car arrived at the middle of the intersection at the same time as the blue car did). We call a set of events that occur together at the same time an *observable*. A pair (O, t) of a set of observable events O together with its time-stamp t represents an *observation*. An observation (O, t) in fact consists of a set of event occurrences: occurrences of events in O at the same time t. We call an infinite sequence of observations a *Timed-Event Stream* (TES). A *behavior* is a set of TESs. A *component* is a behavior with an interface.

Consider two robot components, each interacting with its own local battery component, and sharing a field resource. The fact that the robots share the field through which they roam, forces them to somehow coordinate their (move) actions. Coordination is a set of constraints imposed on the otherwise possible observable behavior of components. In the case of our robots, if nothing else, at least physics prevents the two robots from occupying the same field space at the same time. More sophisticated coordination may be imposed (by the robots themselves or by some other external entity) to restrict the behavior of the robots and circumvent some undesirable outcomes, including hard constraints imposed by the physics of the field. The behaviors of components consist of timed-event streams, where events may include some measures

Table 1.1: Each column displays a segment of a timed-event stream for a robot, a battery, and a field component, where observables are singleton events. For $t \in \mathbb{R}_+$, we use $R(t), B(t)$, and $F(t)$ to respectively denote the observable at time t for the TES in the Robot, the Battery, and the Field column. An explicit empty set is not mandatory if no event is observed.

	Robot (R)	Battery (B)	Field (F)
1s	$\{(read(loc, R); (0; 0))\}$		$\{(loc(I); (0; 0))\}$
2s	$\{(move(R); (N, 20\text{W}))\}$	$\{(discharge(B); 20\text{W})\}$	$\{(move(I); (N, 40\text{N}))\}$
3s	$\{(read(loc, R); (0; 1))\}$		$\{(loc(I); (0; 1))\}$
4s	$\{(read(bat, R); 2000\text{Wh})\}$	$\{(read(B); 2000\text{Wh})\}$	
...	...	...	...

	Robot-Battery-Field
1s	$R(1) \cup F(1)$
2s	$R(2) \cup B(2) \cup F(2)$
3s	$R(3) \cup F(3)$
4s	$R(4) \cup B(4)$
...	...

of physical quantities. We give in the sequel a detailed description of three components, a robot (R), a battery (B), and a field (F), and of their interactions. We use the International System of Units to quantify physical values, with time in seconds (s), charging status in Watt hour (Wh), distance in meters (m), force in newtons (N), speed in meters per second ($\mathrm{m\,s^{-1}}$).

A *robot* component, with identifier R, has two kinds of events: a read event $(read(bat, R); b)$ that measures the level b of its battery or $(read(loc, R); l)$ that obtains its position l, and a move event $(move(R); (d, \alpha))$ when the robot moves in the direction d with energy α (in W). The TES in the Robot column in Table 1.1 shows a scenario where robot R reads its location and gets the value $(0; 0)$ at time 1s, then moves north with 20W at time 2s, reads its location and gets $(0; 1)$ at time 3s, and reads its battery value and gets 2000Wh at time 4s, ...

A *battery* component, with identifier B, has three kinds of events: a charge event $(charge(B); \eta_c)$, a discharge event $(discharge(B); \eta_d)$, and a read event $(read(B); s)$, where η_d and η_c are respectively the discharge and charge rates of the battery, and s is the current charge status. The TES in the Battery column in Table 1.1 shows a scenario where the battery discharged at a rate of 20W at time 2s, and reported its charge-level of 2000Wh at time 4s, ...

A *field* component, with identifier F, has two kinds of events: a position event $(loc(I); p)$ that obtains the position p of an object I, and a move event $(move(I); (d, F))$ of the object I in the direction d with traction force F (in N). The TES in the Field column in Table 1.1 shows a scenario where the field has the object I at location $(0; 0)$ at time 1s, then the object I moves in the north direction with a traction force of 40N

at time 2s, subsequently to which the object I is at location $(0; 1)$ at time 3s, ...

When components interact with each other, in a shared environment, behaviors in their composition must also compose with a behavior of the environment. For instance, a battery component may constrain how many amperes it delivers, and therefore restrict the speed of the robot that interacts with it. We specify interaction explicitly as an exogenous binary operation that constrains the composable behaviors of its operand components.

The *robot-battery* interaction imposes that a move event in the behavior of a robot coincides with a discharge event in the behavior of the robot's battery, such that the discharge rate of the battery is proportional to the energy needed by the robot. The physicality of the battery prevents the robot from moving if the energy level of the battery is not sufficient (i.e., such an anomalous TES would not exist in the battery's behavior, and therefore cannot compose with a robot's behavior). Moreover, a read event in the behavior of a robot component should also coincide with a read event in the behavior of its corresponding battery component, such that the two events contain the same charge value.

The *robot-field* interaction imposes that a move event in the behavior of a robot coincides with a move event of an object on the field, such that the traction force on the field is proportional to the energy that the robot puts in the move. A read event in the behavior of a robot coincides with a position event of the corresponding robot object on the field, such that the two events contain the same position value. Additional interaction constraints may be imposed by the physics of the field. For instance, the constraint "no two robots can be observed at the same location" would rule out every behavior where the two robots are observed at the same location.

A TES for the composite Robot-Battery-Field system collects, in sequence, all observations from a TES in a Robot, a Battery, and a Field component behavior, such that at any moment the interaction constraints are satisfied. The column Robot-Battery-Field in Table 1.1 displays the first elements of such a TES.

Chapter 2

A semantic model for interacting cyber-physical systems

Cyber-physical systems often describe systems in which a program (cyber) has to regulate and control a physical quantity. For instance, consider a heating system equipped with sensors and a controller. The controller has as objective to maintain the temperature of a room within some bounds. The controller (cyber) frequently reads the temperature sensors located in the room, and takes decision as to turn the heater on or off. The decision made by the controller changes the dynamic of the temperature profile, and eventually the next readings of the sensors. A similar structure is observed if one wants to direct autonomously a car on a field. The car has some position and energy sensors, and actuates its wheels depending on the value that its controller reads. The decision made by the car changes the location of the car on the field and modifies the remaining energy in its battery.

The two examples above are commonly considered to belong to the field of *control theory*, i.e., the design of control algorithms that monitor observable measures to maintain some invariants. The design of a control algorithm is directly subject to the physical system with which it interacts. As the physical system becomes more complex, finding a suitable model becomes challenging as well. For instance, if the room has several heaters and sensors distributed over the space, the decision taken at one heater may interfere with the decision taken at the other heater. Coordination is

then necessary between controllers. Similarly, in the case where two robots move on the same shared field, the decision of one robot may directly depend on the decision of the other robot to move.

Our model, introduced in Section 2.1, builds on top of existing *control theory* to define cyber-physical systems as a composition of its parts. Similar approaches are taken in software design [3, 51, 33], where a complex monolithic software is broken into subparts that interact. Doing so requires a model to specify explicitly the interaction that occurs in between each parts. We use *components* to refer to the parts of a cyber-physical system, and defines several *product* operations over such components to express their interactions. Intuitively, a component encapsulates both a cyber or a physical process.

Composition is an important feature of a specification language, as it enables the design of a complex system in terms of a product of its parts. Decomposition is equally important in order to reason about structural properties. Usually, however, a system can be decomposed in more than one way, each optimizing for different criteria. In Section 2.2, we extend our model to reason about decomposition. Components compose using a family of algebraic products, and decompose, under some conditions, given a corresponding family of division operators. We use division to specify invariants of a system of components, and to model desirable updates. We apply our framework to design a cyber-physical system consisting of robots moving on a shared field, and identify desirable updates using our division operator

As a theoretical application of our model, we study in Section 2.3 the operation of linearization, that transforms a transactional component, i.e., observing multiple events at the same time, to a linear component, i.e., observing a single event at a time. We consider the class of components for which occurrences of events are independent of the precise value of the time at which they happen, but depend on the past or future occurrences of other events. We give conditions for a linearization to be valid, which intuitively preserves the integrity of the behaviors after linearization, and give two instances of valid linearizations.

2.1 Algebra of Components

The definition of components in this section is similar to the one defined in [3, 51]. Intuitively, a component denotes a set of (infinite) sequences of observations. Whether it is a cyber process or a physical process, our notion of component captures all of its possible sequences of observations.

A model of interaction emerges naturally from our component model by relating observation of events from one component to observation of events from another component. Moreover, we give a construction to lift constraints on observations to constraints on infinite sequences of observations, and ultimately define, from those interaction constraints, algebraic operations on components.

2.1.1 Notations

An *event* is a simplex (the most primitive form of an) observable element. An event may or may not have internal structure. For instance, the successive ticks of a clock are occurrences of a tick event that has no internal structure; successive readings of a thermometer, on the other hand, constitute occurrences of a temperature-reading event, each of which has the internal structure of a name-value pair. Similarly, we can consider successive transmissions by a mobile sensor as occurrences of a structured event, each instance of which includes geolocation coordinates, barometric pressure, temperature, humidity, etc. Regardless of whether or not events have internal structures, in the sequel, we regard events as uninterpreted simplex observable elements.

Notation 1 (Events)**.** *We use* $\mathbb{E}$ *to denote the universal set of events.*

An *observable* is a set of event occurrences that happen together and an *observation* is a pair (O, t) of an observable O and a time-stamp $t \in \mathbb{R}_+$.[1] An observation (O, t) represents an act of atomically observing occurrences of events in O at time t. Atomicity, in its general form, consists of two properties: all events in the set must occur together, and no *interfering* event can occur in between any two events from the set. The second clause is in general formalized by a dependence relation (see Section 2.3). In the case of an observation, atomically observing occurrences of events in O at time t means there exists a small $\epsilon \in \mathbb{R}_+$ such that during the time interval $[t - \epsilon, t + \epsilon]$:

1. every event $e \in O$ is observed exactly once[2], and
2. no event $e \notin O$ is observed.

In the absence of a specified dependence relation, we make the safe, conservative assumption that all events depend on each other. Therefore, the atomicity of an

[1] Any totally ordered dense set would be suitable as the domain for time (e.g., positive rationals $\mathbb{Q}_+$). For simplicity, we use $\mathbb{R}_+$, the set of real numbers $r \geq 0$ for this purpose.

[2] A finer time granularity, i.e., a smaller ϵ, may reveal some ordering relation on the set of events that occur in the same set of observation.

observation, defined above, assumes the dependence relation to be total, i.e., all events are dependent, and no event can interleave within an observation.

We write $\langle s_0, s_1, ..., s_{n-1} \rangle$ to denote a *finite sequence of size n* of elements over an arbitrary set S, where $s_i \in S$ for $0 \leq i \leq n-1$. The set of all finite sequences of elements in S is denoted as S^*. A *stream* over a domain S is a function $\sigma : \mathbb{N} \to S$.[3] We use $\sigma(i)$ to represent the $i+1^{st}$ element of σ, and given a finite sequence $s = \langle s_0, ..., s_{n-1} \rangle$, we write $s \cdot \sigma$ to denote the stream $\tau \in \mathbb{N} \to S$ such that $\tau(i) = s_i$ for $0 \leq i \leq n-1$ and $\tau(i) = \sigma(i-n)$ for $n \leq i$. We use $\sigma^{(n)}$ to denote the *n-th* derivative of σ, such that $\sigma^{(n)}(i) = \sigma(i+n)$ for all $i \in \mathbb{N}$. We use σ' as an abbreviation for the first derivative of the stream σ, i.e., $\sigma' = \sigma^{(1)}$. We use $\mathcal{P}(X)$ to denote the power set of X.

A *Timed-Event Stream (TES)* over a set of events E and a set of time-stamps $\mathbb{R}_+$ is a stream $\sigma \in \mathbb{N} \to (\mathcal{P}(E) \times \mathbb{R}_+)$ where, for every $i \in \mathbb{N}$, let $\sigma(i) = (O_i, t_i)$ and:

1. $t_i < t_{i+1}$, [i.e., time monotonically increases] and
2. for every $n \in \mathbb{N}$, there exists $k \in \mathbb{N}$ such that $t_k > n$ [i.e., time is non-Zeno progressive].

Notation 2 (Time stream)**.** *We use $OS(\mathbb{R}_+)$ to refer to the set of all monotonically increasing and non-Zeno infinite sequences of elements in $\mathbb{R}_+$.*

Notation 3 (Timed-Event Stream)**.** *We use $TES(E)$ to denote the set of all TESs whose observables are subsets of the event set E with elements in $\mathbb{R}_+$ as their time-stamps.*

Given a sequence $\sigma \in TES(E)$ with $\sigma(i) = (O_i, t_i)$ for $i \in \mathbb{N}$, we use the projections $\mathsf{pr}_1(\sigma) \in \mathbb{N} \to \mathcal{P}(E)$ and $\mathsf{pr}_2(\sigma) \in OS(\mathbb{R}_+)$ to denote respectively the sequence of observables where $\mathsf{pr}_1(\sigma)(i) = O_i$ and the sequence of time stamps where $\mathsf{pr}_2(\sigma)(i) = t_i$.

Notation 4 (Observable time)**.** *For $\sigma \in TES(E)$ and $t \in \mathbb{R}_+$, we use $\sigma(t)$ to denote the observable O in σ if there exists $i \in \mathbb{N}$ with $\sigma(i) = (O, t)$, and $\emptyset$ otherwise. We write $\Theta(\sigma)$ for the set of all $t \in \mathbb{R}_+$ such that there exists $i \in \mathbb{N}$ with $\sigma(i) = (O_i, t)$ with $O_i \subseteq E$.*

Note that, for $t \in \mathbb{R}_+$ where $\sigma(t) = \emptyset$, the meaning of $\sigma(t)$ is ambiguous as it may mean either $t \notin \Theta(\sigma)$, or there exists an $i \in N$ such that $\sigma(i) = (\emptyset, t)$. The ambiguity is resolved by checking if $t \in \Theta(\sigma)$.

[3] The set $\mathbb{N}$ denotes the set of natural numbers $n \geq 0$.

Notation 5 (Pair derivative)**.** *For a pair (σ, τ) of TESs, we use $(\sigma, \tau)'$ to denote the new pair of TESs for which the observation(s) with the smallest time stamp has been dropped, i.e., $(\sigma, \tau)' = (\sigma^{(x)}, \tau^{(y)})$ with x (resp. y) is 1 if $\mathsf{pr}_2(\sigma)(0) \leq \mathsf{pr}_2(\tau)(0)$ (resp. $\mathsf{pr}_2(\tau)(0) \leq \mathsf{pr}_2(\sigma)(0)$) and 0 otherwise.*

2.1.2 Components

The design of complex systems becomes simpler if such systems can be decomposed into smaller sub-systems that interact with each other. In order to simplify the design of cyber-physical systems, we abstract from the internal details of both cyber and physical processes, to expose a uniform semantic model. As a first class entity, a component encapsulates a behavior (set of TESs) and an interface (set of events).

Like existing semantic models, such as time-data streams [3], time signal [79], or discrete clock [33], we use a dense model of time. However, we allow for arbitrary but finite interleavings of observations. In addition, our structure of an observation imposes atomicity of event occurrences within an observation. These distinctions mean that for every $\sigma(i) = (O, t), i \geq 0$ of a $\sigma \in TES(E)$: (1) O is finite; and (2) there exists a real number $\epsilon > 0$ such that in the open interval $(t - \epsilon, t + \epsilon)$ no event $e \notin O$ occurs, and every event $e \in O$ occurs exactly once. Such a constraint abstracts from the precise timing of the occurrence of each event in the set O, and turns an observation into an all-or-nothing transaction.

Definition 1 (Component)**.** *A component is a tuple $C = (E, L)$ where $E \subseteq \mathbb{E}$ is a set of events, and $L \subseteq TES(E)$ is a set of TESs. We call E the* interface *and L the externally observable* behavior *of C.*

In contrast with other component models where observables range over the same universal set of events, therefore making component overly specified, our model encapsulates the set of observable events of a component in its interface. Thus, a component *cannot observe* an event that is not in its interface. Moreover, Definition 1 makes no distinction between cyber and physical components. We use the following examples to describe some cyber and physical aspects of components.

Example 1. *Consider a set of two events $E = \{0, 1\}$, and restrict our observations to $\{1\}$ and $\{0\}$. A component whose behavior contains TESs with alternating observations of $\{1\}$ and $\{0\}$ is defined by the tuple (E, L) where*

$$L = \{\sigma \in TES(E) \mid \forall i \in \mathbb{N}. \quad (\mathsf{pr}_1(\sigma)(i) = \{0\} \wedge \mathsf{pr}_1(\sigma)(i+1) = \{1\}) \vee (\mathsf{pr}_1(\sigma)(i) = \{1\} \wedge \mathsf{pr}_1(\sigma)(i+1) = \{0\})\}$$

Note that this component is oblivious to time, and any stream of monotonically increasing non-Zeno real numbers would serve as a valid stream of time stamps for any such sequence of observations. ■

Example 2. *Consider a component encapsulating a continuous function* $f : (D_0 \times \mathbb{R}_+) \to D$*, where* D_0 *is a set of initial values, and* D *is the codomain of values for* f*. Such a function can describe the evolution of a physical system over time, where* $f(d_0, t) = d$ *means that at time* t *the state of the system is described by the value* $d \in D$ *if initialized with* d_0*. We define the set of all events for this component as the range of function* f *given an initial parameter* $d_0 \in D_0$*. The component is then defined as the pair* (D, L_f) *such that:*

$$L_f = \{\sigma \in TES(D) \mid \exists d_0 \in D_0.\ \forall i \in \mathbb{N}.\ \mathsf{pr}_1(\sigma)(i) = \{f(d_0, \mathsf{pr}_2(\sigma)(i))\}\}$$

Observe that the behavior of this component contains all possible discrete samplings of the function f *at monotonically increasing and non-Zeno sequences of time stamp. Different instances of* f *would account for various cyber and physical aspects of components.* ■

Components are declarative entities that may denote either the behavior of a specification, or the behavior of an implementation. The usual relation between the behavior of a program and the property of such program constitutes a refinement relation.

Definition 2 (Refinement)**.** *A component* B *is a refinement of component* A*, written as* $B \sqsubseteq A$*, if and only if* $E_B \subseteq E_A$ *and* $L_B \subseteq L_A$*.*

Lemma 1. *The relation* $\sqsubseteq$ *is a partial order on components.*

Proof. Follows from reflexivity, antisymmetry, and transitivity of set inclusion. □

An alternative to refinement is *containment*. The containment relation makes use of a point-wise inclusion relation on observations of two TESs. The containment relation on components requires that every TES in the behavior of one is point-wise contained in a TES from the behavior of the other.

Definition 3 (Containment)**.** *A TES* σ *is contained in a TES* τ*, written as* $\sigma \leq \tau$*, if and only if, for all* $i \in \mathbb{N}$*,* $\mathsf{pr}_1(\sigma)(i) \subseteq \mathsf{pr}_1(\tau)(i)$ *and* $\mathsf{pr}_2(\sigma) = \mathsf{pr}_2(\tau)$*.*

We extend the containment relation to components: a component $A = (E_A, L_A)$ is contained in a component $B = (E_B, L_B)$, written $A \leq B$, if and only if $E_A \subseteq E_B$, and for every $\sigma \in L_A$, there exists a $\tau \in L_B$ such that $\sigma \leq \tau$.

Lemma 2. *The relation $\leq$ is a pre-order over arbitrary set of components. Let $\mathcal{C}$ be a set of components such that, for all components $A \in \mathcal{C}$ and for any two TESs $\sigma : A$ and $\tau : A$, $(\sigma \leq \tau \wedge \tau \leq \sigma) \implies \sigma = \tau$, then $\leq$ is a partial order on $\mathcal{C}$.*

Proof. Let $A = (E_A, L_A)$, $B = (E_B, L_B)$, and $C = (E_C, L_C)$ be three components. We show that $\leq$ is reflexive, transitive, and antisymmetric for any set $\mathcal{C}$ that satisfies the above condition:

1. reflexivity: $A \leq A$ holds.
2. transitivity. Let $A \leq B$ and $B \leq C$. Then, for all $\sigma : A$, there exists $\tau : B$ such that $\sigma \leq \tau$, and for all $\tau : B$, there exists $\delta : C$ such that $\tau \leq \delta$. Then, we conclude that for all $\sigma : A$, there exists $\delta : C$ such that $\sigma \leq \delta$ and $A \leq C$.
3. antisymmetric. We suppose that A and B are elements of the set $\mathcal{C}$. If $A \leq B$ and $B \leq A$, then for all $\sigma : B$, there exists $\tau : A$ such that $\sigma \leq \tau$. As well, for any $\tau : A$, there exists $\sigma : B$ such that $\tau \leq \sigma$. Thus, for any $\sigma : B$, there exists $\tau : A$ and $\delta : B$ with $\sigma \leq \tau \leq \delta$. Given the assumption of A and B, we can conclude that $\sigma = \tau = \delta$. Similarly, we show that $L_A \subseteq L_B$, and that $A = B$.

□

Remark 1. *The restriction in Lemma 2 to consider components with no internal self containments between distinct TESs is necessary for having $\leq$ as a partial order. Consider for instance the component A with only two TESs in its behavior, $\sigma : A$ and $\tau : A$ where $\mathsf{pr}_1(\sigma) = (\{a, b\})^\omega$ and $\mathsf{pr}_1(\tau) = (\{a\})^\omega$ and $\mathsf{pr}_2(\sigma) = \mathsf{pr}_2(\tau)$. Let B be a component with a singleton behavior $\delta : B$ such that $\mathsf{pr}_1(\delta) = (\{a, b\})^\omega$ and $\mathsf{pr}_2(\delta) = \mathsf{pr}_2(\sigma)$. Then, $A \leq B$, and $B \leq A$, but $A \neq B$.*

2.1.3 Composition

A complex system typically consists of multiple components that interact with each other. The running example in Section 1.3 shows three components, a *robot*, a *bat*, and a *field*, where, for instance, a move observable of a robot must coincide with an accommodating move observable of the field and a discharge observable of its battery. The design challenge is to faithfully represent the interactions among involved components, while keeping the description modular, i.e., specify the robot, the battery, and the field as separate, independent, but interacting components. For that purpose, we capture in an interaction signature the type of the interaction between a pair of components, and we define a family of binary products acting on components, each

parametrized with an interaction signature. As a result, the product of two components, under a given interaction signature, returns a new component whose behavior reflects that the two operand components joint behavior is constrained according to the interaction signature. Such construction opens possibilities for modular reasoning both about the interaction among components and about their resulting composite behavior.

An interaction signature consists of two elements: a composability relation and a composition function. The composability relation specifies which pairs[4] of TESs are allowed to compose, and the composition function constructs a new TES out of a pair TESs. The condition for two TESs to be composable may depend on an external context. For instance, the observation of event a at time t in a TES may conflict with the observation of event b at that same time t in another TES in a context where the latter could have observed a as well. To capture this notion, we generalized the notion of a composability relation to take as parameter a pair of carrier sets of events that acts as a context of alternative events for the pair of TESs. Then, when we write $(\sigma, \tau) \in R(E_1, E_2)$, we mean that σ and τ are composable under the composability relation R given their respective context E_1 and E_2.

Definition 4 (Composability relation on TESs)**.** *A composability relation is a parametrized relation R such that for all $E_1, E_2 \subseteq \mathbb{E}$, we have $R(E_1, E_2) \subseteq TES(E_1) \times TES(E_2)$.*

Definition 5 (Symmetry)**.** *A parametrized relation Q is* symmetric *if, for all (x_1, x_2) and for all (X_1, X_2): $(x_1, x_2) \in Q(X_1, X_2) \iff (x_2, x_1) \in Q(X_2, X_1)$.*

A composability relation on TESs serves as a necessary constraint for two TESs to compose. We define *composition* of TESs as the act of forming a new TES out of two TESs.

Definition 6. *A composition function $\oplus$ on TES is a function $\oplus : TES(\mathbb{E}) \times TES(\mathbb{E}) \to TES(\mathbb{E})$.*

In order to simplify the development of the theory of components, we group a pair of a composability relation and a composition function into an *interaction signature*.

Definition 7. *An interaction signature $\Sigma = (R, \oplus)$ is a pair of a composability relation R and a composition function $\oplus$.*

[4]Non-binary relations may also be considered, i.e., constraints imposed on more than two components.

Example 3 (Union of TESs)**.** *The operation $\cup$ forms the interleaved union of observables occurring in a pair of TESs, i.e., for two TESs σ and τ, we define $\sigma \cup \tau$ to be the TES such that $\Theta(\sigma \cup \tau) = \Theta(\sigma) \cup \Theta(\tau)$ and $(\sigma \cup \tau)(t) = \sigma(t) \cup \tau(t)$ for all $t \in \Theta(\sigma) \cup \Theta(\tau)$.* ■

The following examples present some useful interaction signatures for composition of TESs that, e.g., enforce synchronization or mutual exclusion of observables.

Example 4 (Synchronous interaction)**.** *In this example, we define the synchronous interaction signature $\Sigma_{sync} = (R_{sync}, \cup)$. In a cyber-physical system, the action (of a cyber system) and the reaction (of a physical system) co-exist simultaneously in the same observation, and are therefore synchronous.*

Then, $R_{sync}(E_1, E_2)$ relates pairs of TESs such that all shared events occur at the same time in both TESs, i.e., $(\sigma, \tau) \in R_{sync}(E_1, E_2)$ if and only if, for all time stamps $t \in \mathbb{R}_+$, $\sigma(t) \cap E_2 = \tau(t) \cap E_1$. A synchronous interaction signature Σ_{sync} filters pairs of TESs that satisfy the R_{sync} relation and merges composable pairs of observations. ■

Example 5 (Asynchronous interaction)**.** *In this example, we define the asynchronous interaction signature $\Sigma_{async} = (R_{async}, \cup)$. Typically, the asynchronous interaction signature prevents the same event to occur at the same time in a pair of TESs.*

Then, $R_{async}(E_1, E_2)$ relates pair of TESs such that all shared event between E_1 and E_2 occur at different time, i.e., $(\sigma, \tau) \in R_{async}(E_1, E_2)$ if and only if $\sigma(t) \cap \tau(t) = \emptyset$ for all $t \in \Theta(\sigma) \cup \Theta(\tau)$. An asynchronous interaction signature $\Sigma = (R_{async}, \cup)$ filters pairs of TESs that satisfy the R_{async} relation and merges composable pairs. ■

Example 6 (Free interaction)**.** *A free interaction signature, $\Sigma_{free} = (R_{free}, \cup)$, uses R_{free} for the most permissive composability relation on TESs such that, for any $E_1, E_2 \subseteq \mathbb{E}$ and any $\sigma \in TES(E_1)$ and $\tau \in TES(E_2)$, we have $(\sigma, \tau) \in R_{free}(E_1, E_2)$.* ■

We define a binary product operation on components, parametrized by an interaction signature. Intuitively, the newly formed component describes, by its behavior, the evolution of the joint system under the constraint that the interactions in the system satisfy the composability relation. Formally, the product operation returns another component, whose set of events is the union of sets of events of its operands, and its behavior is obtained by composing all pairs of TESs in the behavior of its operands deemed composable by the composability relation.

Definition 8 (Product). *Let $\Sigma = (R, \oplus)$ be an interaction signature, and $C_i = (E_i, L_i)$, $i \in \{1, 2\}$, two components. The product of C_1 and C_2, under Σ, denoted as $C_1 \times_\Sigma C_2$, is the component (E, L) where $E = E_1 \cup E_2$ and L is defined by*

$$L = \{\sigma_1 \oplus \sigma_2 \mid \sigma_1 \in L_1,\ \sigma_2 \in L_2,\ (\sigma_1, \sigma_2) \in R(E_1, E_2)\}$$

The following examples define several products on components given the interaction signatures introduced in Example 4, 5, and 6.

Example 7 (Synchronous product). *The behavior of component $C_1 \times_{(R_{sync}, \oplus)} C_2$ contains TESs obtained from the composition under $\oplus$ of every pair $\sigma_1 \in L_1$ and $\sigma_2 \in L_2$ of TESs that are related by the synchronous composability relation R_{sync} (see Example 4) which excludes all event occurrences that do not synchronize. In the case where $\oplus = \cup$, we write $\bowtie = \times_{(R_{sync}, \cup)}$, and we call the operator $\bowtie$ the join operator.* ■

Example 8 (Asynchronous product). *The behavior of component $C_1 \times_{(R_{async}, \oplus)} C_2$ contains TESs resulting from the composition under $\oplus$ of every pair $\sigma_1 \in L_1$ and $\sigma_2 \in L_2$ of TESs that are related by the mutual exclusion composability relation R_{async} (see Example 5) which may exclude some simultaneous event occurrences. In the case where $\oplus = \cup$, we write $\between = \times_{(R_{async}, \cup)}$.* ■

Example 9 (Free product). *The behavior of component $C_1 \times_{(R_{free}, \oplus)} C_2$ contains every TES obtained from the composition under $\oplus$ of every pair $\sigma_1 \in L_1$ and $\sigma_2 \in L_2$ of TESs. This product does not impose any constraint on event occurrences of its operands (see Example 6).* ■

Example 10. *Consider a suitable interaction signature Σ that captures the interactions between a robot R and its field F, such that the expression $R \times_\Sigma F$ represents the resulting system. For instance, Σ may force every observable move of the robot to synchronize with a displacement of the robot on the field F, and every read observable of the robot with a location displayed by the field. In the case of two interacting robots roaming on the same field, one would like to build the resulting system compositionally as an expression of the form $(R_1 \times_{\Sigma_1} F_1) \times_{\Sigma_3} (R_2 \times_{\Sigma_2} F_2)$, where each of the Σ_i locally captures the interaction between its respective component. Note that, for instance, Σ_3 may enforce that the two fields F_1 and F_2 exclude the joint observations of R_1 and R_2 to be at the same location.* ■

The product of two components indirectly depends on the interface of its operands, since its composability relation does so. Therefore, it is *a priori* not certain that algebraic properties such as commutativity or associativity hold for such user defined

products. Algebraic properties are important when designing a complex system in order to find equivalent and sometimes simpler expressions. Lemma 3 relates properties of a parametrized product with properties of its parameter, i.e., properties of the interaction signature. Intuitively, the first item of Lemma 3 considers interaction signatures that yield symmetric operations. As a result, the order of which components appear in the product parametrized by such signatures is irrelevant. The second item shows conditions on interaction signatures that allow flattening of nested products: the product of A with $B \times_\Sigma C$ becomes equivalent to the product of $A \times_\Sigma B$ with C. When an interaction signature satisfies both algebraic properties, the resulting product acts as an n-ary top level operator on a multiset of components. For instance, the synchronous interaction signature of Example 4 is one such top level n-ary operator.[5]

Lemma 3. *Let $\Sigma = (R, \oplus)$ be an interaction signature. Then:*

- *if R is symmetric, then $\times_\Sigma$ is* commutative *if and only if $\sigma_1 \oplus \sigma_2 = \sigma_2 \oplus \sigma_1$ for all $(\sigma_1, \sigma_2) \in R$;*
- *if R is such that, for all $E_1, E_2, E_3 \subseteq \mathbb{E}$,*

$$(\sigma_1, \sigma_2 \oplus \sigma_3) \in R(E_1, E_2 \cup E_3) \wedge (\sigma_2, \sigma_3) \in R(E_2, E_3) \iff$$
$$(\sigma_1, \sigma_2) \in R(E_1, E_2) \wedge (\sigma_1 \oplus \sigma_2, \sigma_3) \in R(E_1 \cup E_2, E_3)$$

  then $\times_\Sigma$ is associative *if and only if $\sigma_1 \oplus (\sigma_2 \oplus \sigma_3) = (\sigma_1 \oplus \sigma_2) \oplus \sigma_3$ for all $(\sigma_1, \sigma_2 \oplus \sigma_3) \in R(E_1, E_2 \cup E_3)$ with $(\sigma_2, \sigma_3) \in R(E_2, E_3)$;*

- *if for all $E \subseteq \mathbb{E}$ and $\sigma, \tau \in TES(E)$, we have $(\sigma, \tau) \in R(E, E) \implies \sigma = \tau$, then $\times_\Sigma$ is* idempotent *if and only if $\sigma \oplus \sigma = \sigma$ for all $(\sigma, \sigma) \in R$.*

Proof. Commutativity. Let $C_1 = (E_1, L_1)$ and $C_2 = (E_2, L_2)$ be two components, and $\Sigma = (R, \oplus)$ be an interaction signature with R symmetric as in Definition 5. We write $C = (E, L) = C_1 \times_\Sigma C_2$ and $C' = (E', L') = C_2 \times_\Sigma C_1$. We first observe that $E = E_1 \cup E_2 = E'$. The condition for the product of two components to be commutative reduces to showing that $L = L'$, also equivalently written as:

$$\begin{aligned} L = L' \iff & \{\sigma_1 \oplus \sigma_2 \mid \sigma_1 \in L_1,\ \sigma_2 \in L_2,\ (\sigma_1, \sigma_2) \in R(E_1, E_2)\} \\ & = \{\sigma_2 \oplus \sigma_1 \mid \sigma_1 \in L_1,\ \sigma_2 \in L_2,\ (\sigma_2, \sigma_1) \in R(E_2, E_1)\} \end{aligned}$$

If $\sigma_1 \oplus \sigma_2 = \sigma_2 \oplus \sigma_1$ for $(\sigma_1, \sigma_2) \in R(E_1, E_2)$, then $L = L'$ and $\times_\Sigma$ is commutative.

[5] Distributivity holds for some products. We leave the study of the conditions under which distributivity holds as future work.

Oppositely, if $L = L'$, we show that $\oplus$ is commutative. Let C_σ be the component $(E_\sigma, \{\sigma\})$ where $E_\sigma = \bigcup\{\sigma(i) \mid i \in \mathbb{N}\}$. Thus, for any $(\sigma_1, \sigma_2) \in R(E_1, E_2)$, $C_{\sigma_1} \times_\Sigma C_{\sigma_2} = (E_{\sigma_1} \cup E_{\sigma_2}, \{\sigma_1 \oplus \sigma_2\}$. A necessary condition for $\times_\Sigma$ to be commutative is that $\{\sigma_1 \oplus \sigma_2\} = \{\sigma_2 \oplus \sigma_1\}$, which imposes that $\sigma_1 \oplus \sigma_2 = \sigma_2 \oplus \sigma_1$.

Associativity. Let $(R, \oplus)$ be a pair of a composability relation and a composition function on TESs with R such that, for every $(\sigma_1, \sigma_2, \sigma_3) \in L_1 \times L_2 \times L_3$:

$$\begin{aligned}&(\sigma_1, \sigma_2) \in R(E_1, E_2) \wedge (\sigma_1 \oplus \sigma_2, \sigma_3) \in R(E_1 \cup E_2, E_3) \iff\\ &(\sigma_2, \sigma_3) \in R(E_2, E_3) \wedge (\sigma_1, \sigma_2 \oplus \sigma_3) \in R(E_1, E_2 \cup E_3)\end{aligned}$$

We consider three components $C_i = (E_i, L_i)$, with $i \in \{1, 2, 3\}$.

The set of events for component $((C_1 \times_\Sigma C_2) \times_\Sigma C_3)$ is the set $E_1 \cup E_2 \cup E_3$, which is equal to the set of events for component $(C_1 \times_\Sigma (C_2 \times_\Sigma C_3))$.

Let L' and L'' respectively be the behaviors of components $(C_1 \times_\Sigma C_2) \times_\Sigma C_3$ and $C_1 \times_\Sigma (C_2 \times_\Sigma C_3)$. If $\sigma_1 \oplus (\sigma_2 \oplus \sigma_3) = (\sigma_1 \oplus \sigma_2) \oplus \sigma_3$ for all $(\sigma_1, \sigma_2 \oplus \sigma_3) \in R(E_1, E_2 \cup E_3)$ with $(\sigma_2, \sigma_3) \in R(E_2, E_3)$, then $L' = L''$. We show some sufficient conditions for $L' = L''$, also written as

$$\begin{aligned}L' &= \{(\sigma_1 \oplus \sigma_2) \oplus \sigma_3 \mid \sigma_1 \in L_1,\ \sigma_2 \in L_2,\ \sigma_3 \in L_3.\ (\sigma_1, \sigma_2) \in R(E_1, E_2) \wedge\\ &\qquad\qquad (\sigma_1 \oplus \sigma_2, \sigma_3) \in R(E_1 \cup E_2, E_3)\}\\ &= \{(\sigma_1 \oplus \sigma_2) \oplus \sigma_3 \mid \sigma_1 \in L_1,\ \sigma_2 \in L_2,\ \sigma_3 \in L_3.\ (\sigma_2, \sigma_3) \in R(E_2, E_3) \wedge\\ &\qquad\qquad (\sigma_1, \sigma_2 \oplus \sigma_3) \in R(E_1, E_2 \cup E_3)\}\\ &= \{\sigma_1 \oplus (\sigma_2 \oplus \sigma_3) \mid \sigma_1 \in L_1,\ \sigma_2 \in L_2,\ \sigma_3 \in L_3.\ (\sigma_2, \sigma_3) \in R(E_2, E_3) \wedge\\ &\qquad\qquad (\sigma_1, \sigma_2 \oplus \sigma_3) \in R(E_1, E_2 \cup E_3)\}\\ &= L''\end{aligned}$$

using the assumption on R for the first equality, and the assumption on $\oplus$ for the second equality.

Let $(\sigma_1, \sigma_2 \oplus \sigma_3) \in R(E_1, E_2 \cup E_3)$ with $(\sigma_2, \sigma_3) \in R(E_2, E_3)$, then $C_{\sigma_1} \times_\Sigma (C_{\sigma_2} \times_\Sigma C_{\sigma_3}) = (C_{\sigma_1} \times_\Sigma C_{\sigma_2}) \times_\Sigma C_{\sigma_3}$ which then implies that $\sigma_1 \oplus (\sigma_2 \oplus \sigma_3) = (\sigma_1 \oplus \sigma_2) \oplus \sigma_3$.

Idempotency. We show that if for all $E \subseteq \mathbb{E}$, and $\sigma, \tau \in TES(E)$, we have that $(\sigma, \tau) \in R(E, E)$ implies $\sigma = \tau$, then $\times_\Sigma$ is idempotent if and only if $\sigma \oplus \sigma = \sigma$ for $(\sigma, \sigma) \in R(E, E)$. We first observe that, given a component $C = (E, L)$, the component $C \times_\Sigma C = (E, L')$ has the same set of events, E.

We show that $(\sigma_1, \sigma_2) \in R(E, E) \implies \sigma_1 = \sigma_2$ and $\oplus$ idempotent is a sufficient

condition for having $L' = L$. Indeed,

$$\begin{aligned} L' &= \{\sigma_1 \oplus \sigma_2 \mid \sigma_1, \sigma_2 \in L,\ (\sigma_1, \sigma_2) \in R(E, E)\} \\ &= \{\sigma_1 \oplus \sigma_1 \mid \sigma_1 \in L\} \\ &= L \end{aligned}$$

Similar to the previous cases, if for all $E \subseteq \mathbb{E}$, and $\sigma, \tau \in TES(E)$, we have that $(\sigma, \tau) \in R(E, E)$ implies $\sigma = \tau$, then $\times_\Sigma$ is idempotent if and only if $\oplus$ is idempotent. Conversely, if $C_\sigma \times_\Sigma C_\sigma = C_\sigma$ and $(\sigma, \sigma) \in R(E, E)$, then $\sigma \oplus \sigma = \sigma$. □

Monotonicity shows that the inclusion of component's behavior is preserved by product. Let A and B be two components such that $B \sqsubseteq A$. Suppose that P is a component that models a property satisfied by component A and preserved under product with a component C, then P is satisfied by component B and component $B \times C$, by monotonicity. Note that the definition of monotonicity assumes $\times$ to be commutative. That assumption can be relaxed by defining left and right monotonicity.

Definition 9 (Monotonicity)**.** *Let $\times$ be a commutative product. Then, $\times$ is monotonic if and only if, for $B \sqsubseteq A$ and for any C, we have $B \times C \sqsubseteq A \times C$.*

Lemma 4 (Monotonicity of $\bowtie$)**.** *The product $\bowtie$ in Example 4 is monotonic.*

Proof. Let A, B, and C be three components, such that $B \sqsubseteq A$. Then, the interface of $B \bowtie C$ is $E_B \cup E_C$, which is included in $E_A \cup E_C$ the interface of $A \bowtie C$.

For any TES $\sigma : B \bowtie C$, there exist two TESs $\beta : B$ and $\delta : C$ such that (β, δ) are synchronous, and $\sigma = \beta \cup \delta$. Since for any $\beta : B$ we also have $\beta : A$, then σ is also an element of the behavior of $A \bowtie C$, and $B \bowtie C \sqsubseteq A \bowtie C$. □

The algebraic nature of our formalism allows the possibility to introduce other kinds of operations on components, such as division. Intuitively, the operation of division is parametrized by an interaction signature Σ and follows two steps. First, the set of quotients of component A divided by component B is constructed as the set of all components C such that $A = B \times_\Sigma C$. Practically, every element in the set of quotients leads to the same composite behavior captured in A, when composed with B under Σ. Then, one component is chosen from the set of quotients as the result of division. We leave as future work the study of the structure of the set of all quotients and the choice of a specific element from that set to define the operation of division.

2.1.4 A co-inductive construction

In this section, we show how local constraints on observations can be co-inductively *lifted* to global constraints on TESs. We get, as a result, a finite specification of some interaction signatures using simpler relations on observations. Moreover, we get a co-inductive proof mechanism to relate an interaction signature defined on TESs with an interaction signature lifted from constraints on observables, as shown in Lemma 7. Such construction gives, as well, an operational perspective on deriving an interaction signature as a step-wise constraint imposed on observables. Practically, the operational approach of the co-inductive definition is relevant when considering robots and their step-wise decision on their next observation.

The intuition for such construction is that, in some cases, the condition for two TESs to be composable depends only on a composability relation on observations. An example of composability constraint for a robot with its battery and a field enforces that each *move* event *discharges* the battery and *changes* the state of the field. As a result, every *move* event observed by the robot must coincide with a *discharge* event observed by the battery and a change of state observed by the field. The lifting of such composability relation on observations to a constraint on TESs is defined co-inductively. Finally, Lemma 14 gives weaker conditions for Lemma 3 to hold.

Definition 10 (Composability relation on observations)**.** *A composability relation on observations is a parametrized relation κ such that for all pairs $(E_1, E_2) \in \mathcal{P}(\mathbb{E})\times\mathcal{P}(\mathbb{E})$, we have $\kappa(E_1, E_2) \subseteq (\mathcal{P}(E_1) \times \mathbb{R}_+) \times (\mathcal{P}(E_2) \times \mathbb{R}_+)$.*

The following examples define locally on observations some relations analogous to those defined globally on TESs in Example 4 and Example 5.

Example 11. *We give two examples of composability relations on observations:*

- $((O_1, t_1), (O_2, t_2)) \in \kappa^{sync}(E_1, E_2)$ *if and only if every shared event always occurs at the same time, i.e., $t_1 < t_2$ implies $O_1 \cap E_2 = \emptyset$, and $t_2 < t_1$ implies $O_2 \cap E_1 = \emptyset$, and $t_2 = t_1$ implies $O_1 \cap E_2 = O_2 \cap E_1$;*

- $((O_1, t_1), (O_2, t_2)) \in \kappa^{async}(E_1, E_2)$ *if and only if no shared event occurs at the same time, i.e., $t_1 = t_2$ implies $O_1 \cap E_2 = \emptyset = O_2 \cap E_1$.* ■

For two composability relations κ_1, κ_2, their intersection or union, written $\kappa_1 \cap \kappa_2$ and $\kappa_1 \cup \kappa_2$ respectively, is defined, for any $E_1, E_2, E_3 \subseteq \mathbb{E}$, as $(\kappa_1 \cap \kappa_2)(E_1, E_2) = \kappa_1(E_1, E_2) \cap \kappa_2(E_1, E_2)$ and $(\kappa_1 \cup \kappa_2)(E_1, E_2) = \kappa_1(E_1, E_2) \cup \kappa_2(E_1, E_2)$.

Definition 11 (Lifting)**.** *Let κ be a composability relation on observations, and, for any $\mathcal{R} \subseteq TES(E_1) \times TES(E_2)$, let $\Phi_\kappa(E_1, E_2)(\mathcal{R}) \subseteq TES(E_1) \times TES(E_2)$ be such that:*

$$\Phi_\kappa(E_1, E_2)(\mathcal{R}) = \{(\tau_1, \tau_2) \mid (\tau_1(0), \tau_2(0)) \in \kappa(E_1, E_2) \wedge (\tau_1, \tau_2)' \in \mathcal{R})\}$$

The lifting *of κ on TESs, written $[\kappa]$, is the parametrized relation obtained by taking the greatest post fixed point of the function $\Phi_\kappa(E_1, E_2)$ for arbitrary pair $E_1, E_2 \subseteq \mathbb{E}$, i.e., the relation $[\kappa](E_1, E_2) = \bigcup_{\mathcal{R} \subseteq TES(E_1) \times TES(E_2)} \{\mathcal{R} \mid \mathcal{R} \subseteq \Phi_\kappa(E_1, E_2)(\mathcal{R})\}$.*

Lemma 5 (Correctness)**.** *For any $E_1, E_2 \subseteq \mathbb{E}$, the function $\Phi_\kappa(E_1, E_2)$ is monotone, and therefore has a greatest post fixed point.*

Proof. Let κ be a composability relation on observations, and let $E_1, E_2 \subseteq \mathbb{E}$. We recall that the function $\Phi_\kappa(E_1, E_2)$ is such that, for any $\mathcal{R} \subseteq TES(E_1) \times TES(E_2)$:

$$\Phi_\kappa(E_1, E_2)(\mathcal{R}) = \{(\tau_1, \tau_2) \mid \quad (\tau_1(0), \tau_2(0)) \in \kappa(E_1, E_2) \wedge (\tau_1, \tau_2)' \in \mathcal{R})\}$$

Let $\mathcal{R}_1, \mathcal{R}_2 \subseteq TES(E_1) \times TES(E_2)$ be such that $\mathcal{R}_1 \subseteq \mathcal{R}_2$. We show that $\Phi_\kappa(E_1, E_2)(\mathcal{R}_1) \subseteq \Phi_\kappa(E_1, E_2)(\mathcal{R}_2)$. For any $(\tau_1, \tau_2) \in TES(E_1) \times TES(E_2)$,

$$\begin{aligned}
(\tau_1, \tau_2) \in \Phi_\kappa(E_1, E_2)(\mathcal{R}_1) &\iff (\tau_1(0), \tau_2(0)) \in \kappa(E_1, E_2) \wedge (\tau_1, \tau_2)' \in \mathcal{R}_1) \\
&\implies (\tau_1(0), \tau_2(0)) \in \kappa(E_1, E_2) \wedge (\tau_1, \tau_2)' \in \mathcal{R}_2) \\
&\implies (\tau_1, \tau_2) \in \Phi_\kappa(E_1, E_2)(\mathcal{R}_2)
\end{aligned}$$

Therefore, $\mathcal{R}_1 \subseteq \mathcal{R}_2$ implies that $\Phi_\kappa(E_1, E_2)(\mathcal{R}_1) \subseteq \Phi_\kappa(E_1, E_2)(\mathcal{R}_2)$, and we conclude that $\Phi_\kappa(E_1, E_2)$ is monotonic. We use the Knaster-Tarski theorem, where the underlying lattice is the powerset of TESs with inclusion relation, for the existence of a greatest fixed point of the monotonic function $\Phi_\kappa(E_1, E_2)$ applying to that lattice. Thus, $\Phi_\kappa(E_1, E_2)$ has a greatest fixed point defined as:

$$[\kappa](E_1, E_2) = \bigcup \{\mathcal{R} \mid \mathcal{R} \subseteq \Phi_\kappa(E_1, E_2)(\mathcal{R})\}$$

□

Lemma 6. *If κ is a composability relation on observations, then the lifting $[\kappa]$ is a composability relation on TESs. Moreover, if κ is symmetric (as in Definition 5), then $[\kappa]$ is symmetric.*

Proof. We first note that, given a composability relation κ on observables, the lifting $[\kappa]$ is a composability relation on TESs. Indeed, for any pair of interfaces $E_1, E_2 \subseteq \mathbb{E}$, any $(\sigma, \tau) \in [\kappa](E_1, E_2)$ is a pair in $TES(E_1) \times TES(E_2)$.

If κ is symmetric (as in Definition 5), we show that $[\kappa]$ is also symmetric. Given a set $\mathcal{R} \subseteq TES(E_1) \times TES(E_2)$, we use the notation $\overline{\mathcal{R}}$ to denote the smallest set such that $(\sigma, \tau) \in \mathcal{R} \iff (\tau, \sigma) \in \overline{\mathcal{R}}$. Let $E_1, E_2 \subseteq \mathbb{E}$.

If κ is symmetric, then for $\mathcal{R} \subseteq TES(E_1) \times TES(E_2)$,

$$\begin{aligned}
\Phi_\kappa(E_1, E_2)(\mathcal{R}) &= \{(\tau_1, \tau_2) \mid \quad (\tau_1(0), \tau_2(0)) \in \kappa(E_1, E_2) \wedge (\tau_1, \tau_2)' \in \mathcal{R})\} \\
&= \{(\tau_1, \tau_2) \mid \quad (\tau_2(0), \tau_1(0)) \in \kappa(E_2, E_1) \wedge (\tau_1, \tau_2)' \in \mathcal{R})\} \\
&= \{(\tau_1, \tau_2) \mid \quad (\tau_2(0), \tau_1(0)) \in \kappa(E_2, E_1) \wedge (\tau_2, \tau_1)' \in \overline{\mathcal{R}})\} \\
&= \{(\tau_1, \tau_2) \mid \quad (\tau_2, \tau_1) \in \Phi_\kappa(E_2, E_1)(\overline{\mathcal{R}})\} \qquad (1)
\end{aligned}$$

which shows that $[\kappa]$ is symmetric since, for any $E_1, E_2 \subseteq \mathbb{E}$, $[\kappa](E_1, E_2) = \bigcup_{\mathcal{R} \subseteq TES(E_1) \times TES(E_2)} \{\mathcal{R} \mid \mathcal{R} \subseteq \Phi_\kappa(E_1, E_2)(\mathcal{R})\}$, and

$$\begin{aligned}
(\sigma, \tau) \in [\kappa](E_1, E_2) &\iff \exists \mathcal{R}.\ (\sigma, \tau) \in \mathcal{R} \wedge \mathcal{R} \subseteq \Phi_\kappa(E_1, E_2)(\mathcal{R}) \\
&\iff \exists \overline{\mathcal{R}}.\ (\tau, \sigma) \in \overline{\mathcal{R}} \wedge \overline{\mathcal{R}} \subseteq \Phi_\kappa(E_2, E_1)(\overline{\mathcal{R}}) \\
&\iff (\tau, \sigma) \in [\kappa](E_2, E_1)
\end{aligned}$$

where the first equivalence is given by the fact that $[\kappa](E_1, E_2)$ is the greatest post fixed point of $\Phi_\kappa(E_1, E_2)$, the second equivalence is obtained from equality (1), and the third equivalence is given by the fact that $[\kappa](E_2, E_1)$ is the greatest post fixed point.

□

As a consequence of Lemma 6, any composability relation on observations gives rise to a composability relation on TESs. Lemma 7 relates (a)synchornous composability relations on TESs with (a)synchronous composability relations on observations.

Lemma 7. *Let R_{sync} and R_{async} be composability relations defined in Example 4 and Example 5, respectively. Let κ^{sync} be the relation on observations defined in Example 11. For all E_1 and E_2, $R_{sync}(E_1, E_2) = [\kappa^{sync}](E_1, E_2)$ and $R_{async}(E_1, E_2) = [\kappa^{excl}](E_1, E_2)$. When $E_1 \cap E_2 = \emptyset$, then $R_{sync}(E_1, E_2) = R_{async}(E_1, E_2)$.*

Proof. We proof the result for $R_{sync}(E_1, E_2) = [\kappa^{sync}](E_1, E_2)$ and similar reasoning can be applied for $R_{async}(E_1, E_2) = [\kappa^{excl}](E_1, E_2)$. We first show that $R_{sync}(E_1, E_2) \subseteq [\kappa^{sync}](E_1, E_2)$, which is equivalent to showing $R_{sync}(E_1, E_2) \subseteq \Phi_{\kappa^{sync}}(E_1, E_2)(R_{sync}(E_1, E_2))$.

First, we observe that if $(\sigma,\tau) \in R_{sync}(E_1,E_2)$, then $(\sigma,\tau)' \in R_{sync}(E_1,E_2)$, as dropping the first observation(s) of σ and τ preserves the property imposed by R_{sync}. For all $(\sigma,\tau) \in R_{sync}(E_1,E_2)$, by definition of R_{sync} and κ^{sync}, we have that $(\sigma(0),\tau(0)) \in \kappa^{sync}(E_1,E_2)$. Since $\Phi_{\kappa^{sync}}(E_1,E_2)(R_{sync}(E_1,E_2))$ is equal to the set

$$\{(\sigma,\tau) \mid (\tau(0),\sigma(0)) \in \kappa^{sync}(E_1,E_2) \wedge (\sigma,\tau)' \in R_{sync}(E_1,E_2)\}$$

we conclude that $R_{sync}(E_1,E_2) \subseteq \Phi_{\kappa^{sync}}(E_1,E_2)(R_{sync}(E_1,E_2))$.

For the other inclusion, let first introduce $time((\sigma,\tau))$ as the smallest time stamp of the head of both streams σ and τ, i.e., $time((\sigma,\tau)) = \min(t_1,t_2)$ where $t_1 = \mathsf{pr}_2(\sigma)(0)$ and $t_2 = \mathsf{pr}_2(\tau)(0)$. For a pair $(\sigma,\tau) \in [\kappa^{sync}](E_1,E_2)$, we have:

$$\begin{aligned}
&(\sigma(0),\tau(0)) \in \kappa^{sync}(E_1,E_2) \wedge (\sigma,\tau)' \in [\kappa^{sync}](E_1,E_2)\\
&\implies \forall t < time((\sigma,\tau)').\ \sigma(t) \cap E_2 = \tau(t) \cap E_1 \wedge (\sigma,\tau)' \in [\kappa^{sync}](E_1,E_2)\\
&\implies \forall n \in \mathbb{N}. \forall t < time((\sigma,\tau)^{(n)}).\ \sigma(t) \cap E_2 = \tau(t) \cap E_1 \wedge\\
&\qquad\qquad (\sigma,\tau)^{(n)} \in [\kappa^{sync}](E_1,E_2)\\
&\implies \forall t.\ \sigma(t) \cap E_2 = \tau(t) \cap E_1\\
&\implies (\sigma,\tau) \in R_{sync}
\end{aligned}$$

We conclude that $R_{sync} = [\kappa^{sync}]$. □

Example 12 (Intersection)**.** *For any two components $C_1 = (E_1,L_1)$ and $C_2 = (E_2,L_2)$, we define the intersection $C_1 \cap C_2$ to be the component $C_1 \times_{([\kappa^{sync}],[\cap])} C_2 = (E_1 \cup E_2, L)$ with κ^{sync} defined in Example 11.* ■

For the following definitions, let $C_1 = (E_1,L_1)$ and $C_2 = (E_2,L_2)$ be two components, and $\oplus$ be a composition function on TESs. We use $\sqcap \subseteq \mathcal{P}(\mathbb{E}) \times \mathcal{P}(\mathbb{E})$ to range over relations on observables.

Definition 12 (Synchronous observations)**.** *The composability relations introduced in Example 11 can also be extended to synchronize pairs of distinct events. Two observations are synchronous under $\sqcap$ if, intuitively, the two following conditions hold:*

1. *every observable that can compose (under $\sqcap$) with another observable must occur simultaneously with one of its related observables; and*

2. *only an observable that does not compose (under $\sqcap$) with any other observable can happen before another observable, i.e., at a strictly lower time.*

To formalize the conditions above, we introduce the independence relation $ind_\sqcap(X,Y) = \forall x \subseteq X.\forall y \subseteq Y.(x,y) \notin \sqcap$.

The synchronous *composability relation on observations* $\kappa_\sqcap^{sync}(E_1,E_2)$ *is the smallest set such that, for all* $O_1 \subseteq E_1$ *and* $O_2 \subseteq E_2$*:*

- *if* $(O_1,O_2) \in \sqcap \cup (\emptyset,\emptyset)$*, then for all* $(O_1',O_2') \in \mathcal{P}(E_1) \times \mathcal{P}(E_2)$ *such that* $ind_\sqcap(O_1',E_2)$ *and* $ind_\sqcap(E_1,O_2')$ *and for all time stamps* t*, we have* $((O_1 \cup O_1',t),(O_2 \cup O_2',t)) \in \kappa_\sqcap^{sync}(E_1,E_2)$.

- *if* $ind_\sqcap(O_1,E_2)$*, then for all* $O_2' \subseteq E_2$ *and for all* $t_1 < t_2$*, we have* $((O_1,t_1),(O_2,t_2)) \in \kappa_\sqcap^{sync}(E_1,E_2)$*. Reciprocally, if* $ind_\sqcap(E_1,O_2)$ *then for all* $O_1' \subseteq E_1$ *and* $t_2 < t_1$*, we have* $((O_1,t_1),(O_2,t_2)) \in \kappa_\sqcap^{sync}(E_1,E_2)$*;*

Example 13. *Although the relation in Definition 12 is a binary relation on observations, we show in this example how to synchronize multiple events transitively. For instance, consider three components,* $A = (\{a\},L_A)$*,* $B = (\{b\},L_B)$*, and* $C = (\{c\},L_C)$*. Let* $\sqcap$ *be the smallest symmetric relation with* $\{(\{a\},\{b\}),(\{b\},\{c\})\} \subseteq \sqcap$*. Then,* $\kappa_\sqcap^{sync}$ *enforces every observable in* A *and* C *to occur at the same time as an observable in* B*. Let* $\Sigma = ([\kappa_\sqcap^{sync}],\cup)$ *with* $\cup$ *defined in Example 3. Observe that, in general,* $(A \times_\Sigma B) \times_\Sigma C \neq (A \times_\Sigma C) \times_\Sigma B$*. On the left hand side, the product of* A *and* B *synchronizes every occurrence of event* a *with an occurrence of event* b*, which results in observables of the form* $\{a,b\}$ *only (no interleaving is allowed by* $\kappa_\sqcap^{sync}$*). Since* b *and* c *are also related events, the composition with* C *leads to the component with observables* $\{a,b,c\}$*. On the right hand side,* A *and* C *have independent observables and their composition allows for every interleaving. The product with* B*, however, synchronizes every occurrence of event* a *or* c *with an occurrence of event* b*, which results in interleaving of observables* $\{a,b\}$ *and* $\{c,b\}$*. Finally, observe that the component* $(A \times_\Sigma B) \times_\Sigma C$ *transitively synchronizes occurrences of event* a *with occurrences of event* c *through occurrences of event* b*.* ■

The behavior of component $C_1 \times_{([\kappa_\sqcap^{sync}],\oplus)} C_2$ contains TESs obtained from the composition under $\oplus$ of every pair $\sigma_1 \in L_1$ and $\sigma_2 \in L_2$ of TESs that are related by the synchronous composability relation $[\kappa_\sqcap^{sync}]$ which, depending on $\sqcap$, excludes all event occurrences that do not synchronize.[6] Note that in the case where $\sqcap = \{(O,O) \mid O \subseteq E_1 \cup E_2 \setminus \emptyset\}$, then $\kappa_\sqcap^{sync} = \kappa^{sync}$.

[6] If we let $\oplus$ be the element wise set union, define an event as a set of port assignments, and in the pair $([\kappa_\sqcap^{sync}],\oplus)$ let $\sqcap$ be true if and only if all common ports get the same value assigned, then this composition operator produces results similar to the composition operation in Reo [3].

Definition 13 (Mutual exclusion)**.** *Let $\sqcap \subseteq \mathcal{P}(\mathbb{E})^2$ be a relation on observables. We define two observations to be mutually exclusive under the relation $\sqcap$ if no pair of observables in $\sqcap$ can be observed at the same time. The mutually exclusive composability relation $\kappa_{\sqcap}^{excl}$ on observations allows the composition of two observations (O_1, t_1) and (O_2, t_2), i.e., $((O_1, t_1), (O_2, t_2)) \in \kappa_{\sqcap}^{excl}(E_1, E_2)$, if and only if $t_1 = t_2 \implies \neg(O_1 \sqcap O_2)$.*

Example 14. *Following Example 10, we introduce an interaction signature that composes two robot-field subsystems while excluding the possibility for the robots to both observe the same location on their fields. We define $\sqcap = \{(\{(loc(R_1); l)\}, \{(loc(R_2); l)\}) \mid l \in [0; 20] \times [0; 20]\}$ as the set of pairs of observables containing, for both robots R_1 and R_2, an event that displays the same location as the other robot. Let $\Sigma = ([\kappa_{\sqcap}^{excl}], \cup)$ with $\cup$ defined in Example 3. Then, the product of the two subsystems, using the interaction signature Σ, excludes the possibility for the two robots to observe the same location at the same time. Strictly speaking, the exclusion imposed by the interaction signature Σ does not imply that the two robots can not* effectively *be on the same physical location. We show in Section 2.1.6 how, combined with hyper-properties, such interaction signature may imply a safety property.* ■

The behavior of component $C_1 \times_{([\kappa_{\sqcap}^{excl}], \oplus)} C_2$ contains TESs resulting from the composition under $\oplus$ of every pair $\sigma_1 \in L_1$ and $\sigma_2 \in L_2$ of TESs that are related by the mutual exclusion composability relation $[\kappa_{\sqcap}^{excl}]$ which, depending on $\sqcap$, may exclude some simultaneous event occurrences.

The lifting of composability relations distributes across the intersection.[7]

Lemma 8. *For all composability relations κ_1, κ_2 and interfaces E_1, E_2:*

$$[\kappa_1 \cap \kappa_2](E_1, E_2) = [\kappa_1](E_1, E_2) \cap [\kappa_2](E_1, E_2)$$

[7] The lifting does not distribute across the union, however.

Proof.

$$\begin{aligned}
[\kappa_1](E_1,E_2)\cap[\kappa_2](E_1,E_2) =& \bigcup\{\mathcal{R} \mid \mathcal{R}\subseteq\Phi_{\kappa_1}(E_1,E_2)(\mathcal{R})\}\cap \\
& \bigcup\{\mathcal{R} \mid \mathcal{R}\subseteq\Phi_{\kappa_2}(E_1,E_2)(\mathcal{R})\} \\
=& \bigcup\{\mathcal{R} \mid \mathcal{R}\subseteq\Phi_{\kappa_1}(E_1,E_2)(\mathcal{R}) \textit{ and} \\
& \qquad \mathcal{R}\subseteq\Phi_{\kappa_2}(E_1,E_2)(\mathcal{R})\} \\
=& \bigcup\{\mathcal{R} \mid \mathcal{R}\subseteq\Phi_{\kappa_1}(E_1,E_2)(\mathcal{R})\cap\Phi_{\kappa_2}(E_1,E_2)(\mathcal{R})\} \\
=& \bigcup\{\mathcal{R} \mid \mathcal{R}\subseteq\Phi_{\kappa_1\cap\kappa_2}(E_1,E_2)(\mathcal{R})\} \\
=& [\kappa_1\cap\kappa_2](E_1,E_2)\}
\end{aligned}$$

since

$$\begin{aligned}
\Phi_{\kappa_1}(E_1,E_2)(\mathcal{R})\cap\Phi_{\kappa_2}(E_1,E_2)(\mathcal{R}) =& \{(\tau_1,\tau_2) \mid (\tau_1(0),\tau_2(0))\in\kappa_1(E_1,E_2)\wedge \\
& \qquad (\tau_1(0),\tau_2(0))\in\kappa_2(E_1,E_2)\wedge \\
& \qquad (\tau_1,\tau_2)'\in\mathcal{R})\} \\
=& \{(\tau_1,\tau_2) \mid (\tau_1(0),\tau_2(0))\in\kappa_1(E_1,E_2)\cap\kappa_2(E_1,E_2)\wedge \\
& \qquad (\tau_1,\tau_2)'\in\mathcal{R})\} \\
=& \Phi_{\kappa_1\cap\kappa_2}(E_1,\,E_2)(\mathcal{R})
\end{aligned}$$

□

Similarly, we give a mechanism to lift a composition function on observables to a composition function on TESs. Such lifting operation interleaves observations with different time stamps, and composes observations that occur at the same time.

Definition 14 (Lifting - composition function)**.** *Let* $+ : \mathcal{P}(\mathbb{E})\times\mathcal{P}(\mathbb{E})\to\mathcal{P}(\mathbb{E})$ *be a composition function on observables. The* lifting *of* $+$ *to TESs is* $[+] : TES(\mathbb{E})\times TES(\mathbb{E})\to TES(\mathbb{E})$ *such that, for* $\sigma_i\in TES(\mathbb{E})$ *where* $\sigma_i(0)=(O_i,t_i)$ *with* $i\in\{1,2\}$*:*

$$\sigma_1[+]\sigma_2 = \begin{cases} \langle\sigma_1(0)\rangle\cdot(\sigma_1'[+]\sigma_2) & \textit{if } t_1<t_2 \\ \langle\sigma_2(0)\rangle\cdot(\sigma_1[+]\sigma_2') & \textit{if } t_2<t_1 \\ \langle(O_1+O_2,t_1)\rangle\cdot(\sigma_1'[+]\sigma_2') & \textit{otherwise} \end{cases}$$

Definition 14 composes observations only if their time stamp is the same. Alternative definitions might consider time intervals instead of exact times.

Remark 2. *The last clause of Definition 14 considers the case where two observations occur at the same time. Recall that the time of an observation, as introduced earlier, is an abstraction that requires every event of the observation to occur after the events of the previous observation, and before the events of the next observation. Moreover, the time of two related observations, during composition, may be constrained by the interaction signature of the composition. For instance, the synchronous composability relation in Example 4 requires related observations to occur at the same time. Given those two facts, the likelihood that two observations have the same time is non zero.*

Lemma 9. *Let κ_1 and κ_2 be two composability relations and $\times_{([\kappa_1 \cap \kappa_2],\oplus)}$ be a product on components. Then,*

$$C_1 \times_{([\kappa_1 \cap \kappa_2],\oplus)} C_2 = C_1 \times_{([\kappa_1] \cap [\kappa_2],\oplus)} C_2 = (C_1 \times_{([\kappa_1],\oplus)} C_2) \cap (C_1 \times_{([\kappa_2],\oplus)} C_2)$$

Proof. Let $C_1 \times_{[\kappa_1 \cap \kappa_2],\oplus} C_2 = (E, L)$ and $(C_1 \times_{([\kappa_1],\oplus)} C_2) \cap (C_1 \times_{([\kappa_2],\oplus)} C_2) = (E', L')$. We have $E = E_1 \cup E_2 = E'$. We show $L = L'$.

$$\begin{aligned} L &= \{\sigma_1 \oplus \sigma_2 \mid \sigma_1 \in L_1, \sigma_2 \in L_2, (\sigma_1, \sigma_2) \in [\kappa_1 \cap \kappa_2](E_1, E_2)\} \\ &= \{\sigma_1 \oplus \sigma_2 \mid \sigma_1 \in L_1, \sigma_2 \in L_2, (\sigma_1, \sigma_2) \in [\kappa_1](E_1, E_2) \cap [\kappa_2](E_1, E_2)\} \\ &= L' \end{aligned}$$

□

Example 15. *Composability relations as defined in Definition 12 and Definition 13 can be combined to form new relations, and therefore new products. The behavior of component $C_1 \times_{([\kappa_\sqcap^{sync} \cap \kappa_\sqcap^{excl}]),\oplus)} C_2$ contains all TESs that are in the behavior of both $C_1 \times_{([\kappa_\sqcap^{sync}],\oplus)} C_2$ and $C_1 \times_{([\kappa_\sqcap^{excl}],\oplus)} C_2$, which excludes observations containing an occurrence of at least one event related by $\sqcap$.* ■

Co-inductive constructions of interaction signatures make proving algebraic results of Lemma 3 easier. Lemma 10 gives sufficient conditions to lift, by co-induction, properties of the underlying relation and composition function to meet the conditions of Lemma 3.

Lemma 10. *Let $\Sigma = ([\kappa], [+])$ be an interaction signature with $+$ be a composition function on observables and let κ be a composability relation on observations. Then,*

- $\times_\Sigma$ *is* commutative *if κ is symmetric and $+$ is commutative;*

- $\times_\Sigma$ *is* associative *if* $+$ *is associative and, for all* E_1, E_2, E_3 *and for all triple* $O_i \in \mathcal{P}(E_i)$ *with* $i \in \{1,2,3\}$ *and for all* $t \in \mathbb{R}_+$*:*

$$((O_1,t),(O_2,t)) \in \kappa(E_1,E_2) \wedge ((O_1+O_2,t),(O_3,t)) \in \kappa(E_1 \cup E_2, E_3)$$

if and only if

$$((O_2,t),(O_3,t)) \in \kappa(E_2,E_3) \wedge ((O_1,t),(O_2+O_3,t)) \in \kappa(E_1, E_2 \cup E_3)$$

with κ *such that* $((O_1,t_1),(O_2,t_2)) \in \kappa(E_1,E_2)$ *implies*

1. $((O_1,t_1),(\emptyset,t_2)) \in \kappa(E_1,E_2)$ *if* $t_1 < t_2$*; and*
2. $((\emptyset,t_1),(O_2,t_2)) \in \kappa(E_1,E_2)$ *if* $t_2 < t_1$*;*

and with $O + \emptyset = \emptyset + O = O$ *for all* $O \subseteq \mathcal{P}(\mathbb{E})$*.*

- $\times_\Sigma$ *is* idempotent *if* $+$ *is idempotent and, for all* $E \subseteq \mathbb{E}$ *we have* $((O_1,t_1),(O_2,t_2)) \in \kappa(E,E) \implies (O_1,t_1) = (O_2,t_2)$*.*

Proof. Commutativity. From Lemma 6, if κ is symmetric, then its lifting $[\kappa]$ is also symmetric. Therefore, it is sufficient for κ to be symmetric and for $+$ to be commutative in order for $[\kappa]$ to be symmetric and $[+]$ to be commutative, and therefore $\times_{([\kappa],[+])}$ to be commutative.

Associativity. We recall that, for a TES σ, the domain of σ is the collection of all time stamps of observations, i.e., $dom(\sigma) = \{t \mid \exists i \in \mathbb{N}.\sigma(i) = (O,t)\}$. We define a domain equalizer function $\equiv$ that, given a tuple $(\sigma_1,...,\sigma_n)$ returns a new tuple $(\sigma_1,...,\sigma_n)_\equiv = (\tau_1,...,\tau_n)$ such that the domains of TESs τ_i are the same, i.e., $\exists c.dom(\tau_i) = c$; and τ_i differs with σ_i only by empty observations, i.e., for all $t \in \mathbb{R}_+$, $\sigma_i(t) = \tau_i(t)$. Thus, the operation $\equiv$ adds silent observations to all TESs of the tuple such that the resulting domain for all τ_i is equal.

Let us first introduce a property induced by the assumptions on the composability relation κ. The fact that, forall $O_1 \subseteq E_1$ and $O_2 \subseteq E_2$, $((O_1,t_1),(O_2,t_2)) \in \kappa(E_1,E_2) \wedge t_1 < t_2 \iff ((O_1,t_1),(\emptyset,t_1)) \in \kappa(E_1,E_2)$ implies that a pair of TESs (σ_1,σ_2) is related by $[\kappa]$ if and only if their extension with silent observations on equal domains, i.e., $(\sigma_1,\sigma_2)_\equiv$, is also related by $[\kappa]$. We show by co-induction such property.

For $R \subseteq TES(E_1) \times TES(E_2)$, let

$$\begin{aligned}\Phi^{\equiv}_{k}(R) = \{(\sigma_1,\sigma_2) \mid\ & \sigma_1(0) = (O_1,t_1) \wedge \sigma_2(0) = (O_2,t_2) \wedge \\ & t_1 < t_2 \implies (\sigma_1(0),(\emptyset,t_1)) \in \kappa(E_1,E_2) \wedge \\ & t_2 < t_1 \implies ((\emptyset,t_2),\sigma_2(0)) \in \kappa(E_1,E_2) \wedge \\ & t_1 = t_2 \implies (\sigma_1(0),\sigma_2(0)) \in \kappa(E_1,E_2) \wedge \\ & (\sigma_1,\sigma_2)' \in R\}\end{aligned}$$

The function $\Phi^{\equiv}_{\kappa}$ is a monotonous function applied on a complete lattice. By Knaster Tarksi theorem, $\Phi^{\equiv}_{\kappa}$ has a greatest fixed point that we call $[\kappa_{\equiv}]$. Given the assumption on κ, we can show that $[\kappa] \subseteq \Phi^{\equiv}_{\kappa}([\kappa])$ and $[\kappa_{\equiv}] \subseteq \Phi_{\kappa}([\kappa_{\equiv}])$, which implies that $[\kappa] = [\kappa_{\equiv}]$. It follows that $(\sigma_1,\sigma_2) \in [\kappa] \iff (\sigma_1,\sigma_2)_{\equiv} \in [\kappa]$.

A sufficient condition for the product $\times_{([\kappa],[+])}$ to be associative is that $+$ is associative and for every $\sigma_i \in TES(E_i)$ for $i \in \{1,2,3\}$:

$$\begin{aligned}P_1 :=& (\sigma_1,\sigma_2) \in [\kappa](E_1,E_2) \wedge (\sigma_1[+]\sigma_2,\sigma_3) \in [\kappa](E_1 \cup E_2,E_3) \iff \\ & (\sigma_2,\sigma_3) \in [\kappa](E_2,E_3) \wedge (\sigma_1,\sigma_2[+]\sigma_3) \in [\kappa](E_1,E_2 \cup E_3)\end{aligned}$$

which is equivalent to

$$\begin{aligned}& (\sigma_1,\sigma_2) \in [\kappa_{\equiv}](E_1,E_2) \wedge (\sigma_1[+]\sigma_2,\sigma_3) \in [\kappa_{\equiv}](E_1 \cup E_2,E_3) \iff \\ & (\sigma_2,\sigma_3) \in [\kappa_{\equiv}](E_2,E_3) \wedge (\sigma_1,\sigma_2[+]\sigma_3) \in [\kappa_{\equiv}](E_1,E_2 \cup E_3)\end{aligned}$$

and with similar arguments as before, is equivalent to

$$\begin{aligned}& (\tau_1,\tau_2) \in [\kappa_{\equiv}](E_1,E_2) \wedge (\tau_1[+]\tau_2,\tau_3) \in [\kappa_{\equiv}](E_1 \cup E_2,E_3) \iff \\ & (\tau_2,\tau_3) \in [\kappa_{\equiv}](E_2,E_3) \wedge (\tau_1,\tau_2[+]\tau_3) \in [\kappa_{\equiv}](E_1,E_2 \cup E_3)\end{aligned}$$

with $(\tau_1,\tau_2,\tau_3) = (\sigma_1,\sigma_2,\sigma_3)_{\equiv}$.

As we can assume that all TESs in a triple satisfying P_1 have the same domain, it is sufficient to show that, for all $O_i \subseteq E_i$ with $i \in \{1,2,3\}$ and all $t \in \mathbb{R}_+$:

$$((O_1,t),(O_2,t)) \in \kappa(E_1,E_2) \wedge ((O_1+O_2,t),(O_3,t)) \in \kappa(E_1 \cup E_2,E_3)$$

if and only if

$$((O_2,t),(O_3,t)) \in \kappa(E_2,E_3) \wedge ((O_1,t),(O_2+O_3,t)) \in \kappa(E_1,E_2 \cup E_3)$$

which is assumed by κ. Thus, given the properties of κ, P_1 holds.

Finally, we prove that if $+$ is associative, then $[+]$ is associative. Let $\sigma_i \in L_i$ and we write $\sigma_i(0) = (O_i, t_i)$ for $i \in \{1,2,3\}$, then:

$$\sigma_1[+](\sigma_2[+]\sigma_3) = \begin{cases} \langle (O_1,t_1)\rangle \cdot (\sigma_1'[+](\sigma_2[+]\sigma_3) & \text{if } t_1 < t_2, t_3 \\ \langle (O_2,t_2)\rangle \cdot (\sigma_1[+](\sigma_2'[+]\sigma_3) & \text{if } t_2 < t_1, t_3 \\ \langle (O_3,t_3)\rangle \cdot (\sigma_1[+](\sigma_2[+]\sigma_3') & \text{if } t_3 < t_2, t_1 \\ \langle (O_1+O_2,t_1)\rangle \cdot (\sigma_1'[+](\sigma_2'[+]\sigma_3) & \text{if } t_1 = t_2 < t_3 \\ \langle (O_2+O_3,t_2)\rangle \cdot (\sigma_1[+](\sigma_2'[+]\sigma_3') & \text{if } t_2 = t_3 < t_1 \\ \langle (O_1+O_3,t_1)\rangle \cdot (\sigma_1'[+](\sigma_2[+]\sigma_3') & \text{if } t_1 = t_3 < t_2 \\ \langle (O_1+(O_2+O_3),t_1)\rangle \cdot (\sigma_1'[+](\sigma_2'[+]\sigma_3') & \text{if } t_1 = t_3 = t_2 \end{cases}$$

The only case that differs from $(\sigma_1[+]\sigma_2)[+]\sigma_3$ is when $t_1 = t_3 = t_2$, which gives $((O_1+O_2)+O_3,t_1)$. Thus, if $((O_1+O_2)+O_3,t_1) = (O_1+(O_2+O_3),t_1)$ for every $O_i \in \mathcal{P}(E_i)$ with $i \in \{1,2,3\}$, then $\sigma_1[+](\sigma_2[+]\sigma_3) = \sigma_1[+](\sigma_2[+]\sigma_3)$ for every $\sigma_i \in L_i$ with $i \in \{1,2,3\}$.

Idempotency. If $+$ is idempotent, then the lifting $[+]$ is also idempotent. We consider $+$ to be idempotent. We show that, for all $E \subseteq \mathbb{E}$ and $o_1, o_2 \in \mathcal{P}(E) \times \mathbb{R}_+$ we have $(o_1,o_2) \in \kappa(E,E) \implies o_1 = o_2$, then for all $\sigma, \tau \in TES(E)$, $(\sigma,\tau) \in [\kappa](E,E) \implies \sigma = \tau$, which is a sufficient condition for $\times_{([\kappa],[+])}$ to be idempotent.

By definition $[\kappa](E,E)$ is the greatest fixed point of the function:

$$\begin{aligned} \Phi_\kappa(E,E)(\mathcal{R}) &= \{(\tau_1,\tau_2) \mid \quad (\tau_1(0),\tau_2(0)) \in \kappa(E,E) \wedge (\tau_1,\tau_2)' \in \mathcal{R})\} \\ &\subseteq \{(\tau_1,\tau_2) \mid \quad \tau_1(0) = \tau_2(0) \wedge (\tau_1',\tau_2') \in \mathcal{R}\} \end{aligned}$$

Therefore, we conclude that $[\kappa](E,E) \subseteq \{(\sigma,\sigma) \mid \sigma \in TES(E)\}$. □

The conditions exposed in Lemma 10 are applicable for the case of the join product, as shown in Theorem 1.

Theorem 1. *The product $\bowtie$ of Example 7 is commutative, associative, and idempotent.*

Proof. Commutativity and idempotency of $\bowtie$ are following from κ^{sync} being symmetric and satisfying the condition for idempotency.

Assume that $((O_1,t),(O_2,t)) \in \kappa^{sync}(E_1,E_2) \wedge ((O_1 \cup O_2,t),(O_3,t)) \in \kappa^{sync}(E_1 \cup E_2,E_3)$ holds, then $O_1 \cap E_2 = O_2 \cap E_1 \wedge (O_1 \cup O_2) \cap E_3 = O_3 \cap (E_1 \cup E_2)$ is true by definition of κ^{sync}.

We first observe that $(O_1 \cup O_2) \cap E_3 \cap E_2 = O_3 \cap (E_1 \cup E_2) \cap E_2$ implies that $O_2 \cap E_3 = O_3 \cap E_2$. Then, $(O_1 \cup O_2) \cap E_3 \cap E_1 = O_3 \cap (E_1 \cup E_2) \cap E_1$ implies that $O_1 \cap E_3 = O_3 \cap E_1$, using $O_1 \cap E_2 = O_2 \cap E_1$ we conclude that $O_1 \cap (E_1 \cup E_3) = (O_2 \cup O_3) \cap E_1$.

Thus, we showed that

$$((O_1,t),(O_2,t)) \in \kappa^{sync}(E_1,E_2) \wedge ((O_1 \cup O_2,t),(O_3,t)) \in \kappa^{sync}(E_1 \cup E_2,E_3)$$

if and only if

$$((O_2,t),(O_3,t)) \in \kappa^{sync}(E_2,E_3) \wedge ((O_1,t),(O_2 \cup O_3,t)) \in \kappa^{sync}(E_1,E_2 \cup E_3)$$

for all $O_i \subseteq E_i$ and $t \in \mathbb{R}_+$. Finally, by definition, κ_{sync} is such that, for all $O_1 \subseteq E_1$ and $O_2 \subseteq E_2$:

1. $((O_1,t_1),(O_2,t_2)) \in \kappa_{sync}(E_1,E_2)$ and $t_1 < t_2$ if and only if $((O_1,t_1),(\emptyset,t_1)) \in \kappa_{sync}(E_1,E_2)$; and

2. $((O_1,t_1),(O_2,t_2)) \in \kappa_{sync}(E_1,E_2)$ and $t_2 < t_1$ if and only if $((\emptyset,t_2),(O_2,t_2)) \in \kappa_{sync}(E_1,E_2)$.

Given that $\cup$ is associative and $O \cup \emptyset = O$ for all O, we conclude that $\bowtie$ is associative. □

We give in Lemma 11 some conditions for two products to distribute, and in Lemma 12 some conditions to extend the underlying relation on observables for a synchronous composability relation.

Lemma 11. *Let C_1, C_2, and C_3 be three components, and let κ_1 and κ_2 be two composability relations on observables such that for all $\sigma_1,\sigma_2,\sigma_3 \in L_1 \times L_2 \times L_3$:*

- *$(\sigma_1,\sigma_2[\cup]\sigma_3) \in [\kappa_1]$ if and only if $(\sigma_1,\sigma_2) \in [\kappa_1]$ and $(\sigma_1,\sigma_3) \in [\kappa_1]$, and*

- *for all $\tau_1 \in L_1$, $(\tau_1[\cup]\sigma_2,\sigma_1[\cup]\sigma_3) \in [\kappa_2]$ if and only if $(\sigma_2,\sigma_3) \in [\kappa_2]$ and $\sigma_1 = \tau_1$.*

Then,

$$C_1 \times_{[\kappa_1]} (C_2 \times_{[\kappa_2]} C_3) = (C_1 \times_{[\kappa_1]} C_2) \times_{[\kappa_2]} (C_1 \times_{[\kappa_1]} C_3)$$

Proof. Let L be the behavior of component $(C_1 \times_{[\kappa_1]} C_2) \times_{[\kappa_2]} (C_1 \times_{[\kappa_1]} C_3)$, L' be the behavior of $C_1 \times_{[\kappa_1]} (C_2 \times_{[\kappa_2]} C_3)$, L_{12} be the behavior of $(C_1 \times_{[\kappa_1]} C_2)$ and L_{13} be the behavior of $(C_1 \times_{[\kappa_1]} C_3)$. Then,

$$\begin{aligned}
L =&\{\sigma_1[\cup](\sigma_2[\cup]\sigma_3) \mid \sigma_1 \in L_1, \sigma_2 \in L_2, \sigma_3 \in L_3, \\
&\qquad (\sigma_1, \sigma_2[\cup]\sigma_3) \in [\kappa_1],\ (\sigma_2, \sigma_3) \in [\kappa_2]\} \\
=&\{\sigma_1[\cup](\sigma_2[\cup]\sigma_3) \mid \sigma_1 \in L_1, \sigma_2 \in L_2, \sigma_3 \in L_3, \\
&\qquad (\sigma_1, \sigma_2) \in [\kappa_1], (\sigma_1, \sigma_3) \in [\kappa_1], (\sigma_2, \sigma_3) \in [\kappa_2]\} \\
=&\{\sigma[\cup]\tau \mid \sigma \in L_{12}, \tau \in L_{13}, (\sigma, \tau) \in [\kappa_2]\} \\
=&L'
\end{aligned}$$

□

Lemma 12. *Let $C_1 = (E_1, L_1)$ and $C_2 = (E_2, L_2)$ be two components. Let $\kappa_\sqcap^{sync}$ be a composability relation on observables with $\sqcap \subseteq \mathcal{P}(E_1) \times \mathcal{P}(E_2)$. Then, for any $\sqcap'$ with $\sqcap' \cap (\mathcal{P}(E_1) \times \mathcal{P}(E_2)) = \emptyset$, then:*

$$C_1 \times_{[\kappa_\sqcap^{sync}]} C_2 = C_1 \times_{[\kappa_{\sqcap\cup\sqcap'}^{sync}]} C_2$$

Proof. For any pair of composable observations $((O_1, t_1), (O_2, t_2)) \in \kappa_\sqcap^{sync}$, we have that $((O_1, t_1), (O_2, t_2)) \in \kappa_{\sqcap\cup\sqcap'}^{sync}$ since $(O_1, O_2) \in \sqcap$ implies that $(O_1, O_2) \in \sqcap \cup \sqcap'$. Conversely, if $(O_1, O_2) \in \sqcap \cup \sqcap'$ and $\sqcap' \cap \mathcal{P}(E_1) \times \mathcal{P}(E_2) = \emptyset$, then $(O_1, O_2) \in \sqcap$. Thus, for any (σ_1, σ_2), $(\sigma_1, \sigma_2) \in [\kappa_\sqcap^{sync}]$ if and only if $(\sigma_1, \sigma_2) \in [\kappa_{\sqcap\cup\sqcap'}^{sync}]$. □

2.1.5 Properties of TESs

We distinguish two kinds of properties of TESs: properties that we call *trace properties*, and properties on sets of TESs that we call *behavior properties*, which correspond to hyper-properties in [28]. The generality of our model permits to interchangeably construct a component from a property and extract a property from a component. As illustrated in Example 18, when composed with a set of interacting components, a component property constrains the components to only expose desired behavior (i.e., behavior in the property). In Section 2.1.6, we provide more intuition for the practical relevance of these properties.

Definition 15. *A trace property P is a subset $P \subseteq TES(E)$ for some set of events E. A component $C = (E, L)$ satisfies a property P, if $L \subseteq P$, which we denote as $C \models P$.*

Example 16. *We distinguish the usual* safety *and* liveness *properties [2, 28], and recall that every trace property can be written as the intersection of a safety and a liveness property. Let X be an arbitrary set, and P be a subset of $\mathbb{N} \to X$. Intuitively, P is safe if every* bad *stream not in P has a finite prefix every completion of which is bad, hence not in P. A property P is a liveness property if every finite sequence in X^* can be completed to yield an infinite sequence in P, where X^* is the set of all finite sequences of elements in X. For instance, the property of terminating behavior for a component with interface E is a liveness property, defined as:*

$$P_{finite}(E) = \{\sigma \in TES(E) \mid \exists n \in \mathbb{N}. \forall i > n.\ \mathsf{pr}_1(\sigma)(i) = \emptyset\}$$

$P_{finite}(E)$ says that, for every finite prefix of any stream in $TES(E)$, there exists a completion of that prefix with an infinite sequence of silent observations $\emptyset$ in $P_{finite}(E)$. ■

Example 17. *A trace property is similar to a component, since it describes a set of TESs, except that it is* a priori *not restricted to any interface* [8]*. A trace property P can then be turned into a component, by constructing the smallest interface E_P such that, for all $\sigma \in P$, and $i \in \mathbb{N}$, $\mathsf{pr}_1(\sigma)(i) \subseteq E_P$. The component $C_P = (E_P, P)$ is then the componentized-version of property P.* ■

Lemma 13. *Given a property P over E, its componentized-version C_P (see Example 17) and a component $C = (E, L)$, then $C \models P$ if and only if $C \cap C_P = C$.*

Proof. We recall the definition of the intersection in Example 12. For any two components $C_1 = (E_1, L_1)$ and $C_2 = (E_2, L_2)$, the intersection $C_1 \cap C_2$ is the component $C_1 \times_{([\kappa^{sync}],[\cap])} C_2 = (E_1 \cup E_2, L)$. Given that $\cap$ satisfies the condition for using Lemma 10, the product $\cap$ is idempotent. Let $C \cap C_P = (E, L')$. If $(\sigma, \tau) \in L'$ then $\sigma = \tau$. Thus, $L' \subseteq L \cap L_P$.

Alternatively, let $\sigma \in L \cap L_P$. We observe that at any point $n \in \mathbb{N}$, we have $(\sigma(n), \sigma(n)) \in \kappa^{sync}(E, E)$. Therefore, $(\sigma, \sigma) \in [\kappa^{sync}_{\sqcap}]$.

We conclude that $L \cap L_P = L'$. □

[8] In our formalism, a property is a set of TESs $L \subseteq TES(E)$ for some $E \subseteq \mathbb{E}$. Two properties P and L are equal if they contain identical TESs, and equality is not subject to the interface over which properties are defined.

Example 18. *We use the term* coordination property *to refer to a property used in order to coordinate behaviors. Given a set of n components $C_i = (E_i, L_i)$, $i \in \{1, ..., n\}$, a coordination property Coord for the composed components is a property over events $E = E_1 \cup ... \cup E_n$, i.e., $Coord \subseteq TES(E)$.*

Consider the synchronous interaction, as introduced in Example 4, of the n components and let $C = C_1 \bowtie C_2 \bowtie ... \bowtie C_n$ be their synchronous product. Typically, a coordination property will not necessarily be satisfied by the composite component C, but some of the behavior of C is contained in the coordination property. The coordination problem is to find (e.g., synthesize) an orchestrator component $Orch = (E_O, L_O)$ such that $C \bowtie Orch \models Coord$. The orchestrator restricts the component C to exhibit only the subset of its behavior that satisfies the coordination property. In other words, in their composition, Orch coordinates C to satisfy Coord. As shown in Example 17, since Coord ranges over the same set E that is the interface of component $C_1 \bowtie C_2 \bowtie ... \bowtie C_n$, a coordination property can be turned into an orchestrator by building its corresponding component. The coordination problem can be made even more general by changing the composability relations or the composition functions used in the construction of C. ■

Trace properties are not sufficient to fully capture the scope of interesting properties of components of cyber-physical systems. Some of their limitations are highlighted in Section 2.1.6. To address this issue, we introduce *behavior properties*, which are strictly more expressive than trace properties, and give two illustrative examples.

Definition 16. *A behavior property ϕ over a set of events E is a hyper-property $\phi \subseteq \mathcal{P}(TES(E))$. A component $C = (E, L)$ satisfies a hyper-property ϕ if $L \in \phi$, which we denote as $C \models\!\models \phi$.*

Example 19. *A component $C = (E, L)$ can be oblivious to time. Any sequence of time-stamps for an acceptable sequence of observables is acceptable in the behavior of such a component. This "obliviousness to time" property is not a trace property, but a hyper-property, defined as:*

$$\phi_{shift}(E) := \{Q \subseteq TES(E) \mid \forall \sigma \in Q. \forall t \in OS(\mathbb{R}_+). \exists \tau \in Q. \\ \mathsf{pr}_1(\sigma) = \mathsf{pr}_1(\tau) \wedge \mathsf{pr}_2(\tau) = t\}$$

Intuitively, if $C \models\!\models \phi_{shift}(E)$, then C is independent of time. ■

Example 20. *We use $\phi_{insert}(X, E)$ to denote the hyper-property that allows for arbitrary insertion of observations in $X \subseteq \mathcal{P}(E)$ into every TES at any point in time,*

i.e., the set defined as:

$$\{Q \subseteq TES(E) \mid \forall \sigma \in Q.\forall i \in \mathbb{N}.\exists \tau \in Q.\exists x \in X. \\ (\forall j < i.\sigma(j) = \tau(j)) \wedge \\ (\exists t \in \mathbb{R}_+.\tau(i) = (x,t)) \wedge \\ (\forall j \geq i.\tau(j+1) = \sigma(j))\}$$

Intuitively, elements of $\phi_{insert}(X,E)$ are closed under insertion of an observation $x \in X$ at an arbitrary time. ■

2.1.6 Components of the running example

This section is inspired by the work on soft-agents [78, 45], and elaborates on the more intuitive version that we presented in Section 1.3. Interactive cyber-physical systems are represented as components, and behavioral properties of those systems are formulated as components, as well. Through these examples, we show how we use component-based descriptions to model a simple scenario of a robot roaming around in a field while taking energy from its battery. We structurally separate the battery, the robot, and the field as independent components, and we explicitly model their interaction in a specific composed system.

Example 21 (Roaming robots)**.** *We capture, as a component, sequences of observations emerging from discrete actions of a robot at a fixed time rate. For simplicity, we consider that the robot can perform actions of two types only: a move in a cardinal direction, and a read of its position sensor. A move action of robot i creates an event of the form $d(i,p)$ where d is the direction, and p is the power displayed by the robot. The read action of robot i generates an event of the form $read(i,(x;y))$ where $(x;y)$ is a coordinate location.*

Formally, we write $R(i,T,P) = (E_R(i,P), L_R(T))$ for the robot component with identifier i with $E_R(i,P)$ the set

$$\{S(i,p), W(i,p), N(i,p), E(i,p), read(i,(x;y)) \mid x,y \in [\![-20,20]\!], p \leq P\}$$

and $L_R(T) \subseteq TES(E_R(i,P))$ be such that all observations are time stamped with a multiple of the period $T \in \mathbb{R}_+$, i.e., for all $\sigma \in L_R(T)$, if $(O,t) \in \sigma$ then there exists $k \in \mathbb{N}$ such that $t = k \cdot T$. The component $R(i,T)$ therefore captures all robots whose directions are restricted to S(outh), W(est), N(orth), and E(ast), whose power

Table 2.1: Three prefixes of timed-event streams for $R(1,T,P)$, $R(2,T,P)$, and $R(3,T,P)$, where T and P are fixed, and each move action consumes the same power p.

t/T	$\sigma : R(1,T,P)$	$\tau : R(2,T,P)$	$\delta : R(3,T,P)$
1	$\{N(1,p)\}$	–	–
2	$\{W(1,p)\}$	–	–
3	$\{W(1,p)\}$	$\{N(2,p)\}$	$\{E(3,p)\}$
4	$\{S(1,p)\}$	$\{W(2,p)\}$	$\{E(3,p)\}$
...	...	...	–

is limited to P, and whose location values are integers in the interval $[\![-20,20]\!]$. The robot stops whenever $p = 0$.

In Table 2.1, we display the prefix of one TES from the behavior of three robot components. Note that each line corresponds to a time instant, for which each robot may or may not have observed events. The symbol '–' represents no observable, while otherwise we show the set of events observed. The time column is factorized by the period T, shared by all robots. Thus, at time $3 \cdot T$, robot $R(1,T,P)$ moves west, while robot $R(2,T,P)$ moves north, and robot $R(3,T,P)$ moves east, all with power p. We use $R(i)$ to denote the robot $R(i,T,P)$ with a fixed by arbitrary period T and upper power P. ■

In the robot component of Example 21, observations occur at a fixed frequency. For some physical components, however, observations may occur at any point in time. For instance, consider the field on which the robot moves. Each time a robot moves may induce a change in the field's state, and the field's state may be observable at any time and frequency. Internally, the field may record its state changes by a continuous function, while restricting the possibilities of the robots to move due to physical limitations. We describe the field on which the robot moves as a component, and we specify, in Example 24, how robot components interact with the field component.

Example 22 (Field)**.** *The field component captures, in its behavior, the dynamics of its state as a sequence of observations. The collection of objects on a field is given by the set I. The state of a field is a triple $(((x;y)_i)_{i\in I}, (\overrightarrow{v_i})_{i\in I}, t)$ that describes, at time $t \in \mathbb{R}_+$, the position $(x;y)_i$ and the velocity $\overrightarrow{v_i}$ of each object in I. We model each object in I by a square of dimension 1 by 1, and the coordinate $(x;y)_i$ represents the central position of the square. We use $\mu = (((x_0,y_0)_i)_{i\in I}, (\overrightarrow{v_{0i}})_{i\in I}, t_0)$ as an initial state for the field, which gives for each robot in $i \in I$ a position and an initial velocity. Note that static obstacles on the field can be modeled as objects $i \in I$ with position $(x;y)_i$ and zero velocity.*

Formally, the field component is the pair $F_\mu(I) = (E_F(I), L_F(I,\mu))$ *with*

$$E_F(I) = \{(x;y)_i, move(i,\overrightarrow{v}) \mid i \in I,\ x,y \in \mathbb{R},\ \overrightarrow{v} \in \mathbb{R}\times\mathbb{R}\}$$

where each event $move(i,\overrightarrow{v})$ *continuously moves object* i *with velocity* $\overrightarrow{v}$*, and event* $(x;y)_i$ *displays the location of object* i *on the field.*

The set $L_F(I,\mu) \subseteq TES(E_F(I))$ *captures all sequences of observations that consistently sample trajectories of each objects in* I*, according to the change of state of the field and the internal constraint. As a physical constraint, we impose that no two objects can overlap, i.e., for any disjoint* $i,j \in I$ *and for all time* $t \in \mathbb{R}_+$*, with* $(x;y)_i$ *and* $(u;v)_j$ *their respective positions, then* $[x-0.5,x+0.5]\cap[u-0.5,u+0.5] = [y-0.5,y+0.5]\cap[v-0.5,v+0.5] = \emptyset$*. Even though the mechanism for such a constraint is hidden in the field component, typically, the move of a robot is eventually limited by the physics of the field. We write* $F(I)$ *when* μ *is fixed but arbitrary.* ■

There is a fundamental difference between the robot component in Example 21 and the field component in Example 22. The robot component has an adequate underlying sampling frequency which prevents missing any event if observations are made by that frequency. However, the field has no such frequency for its observations, which means that there may be another intermediate observation occurring between any two observations. In [57], we capture, as a behavioral property, the property for a component to interleave observations between any two observation.

Example 23 (Protocol)**.** *As shown in Example 22, physics may impose some constraints that force robots to coordinate. A protocol is a component that, for instance, coordinates the synchronous movement of a pair of robots. For example, when two robots face each other on the field, the* $swap(i,j)$ *protocol moves robot* $R(i,P,T)$ *north, west, and then south, as it moves robot* $R(j,P,T)$ *east. Note that the protocol requires the completion of a sequence of moves to succeed. Another robot could be in the way, and therefore delay the last observables of the sequence. The swap component is defined by* $swap(i,j) = (E_P(i,j), L_P(i,j))$ *where* $E_P(i,j) = E_R(i,P) \cup E_R(j,P)$ *and* $L_P(i,j)$ *captures all sequences of observations where the two robots* i *and* j *swap positions.* ■

A useful interaction signature Σ is the one that synchronizes shared events between two components. We write $\Sigma_{sync} = (R_{sync}, \cup)$ for such interaction signature, and give its specification in Example 4. The synchronous interaction signature $\Sigma_{sync} = (R_{sync}, \cup)$ leads to the product $\times_{\Sigma_{sync}}$ that forces two components to observe shared events at the same time. We write $\bowtie$ for such product. As a result, $R(1,P,T) \bowtie$

Table 2.2: Three prefixes of timed-event streams for $R(1,T,P)$, $R(2,T,P)$, and $R(3,T,P)$, together with the prefix resulting from forming their synchronous product with the swap protocol. Initially, $\mu(1) = (0;2)$, $\mu(2) = (0;1)$, $\mu(3) = (0;0)$.

t/T	$\eta : R(1,P,T) \bowtie R(2,P,T) \bowtie R(3,P,T) \bowtie swap(2,3)$
1	$\{N(1,p)\}$
2	$\{W(1,p)\}$
3	$\{W(1,p), N(2,p)\}$
4	$\{S(1,p), W(2,p), E(3,p)\}$
5	$\{S(2,p)\}$
...	...

$R(2,P,T) \bowtie R(3,P,T) \bowtie swap(2,3)$ captures all sequences of moves for the robots constrained by the *swap* protocol, as shown by the elements of table 2.2.

Example 24 (Field-Robot signature)**.** *The interactions occurring between the field and the robot components impose simultaneity on some disjoint events. For instance, every observation of the robot containing the event $d(i,p) \in E_R(i,P)$ must occur at the same time as an observation of the field containing the event $move(i, \overrightarrow{v(d,p)}) \in E_F(I)$ with $\overrightarrow{v(d,p)}$ returning the velocity as a function of direction d and power p. Also, every observation containing the event $read(i, (\lfloor x \rfloor, \lfloor y \rfloor)) \in E_R(i,P)$ must occur at the same time as an event $(x;y)_i \in E_F(I)$ where $\lfloor z \rfloor$ gives the integer part of z.*

Formally, we capture such interaction in the interaction signature $\Sigma_{RF} = (R_{RF}, \cup)$, where R_{RF} is the smallest symmetric relation defined as for all $(\tau, \sigma) \in R_{RF}$, for all $t \in \mathbb{R}_+$, for all $i \in \mathbb{N}$,

$$read(i,(n,m)) \in \tau(t) \iff (\exists (x;y)_i \in \sigma(t) \wedge n = \lfloor x \rfloor \wedge m = \lfloor y \rfloor)$$

and $d(i,p) \in \tau(t) \iff move(i, \overrightarrow{v(d,p)}) \in \sigma(t)$ with $d \in \{N,W,E,S\}$.

As a result, the product $(R(1,T,P) \bowtie R(2,T,P) \bowtie R(3,T,P)) \times_{\Sigma_{RF}} F_\mu(I)$ captures all sequences of observations for the three robots constrained by the field component. ■

Remark 3. *The floor part $\lfloor \cdot \rfloor$ acts as an approximation of the robot sensor on the field's position value. A different interaction signature may, for instance, introduce some errors in the reading.*

The interaction signature may also impose that $d(i,p)$ relates to the speed $(0, 1/T)$, $(0, -1/T)$, $(-1/T, 0)$, and $(1/T, 0)$ when $d = N$, $d = S$, $d = W$, and $d = E$, respectively. Then, for a time interval T, the power p moves the robot by one unit on the field.

Remark 4. *In practice, it is unlikely that two observations happen at exactly at the same time. However, in our framework, the time of an observation is an abstraction that requires every event of the observation to occur after the events of the previous observation, and before the events of the next observation.*

Example 25 (Battery)**.** *A battery component with capacity C in mAH is a pair $B = (E_B(C), L_B(C))$ with events $read(l) \in E_B(C)$ for $0\% \leq l \leq 100\%$, $charge(\mu) \in E_B(C)$, and $discharge(\mu) \in E_B(C)$ with μ a (dis)charging coefficient in % per seconds. The battery displays its capacity with the event capacity(C). The behavior L_B is a set of sequences $\sigma \in L_B$ such that there exists a piecewise linear function $f : \mathbb{R}_+ \to \mathcal{P}(E_B)$ with, for $\sigma(i) = (O_i, t_i)$,*

- *for $\sigma(0) = (O_0, t_0)$, $f([0; t_0]) = 100\%$, i.e., the battery is initially fully charged;*
- *if $O_i = \{read(l)\}$, then $f(t_i) = l$ and the derivation $f'_{[t_{i-1}, t_{i+1}]}$ of f is constant in $[t_{i-1}, t_{i+1}]$, i.e., the observation does not change the slope of f at time t_i;*
- *if $O_i = \{discharge(\mu)\}$, then $f_{[t_i, t_{i+1}]}(t) = \max(f(t_i) - (t - t_i)\mu, 0)$;*
- *if $O_i = \{charge(\mu)\}$, then $f_{[t_i, t_{i+1}]}(t) = \min(f(t_i) + (t - t_i)\mu, 100)$;*

where $f_{[t_1;t_2]}$ is the restriction of function f on the interval $[t_1; t_2]$. There is a priori *no restrictions on the time interval between two observations, as long as the sequence of timestamps is increasing and non-Zeno. Finally, we use B_i for a battery whose events are identified by the natural number i. Then, for $B = (E_B(C), L_B(C))$, we have $B_i = (E_{B_i}(C), L_{B_i}(C))$ with $e \in E_B(C)$ if and only if the corresponding identified event $e_i \in E_{B_i}(C)$, e.g. $read_i(l) \in E_{B_i}(C)$ for all $read(l) \in E_B(C)$. The set of TESs $L_{B_i}(C)$ is obtained by replacing in every TESs in $L_B(C)$ the corresponding identified event in $E_{B_i}(C)$.* □

Example 26. *We define $\Sigma_{RB} = ([\kappa_{RB}], \cup)$ where $\cup$ unions two TESs as defined in the preliminaries, and $[\kappa_{RB}]$ specifies co-inductively (see [57] for details of the construction), from a relation on observations κ_{RB}, how event occurrences relate in the robot and the battery components of capacity C. More specifically, κ_{RB} is the smallest symmetric relation over observations such that $((O_1, t_1), (O_2, t_2)) \in \kappa_{RB}$ implies that $t_1 = t_2$ and*

- *the discharge event in the battery coincides with a move of the robot, i.e., $d(i, p) \in O_1$ if and only if $discharge(\mu) \in O_2$. Moreover, the interaction signature imposes a relation between the discharge coefficient μ and the required power p, i.e., $\mu = p/C$;*

- *the read value of the robot sensor coincides with a value from the battery component, i.e., read*$(i,l) \in O_1$ *if and only if read*$(l) \in O_2$*;*
- *the robot reads the capacity value that corresponds to the battery capacity, i.e., getCapacity(i,c)* $\in O_1$ *if and only if capacity(c)* $\in O_2$*.*

The product $B \times_{\Sigma_{RB}} R(i,P,T)$ *of a robot and a battery component, under the interaction signature* Σ_{RB}*, restricts the behavior of the battery to match the periodic behavior of the robot, and restricts the behavior of the robot to match the sensor values delivered by the battery.*
As a result, the behavior of the product component $B \times_{\Sigma_{RB}} R(i,P,T)$ *contains all observations that the robot performs in interaction with its battery. Note that trace properties, such as* all energy sensor values observed by the robot are within a safety interval*, does not necessarily entail safety of the system: some unobserved energy values may fall outside of the safety interval. Moreover, the frequency by which the robot samples may* reveal *some new observations, and such robot can safely sample at period* T *if, for any period* $T' \leq T$*, the product* $B \times_{\Sigma_{RB}} R(i,P,T')$ *satisfies the safety property.*
In case of a battery B_j *with identifier* j *and a robot* $R(i)$ *with identifier* i*, we use* $\Sigma_{R_iB_j}$ *for the interactio signature that synchronizes, as described above, occurrences of events of the battery* B_j *with occurrences of events of the robot* $R(i)$*.* □

Behavioral properties of components

Consider the system

$$S(n,T_1,...,T_n) = \bowtie_{i\in\{1,...,n\}} (R(i,T_i) \times_{\Sigma_{R_iB_i}} B_i) \times_{\Sigma_{RF}} F(\{1,...,n\}) \tag{2.1}$$

made of n robots $R(i,T_i)$, each interacting with a private battery B_i under the interaction signatures $\Sigma_{R_iB_i}$, and in product with a field F under the interaction signature Σ_{RF}. We use $\bowtie$ for the product with the free interaction signature (i.e., every pair of TESs is composable), and the notation $\bowtie_{i\in\{1,...,n\}} \{C_i\}$ for $C_1 \bowtie ... \bowtie C_n$ as $\bowtie$ is commutative and associative.

We fix $n = 2$ in Equation (2.1) and the same period T for the two robots. We write E for the set of events of the composite system $S(2,T)$. We formulate the scenarios described in Section 1.3 in terms of a satisfaction problem involving a safety property on TESs and a behavioral property on the composite system. We first consider two

safety properties:

$$P_{energy} = \{\sigma \in TES(E) \mid \forall i \in \mathbb{N}.\{read_1(0), read_2(0)\} \not\subseteq \mathsf{pr}_1(\sigma)(i)\}$$

which models that the two batteries don't display simultaneously that their level is empty; and $P_{no-overlap}$ which is the set

$$\{\sigma \in TES(E) \mid \forall i \in \mathbb{N}.\forall (x,y) \in [0,20]^2, \{(x,y)_1, (x,y)_2\} \not\subseteq \mathsf{pr}_1(\sigma)(i)\}$$

that captures all behaviors where the two robots are never observed together at the same location.

Observe that, while both P_{energy} and $P_{no-overlap}$ specify some safety properties, they are not sufficient to ensure the safety of the system. We illustrate some scenarios with the property P_{energy}. If a component never reads its battery level, then the property P_{energy} is trivially satisfied, although effectively the battery may run out of energy. Also, if a component reads its battery level periodically, each of its readings may return an observation agreeing with the property. However, in between two read events, the battery may run out of energy (and somehow recharge). To circumvent those unsafe scenarios, we add an additional behavioral property.

Let $X_{read} = \{read(l_1)_1, read(l_2) \mid 0 \leq l_1 \leq C_1,\ 0 \leq l_2 \leq C_2\}$ be the set of reading events for battery components B_1 and B_2, with capacities C_1 and C_2 respectively. The property $\phi_{insert}(X_{read}, E)$, as detailed in Example 20, defines a class of component behaviors that are closed under insertion of *read* events for the battery component. Therefore, the system $S(2,T)$ is energy safe if $S(2,T) \models P_{energy}$ and its behavior is closed under insertion of battery read events, i.e., $S(2,T) \Vdash \phi_{insert}(X_{read}, E)$. In that case, every TES of the component's behavior is part of a set that is closed under insertion, which means all read events that the robot may do in between two events observe a battery level greater than 0Wh. The behavior property enforces the following safety principle: had there been a violating behavior (i.e., a run where the battery has no energy), then an underlying TES would have observed it, and hence the behavioral property would have been violated.

Another scenario for the two robots is to consider their coordination in order to have them exchange their positions. Let F be initialized to have robot R_1 at position $(0,0)$ and robot R_2 at position $(5,0)$. The property of exchanging position is a liveness

property defined as:

$$\mathrm{P}_{\mathrm{exch}} = \{\sigma \in \mathit{TES}(E) \mid \{(0,0)_1, (5,0)_2\} \subseteq \mathsf{pr}_1(\sigma)(0) \ \mathit{and} \\ \exists i \in \mathbb{N}. \{(5,0)_1, (0,0)_2\} \subseteq \mathsf{pr}_1(\sigma)(i)\}$$

where $(x, y)_i$ is the position of robot i on the field. It is sufficient for a liveness property to be satisfied for the system to be live, i.e., in the case of $\mathrm{P}_{\mathrm{exch}}$ being satisfied, the two robots eventually exchanged position. However, it may be that the two robots exchange their positions before the actual observation happens. In that case, using a similar behavioral property as for safety property will make sure that if there exists a behavior where robots exchange their positions, then such behavior is observed as soon as it happens.

2.2 Division and conformance

Composition is an important feature of a specification language, as it enables the design of a complex system in terms of a product of its parts. Decomposition is equally important in order to reason about structural properties. Usually, however, a system can be decomposed in more than one way, each optimizing for different criteria. We extend an algebraic component-based model for cyber-physical systems to reason about decomposition. Components compose using a family of algebraic products, and decompose, under some conditions, given a corresponding family of division operators. We use division to specify invariants of a system of components, and to model desirable updates. We apply our framework to design a cyber-physical system consisting of robots moving on a shared field, and identify desirable updates using our division operator.

It is common, when modeling a system, to separate its design from its implementation. A design framework employs high level primitives to simplify the specification of *what* behavior is desirable. A formal design, moreover, enables high level analysis that benefit later implementation (e.g., proof of the existence of an implementation, verification of safety properties, etc.). Alternatively, an implementation uses low level operations to specify *how* a behavior is constructed. An implementation provides a precise description of the system's behavior that can be tested, simulated, and, in some conditions, proved correct with respect to the design specification. We present a design framework for cyber-physical systems that enables reasoning about composition and decomposition.

Composition is the act of assembling components to form complex systems. This property is particularly desirable if the underlying parts have different type of specifications (e.g., continuous or discrete), but still need to communicate and interact. Our work in [57] presents a component model that captures both discrete and continuous changes, for which timed-event streams (TESs) are instances of a component behavior. An observation is a set of events with a unique time stamp. A component has an interface that defines which events are observable, and a behavior that denotes all possible sequences of its observations (i.e., a set of TESs). The precise machinery that generates such component is abstracted away. Instead, we present interaction between components as an algebra on components, and we give a wide variety of user defined operations.

Decomposition is dual to composition, as it simplifies a component behavior by removing some of its parts. Decomposition is interesting in two ways: it gives insight on whether a system is composite of a specific component, and it returns a subsystem that, in composition with that component, would give back the initial system. Decomposition is not unique, and may induce a cost or a measure, i.e., a component A may be seen as a product $B \times C$ or $B \times D$ with $C \neq D$. While the qualitative behavior may not change, i.e., the set of sequences of observations stays the same, the substitution of a component with another may somehow *improve* the overall system, e.g., by enhancing its efficiency. For instance, running time is often omitted when specifying systems whose behavior is oblivious to time itself. However, in practice, the time that a program takes to process its inputs matters. Thus, a component may be substituted with a component exposing the same behavior but running faster. Other criteria such as the size of the implementation, the cost of the production, procurement, maintenance, etc., may be considered in changing one component for another. In this paper, we also consider an orthogonal concern: the cost of *coordination*. Intuitively, the cost of coordination captures the fact that events of two components are tightly related. For example, if two events are related, the occurrence of an event in one component implies the occurrence of some events in another component. While such constraints are declarative in our model, their implementation may be costly. Thus, the relation between observable events of two components may increase the underlying cost to concurrently execute those two components. Finally, having operation to study system decomposition brings alternative perspective on fault detections and diagnosis [45].

Formally, we extend our algebra of components [57] with a new type of operator: a division operation. Division intuitively models decomposition, and acts as an inverse composition operation. Practically, the division of component A by component B

returns one component C from all the components D such that $A = B \times D$. Different cost models give rise to different operations of division. We abstractly reason about cost using a partially ordered set of components and show that, for some orders, the set of candidates naturally gives rise to a maximal (minimal) element.

As a running example, we consider a set of robots moving continuously on a shared field. We use the operation of division to specify desirable updates that would prevent robots from interfering with other robots. We also apply division to find simpler components that, if used, would still preserve the entire system behavior. We finally specify the necessary coordination for the robots to self sort on the field.

2.2.1 Divisibility and quotients

Consider two components B and C, and a product $\times$ over components, modelling the interaction constraints on B and C. Then, the composite expression $A = C \times B$ captures, as a component, the concurrent observation of components C and B under the interaction modelled by $\times$. Consider a component D such that $C \times B = D \times B$. If D is different from C, then the equality states that the result of D interacting with B is the same as C interacting with B. Consequently, in this context, component C can be replaced by component D while preserving the global behavior of A.

In general, a component D that can substitute for C is not unique. The set of alternatives for D depends, moreover, on the product $\times$, on the component B, and on the behavior of A. A 'goodness' measure may induce an order on this set of components, and eventually give rise to a *best* substitution. More generally, the problem is to characterize, given two components A and B and an interaction product $\times$, the set of all C such that $A = C \times B$.

The divisibility of a component A by a component B under product $\times$ captures the possibility to write A as a product of B with another component.

Definition 17 (Right (left) divisibility)**.** *A component A is right (respectively, left) divisible by B under the product $\times$ if there exists a component C such that $B \times C = A$ (respectively, $C \times B = A$).*

A is *divisible* by B under $\times$ when A is both left and right divisible by B under $\times$. Intuitively, the set of witnesses for divisibility, contains all the components that, if taken in a product (under the same interaction signature) with the divisor, yield the dividend. Such witnesses are called *quotients*.

Definition 18 (Right (left) quotients)**.** *The right (respectively, left) quotients of A*

by B under the product $\times_\Sigma$, written $A/^*_\Sigma B$ (respectively, $A \backslash^*_\Sigma B$), is the set $\{C \mid B \times_\Sigma C = A\}$ (respectively, $\{C \mid C \times_\Sigma B = A\}$).

If $\times_\Sigma$ is commutative, then $A/^*_\Sigma B = A\backslash^*_\Sigma B$, in which case we write ${}_\Sigma\frac{A}{B}*$. We define left (right) division operators that pick, given a choice function[9], the best element from their respective sets of quotients as their quotients.

Example 27. *Consider a robot that performs 5 moves, and then stops. Each move consumes some energy, and the robot therefore requires sufficient amount of energy to achieve its moves. The product of a robot C with its battery B under the interaction signature Σ is given by the expression $A = C \times_\Sigma B$, where Σ synchronizes a move of robot C with battery B. Note that different batteries behave differently. The set of batteries that would lead to the same behavior is given by the quotients of A by B.*

Definition 19 (Right (left) division)**.** *Let A be divisible by B under $\times_\Sigma$. The right (respectively, left) quotient of A divided by B, under the product $\times_\Sigma$ and the choice function χ over the right (respectively, left) quotients, is the element $\chi(A/^*_\Sigma B)$ (respectively, $\chi(A \backslash^*_\Sigma B)$). We write $A/^\chi_\Sigma B$ (respectively, $A \backslash^\chi_\Sigma B$) to represent the quotient.*

Example 28. *It is usual (e.g., [83]) to consider the greatest common divisor when forming the product of cyber-physical components, so that no observation is missed. Our operation of division, however, gives an alternative perspective. Let $C(H)$ be a component whose observations have time stamps multiple of $H \in \mathbb{R}_+$. Then, let $A = C(H_1) \times_\Sigma C(H_2)$. The set of components $\{C(H) \mid A = C(H_1) \times_\Sigma C(H), H \in \mathbb{R}_+\}$ contains all the quotients of A divisible by $C(H_1)$. The selection of the component with the lowest period H would be one choice function for the division of A by $C(H)$ under Σ.*

If $\times_\Sigma$ is commutative, then $A/^\chi_\Sigma B = A \backslash^\chi_\Sigma B$, in which case we denote the division as ${}_\Sigma\frac{A}{B}\chi$.

Example 29 (Lowest element)**.** *One measure to order the set of quotients is to find a component that* is contained *in all the other component behaviors. Indeed, every quotient has the property that, in composition with the divisor, the resulting component equals the dividend. Then, finding a quotient that is contained in all other quotients may be optimal in terms of behavior complexity.*

[9] We assume the axiom of choice [76] and the existence of a function χ that picks an element from a set.

Table 2.3: Counter example for a lowest element in the division of A by B, with C and D two quotients.

	$\sigma : A$	$\tau : B$	$\delta : C$	$\eta : D$
t_1	$\{0, 1, 2\}$	$\{0, 1\}$	$\{0, 2\}$	$\{1, 2\}$
t_2	$\{0, 1, 2\}$	$\{0, 1\}$	$\{0, 2\}$	$\{1, 2\}$
t_3	$\{0, 1, 2\}$	$\{0, 1\}$	$\{0, 2\}$	$\{1, 2\}$
...	...	...	...	

Let $\mathcal{C}$, the set of right (left) quotients for A divisible by B for product $\times$, is equipped with an ordering such that the lowest element is an element of $\mathcal{C}$, then a function that picks the lowest element can act as a choice function to define the result of the division of A by B. ■

One may consider $\leq$ as a natural ordering on quotients. However, the set of quotients equipped with the containment relation may not have a lowest element. One such example is shown in Table 2.3. Consider A, B, C, and D with $\{0, 1, 2\}$, $\{0, 1\}$, $\{0, 2\}$, and $\{1, 2\}$ as interface, respectively. Using the synchronous composition operation, the TESs τ and η compose with the TES δ to give the TES σ. However, C and D require synchronization on their shared event to compose with B. A smaller component than C and D would be a component F, whose interface is the singleton set containing event 2. However, such component has no shared event with B, and may therefore freely interleave its observations, which does not correspond with observations in A. Thus, F is not an element of the quotients, and C and D have no lower bound in the set of quotients.

We show in the next theorem that a subset of quotients with a shared interface has a lower bound. We discuss how the choice of an interface for a quotient may be guided by some qualitative design choices.

Theorem 2. *Let $\leq$ be the containment relation introduced in Definition 3. Let $\times_\Sigma$ be a commutative, associative, and idempotent product on components, and such that for any two components C and D with the same interface, $C \times_\Sigma D \leq C$. Given A divisible by B under $\times_\Sigma$, any finite subset of quotients sharing the same interface E has a lower bound that is itself a quotient in $A/^*_\Sigma B$.*

Proof. Let $\mathcal{C}(E)$ be a finite subset of the set $\{C \mid C \text{ has interface } E \text{ and } C \in A/^*_\Sigma B\}$. We also write $\times_\Sigma \mathcal{C}(E)$ for the product of all components in $\mathcal{C}(E)$.

For any $C \in \mathcal{C}(E)$, we have

$$\times_\Sigma \mathcal{C}(E) \leq C$$

which makes $\times_\Sigma \mathcal{C}(E)$ a lower bound for $\mathcal{C}(E)$.

Given associativity, commutativity, and idempotency of $\times_\Sigma$, for any $C_1, C_2 \in \mathcal{C}(E)$:

$$\begin{aligned} A &= B \times_\Sigma C_1 \\ A &= B \times_\Sigma C_2 \\ A \times_\Sigma A = A &= (B \times_\Sigma C_1) \times_\Sigma (B \times_\Sigma C_2) \\ A &= B \times_\Sigma (C_1 \times_\Sigma C_2) \end{aligned}$$

which, applied over the set $\mathcal{C}(E)$, gives $A = B \times_\Sigma (\times_\Sigma \mathcal{C}(E))$. Thus, $\times_\Sigma \mathcal{C}(E) \in \mathcal{C}(E)$.[10] □ □

When conditions of Theorem 2 are satisfied, we write $A/\overset{\leq,E}{\Sigma} B$ for the lower bound of the set of quotients with interface E.

Remark 5. *The operation of division defined by Theorem 2 raises several points for discussion. First, the set of quotients sharing the same interface is structured. Indeed, when the interface is fixed, each finite subset of quotients has a lowest element under $\leq$, which makes the definition of a division operator possible. Second, the fact that there is, in general, no minimal element over the set of all quotients reveals the important role that interfaces play in system decomposition. In other words, one may consider another measure to* choose *a quotient interface, that is orthogonal to behavior containment (see Section 2.2.4 for a discussion about the cost of coordination).*

We use **1** to denote the component $(\emptyset, TES(\emptyset))$, and **0** to denote the component $(\emptyset, \emptyset)$, that has the empty interface and no behavior.

A component $A = (E_A, L_A)$ is *closed under insertion of silent observations* if, for any $\sigma \in L_A$, and for any silent observation $(\emptyset, t)$ with $t \in \mathbb{R}_+$, and given $i \in \mathbb{N}$ such that $\sigma(i) = (O, t_1)$ and $\sigma(i+1) = (O', t_2)$ with $t_1 < t < t_2$, then there exists $\tau \in L_A$ such that $\sigma(k) = \tau(k)$ for all $k \leq i$, $\sigma(i+1) = (\emptyset, t)$, and $\sigma(k+2) = \tau(k+1)$ for all $k > i$.

In order to reason about components algebraically, we want some properties to hold. For instance, that a component is divisible by itself and the set of quotients contains the unit element.

Lemma 14. *Let A be a component closed under insertion of silent observations, and*

[10] Strictly speaking, closure under finite product does not necessarily imply closure under infinite product. We leave investigating the conditions under which closure under infinite product holds, for future work.

Σ_{sync} *the synchronous interaction signature introduced in Example 4. Then,* $\mathbf{1} \in A/^{*}_{\Sigma_{sync}} A$.

Proof. For any element $\sigma : A$, and for any $\tau : \mathbf{1}$, we have $(\sigma, \tau) \in R$ and $\sigma[\cup]\tau : A$. Moreover, for any $\sigma : A$, there exists $\tau : \mathbf{1}$ such that $(\sigma, \tau) \in R$ and $\sigma[\cup]\tau = \sigma$. Then, $\mathbf{1}$ is in the set of quotients of A by A. □ □

Remark 6. *Note that Lemma 14 assumes components to be closed under insertion of silent observations. The reason, as shown in the proof, comes from the product of* $\mathbf{1}$ *with a component A that may insert silent observations at arbitrary points in time. A consequence of Lemma 14 is the existence of a choice function that can pick, from the set of quotients, the unit component for the division of A by A.*

Example 30. *Let* $(R(1,P,T) \bowtie R(2,P,T) \bowtie R(3,P,T)) \times_{\Sigma_{RF}} F_{\mu}(I)$ *be the product of three robot components and a field component with* $I = \{1,2,3\}$. *Consider the component* $P = (E, L)$ *with* $E = \{read((n,m),i), (n,m)_i \mid n, m \in \mathbb{N}\}$ *and* $L \subseteq TES(E)$.

Then, $((R(1,P,T) \bowtie R(2,P,T) \bowtie R(3,P,T)) \times_{\Sigma_{RF}} F_{\mu}(I))/^{\leq, E'}_{\Sigma_{RF}} P$, *with* $E' = (E_R(1) \cup E_R(2) \cup E_R(3) \cup E_F(I)) \setminus \{(n,m)_i \mid n, m \in \mathbb{N}\}$, *denotes the component that, in composition with P, recovers the initial system. Note that the component resulting from division ranges over the interface* E'. *As a consequence, all events* $(n,m)_i$ *have been hidden in the quotient. Note that the division exists due to the interaction signature* Σ_{RF} *that imposes simultaneity on occurrence of events* $read((n,m),i)$ *and* $(n,m)_i$. ■

Lemma 15. *Let* $\times_{\Sigma}$ *be commutative. Given A divisible by B under* Σ *and* χ *a choice function on the set of quotients of A divisible by B, then* $B \in A/^{*}_{\Sigma}(A/^{\chi}_{\Sigma} B)$.

Proof. If A is divisible by B under Σ and if χ selects one quotient over the set, then $C = A/^{\chi}_{\Sigma} B$ is such that $A = B \times_{\Sigma} C$. By commutativity of $\times_{\Sigma}$, $A = C \times_{\Sigma} B$ and $B \in A/^{*}_{\Sigma} C$. □

Lemma 16. *Let* $\times_{\Sigma}$ *be associative. If A is divisible by B under* Σ *and B is divisible by C under* Σ, *then A is divisible by C under* Σ.

Proof. If A is divisible by B under Σ, then there exists D such that $A = B \times_{\Sigma} D$. If B is divisible by C under Σ, then there exists E such that $B = C \times_{\Sigma} E$. By substitution, we have $A = (C \times_{\Sigma} E) \times_{\Sigma} D$. Using associativity of $\times_{\Sigma}$, we get $A = C \times_{\Sigma} (E \times_{\Sigma} D)$ which proves that A is divisible by C under $\times_{\Sigma}$. □

2.2.2 Conformance

The criterion for divisibility of A by B, under product $\times$, is the existence of a quotient C such that $B \times C = A$. The equality between $B \times C$ and A makes division a suitable decomposition operator. We can define, a similar operation to describe all components C that *coordinate* B in order for the result to behave in conformance with specification A. In this case, we replace equality with the refinement relation of Definition 2.

Definition 20 (Right (left) conformance)**.** *Component B is right (respectively, left) conformable with component A under $\times$ if there exists a non-empty component C such that $C \times B \sqsubseteq A$ (respectively, $B \times C \sqsubseteq A$).*

Definition 21 (Right (left) conformance coordinators)**.** *The right (respectively, left) conformance coordinators that make B behave in conformance with A under $\times_\Sigma$, denoted as $A \lfloor^*_\Sigma B$ (respectively, $A \lfloor^*_\Sigma B$), is the set $\{C \mid C \times_\Sigma B \sqsubseteq A\}$ (respectively, $\{C \mid B \times_\Sigma C \sqsubseteq A\}$).*

If $\times_\Sigma$ is commutative, then $A \lfloor^*_\Sigma B = A \lfloor^*_\Sigma B$, in which case we write $A \downarrow^*_\Sigma B$. Trivially, every component can be coordinated with the empty coordinator, i.e., the component $\mathbf{0} = (\emptyset, \emptyset)$. However, the set of coordinators having the same interface is structured and gives ways to define non-trivial coordinators, as in Theorem 3.

Definition 22 (Right (left) coordinator)**.** *Let B be conformable with component A, and let χ be a choice function that selects the best component out of a set of components. The right (respectively, left) coordinator that makes B behave in conformance with A, denoted as $A \lfloor^\chi_\sigma B$ (respectively, $A \lfloor^\chi_\Sigma B$), is the component $\chi(A \lfloor^*_\Sigma B)$ (respectively, $\chi(A \lfloor^*_\Sigma B)$).*

Example 31 (Greatest element)**.** *One measure to order the set of coordinators is containment. The refinement relation used to define conformance also accepts coordinators that have no behavior at all, and trivially satisfies the behavior inclusion relation. To maximize the observables of the resulting composite behavior set, corresponds to finding the greatest coordinator under the containment relation.*

More generally, if $\mathcal{C}$, the set of right (left) coordinators for B conformable with A under $\times$, is equipped with an ordering such that the greatest element is an element of $\mathcal{C}$, then the function that picks the greatest element can act as a choice function to select the best conformance coordinator of B to behave as A under $\times$. ■

Following the result of Theorem 2, if the interface of the quotient is fixed, then the subset of quotients that have the same interface has a least element with the

containment relation introduced in Definition 3. We show in Theorem 3 that a similar result holds for the set of coordinators.

Theorem 3. *Let $\leq$ be the containment relation introduced in Definition 3. Let $\times_{(R,\oplus)}$ be a commutative, associative, idempotent, and monotonic (as in Definition 9) product on components. Given B conformable with A under $\times_{(R,\oplus)}$, any finite subset of coordinators sharing the same interface E has an upper bound that is itself a coordinator in $A \downarrow^*_\Sigma B$.*

Proof. Let $\mathcal{C}(E)$ be a finite subset of the set $\{C \mid C \text{ has interface } E \text{ and } C \in A \downarrow^*_{(R,\oplus)} B\}$. We define the union of two components $A = (E_A, L_A)$ and $B = (E_B, L_B)$, as the component $A \cup B = (E_A \cup E_B, L_A \cup L_B)$. The union of all components in $\mathcal{C}(E)$ is the component $\bigcup \mathcal{C}(E) = (E, \bigcup_{C \in \mathcal{C}(E)} L_C)$ where L_C is the behavior of component C. Moreover, we have that, for any component A, B, C, with B and C sharing the same interface E, $(A \times_{(R,\oplus)} B) \cup (A \times_{(R,\oplus)} C) = A \times_{(R,\oplus)} (B \cup C)$. Indeed let L be the behavior of $(A \times_{(R,\oplus)} B) \cup (A \times_{(R,\oplus)} C)$ and S be the behavior of $A \times_{(R,\oplus)} (B \cup C)$:

$$\begin{aligned} L = &\{\sigma \oplus \tau \mid \sigma \in L_A, \tau \in L_B, (\sigma, \tau) \in R(E_A, E)\} \cup \\ &\{\sigma \oplus \tau \mid \sigma \in L_A, \tau \in L_C, (\sigma, \tau) \in R(E_A, E)\} \\ = &\{\sigma \oplus \tau \mid \sigma \in L_A, \tau \in L_B \cup L_C, (\sigma, \tau) \in R(E_A, E)\} = S \end{aligned}$$

We show that $\bigcup \mathcal{C}(E)$ is an upper bound for the set of coordinators $\mathcal{C}(E)$. For any $C \in \mathcal{C}(E)$, we have

$$C \sqsubseteq \bigcup \mathcal{C}(E)$$

which implies that $C \leq \bigcup \mathcal{C}(E)$ and makes $\bigcup \mathcal{C}(E)$ an upper bound for $\mathcal{C}(E)$.

Given associativity, commutativity, and idempotency of $\times_{(R,\oplus)}$, for any $C_1, C_2 \in \mathcal{C}(E)$:

$$\begin{aligned} B \times_{(R,\oplus)} C_1 &\sqsubseteq A \\ B \times_{(R,\oplus)} C_2 &\sqsubseteq A \\ (B \times_{(R,\oplus)} C_1) \cup (B \times_{(R,\oplus)} C_2) &\sqsubseteq A \\ B \times_{(R,\oplus)} (C_1 \cup C_2) &\sqsubseteq A \end{aligned}$$

which, applied over the set $\mathcal{C}(E)$, gives $B \times_{(R,\oplus)} (\bigcup \mathcal{C}(E)) \sqsubseteq A$. Thus, $\bigcup \mathcal{C}(E) \in \mathcal{C}(E)$. □ □

Finding a conformance coordinator that makes B behave in conformance with A

is looser than finding a quotient for A divisible by B: any quotient of A by B under a product $\times_\Sigma$ is therefore a coordinator that makes B conformable with A. Such quotient-coordinator has the property that it "coordinates" B such that the resulting behavior covers the whole behavior of A.

For some suitable products, Theorem 2 and Theorem 3 state the existence, respectively, of a lowest element in the subsets of quotients and a largest element in the set of coordinators that share the same interface. The synchronous product introduced in Example 4 is one product that satisfies the requirements of each theorem.

2.2.3 Applications of Division

In this section, we consider the robot, field, and protocol components introduced in Examples 21, 22, and 23, together with the synchronous product $\bowtie$ of Example 4 and the product $\times_{\Sigma_{RF}}$ of Example 24. Both products are commutative (Lemma 1 in [57]), and we therefore omit the right and left qualifiers for division and conformance.

Initial conditions For each robot, we fix the power requirement of a move and the time period T between two observations to be such that a move of a robot during a period T corresponds to a one unit displacement on the field. Then, each move action of a robot changes the location of the robot by a fixed number of units or none if there is an obstacle. We write $R(i)$ for robot $R(i, P, T)$ with such fixed P and T. As an example, the observation $(\{d(i), \mathit{read}(i, (x; y))\}, t)$ followed by the observation $(\{\mathit{read}(i, (x'; y'))\}, t+T)$ gives only few possibilities for $(x'; y')$: either $(x; y) = (x'; y')$, in which case the robot got blocked in the middle of its move, or $(x'; y')$ increases (or decreases) by one unit the x or y coordinates, according to the direction d.

Let the initial state μ of the field be such that $\mu(1) = (3; 0)$, $\mu(2) = (2; 0)$, and $\mu(3) = (1; 0)$, which defines the initial positions of $R(1)$, $R(2)$ and $R(3)$ respectively, and let there be obstacles throughout the field on the 3×2 rectangle from $(0; -1)$ to $(4; 2)$, i.e., for all $(x, y) \in (\llbracket 0; 4 \rrbracket \times \llbracket -1; 2 \rrbracket) \setminus (\llbracket 1; 3 \rrbracket \times \llbracket 0, 1 \rrbracket)$, there exists $i \in I$ such that $\mu(i) = (x, y)_i$. As a result, the moves of each robot are restricted to the inside of the 3×2 rectangle as displayed in Table 2.5.

Approximation of the Field as a Grid

Problem A field component captures in its behavior the continuous responses of a physical field interacting with robots roaming on its surface. The interface of the field contains therefore an event, per object, for each possible position and each pos-

sible move. In some cases, however, only a subset of those events are of interest. For instance, we may want to consider only integer position of objects on the grid, and discard intermediate observables. As a result, such component would describe a discrete grid instead of a continuous field, while preserving the internal physics: no two objects are located on the same position. We show how to define the grid as a subcomponent of the field, using the division operator.

Definition of the grid We use division to capture a discrete grid component $G_\mu(I) \leq F_\mu(I)$ contained in the field component $F_\mu(I)$. A grid component has the interface $E_G(I)$, where $E_G(I) \subseteq E_F(I)$ with $(x, y)_i \in E_G(I)$ implies $x, y \in \mathbb{N}$.

We use the component $C = (E_G(I), TES(E_G(I)))$ to denote the free component whose behavior contains all TESs ranging over the interface $E_G(I)$. Then, by application of Theorem 2, we use the least element with respect to $\leq$ of the set of quotients of $C \times_{\Sigma_{sync}} F_\mu(I)$ divided by $F_\mu(I)$ under Σ_{sync} to define the grid. Thus,

$$G_\mu(I) =_{\Sigma_{sync}} \frac{C \times_{\Sigma_{sync}} F_\mu(I)}{F_\mu(I)}(\leq, E_G(I)) \tag{2.2}$$

which naturally emerges as a subcomponent of the field component $F_\mu(I)$.

Consequences The grid component inherits some physical constraints from the field $F_\mu(I)$, but is strictly contained in the field component. There is a fundamental difference between an approximation of the position as a robot sensor detects, and a restriction of the field to integer positions as in the grid component. In the former, the component reads a value that does not corresponds precisely to its current position, while in the latter, the position read is exact but observable only for integer values.

As a result, the two component expressions $(R(1) \bowtie R(2) \bowtie R(3)) \times_{\Sigma_{RF}} G_\mu(I)$ and $(R(1) \bowtie R(2) \bowtie R(3)) \times_{\Sigma_{RF}} F_\mu(I)$ restrict each robot behavior in different ways: the grid component allows discrete moves only and the position that a robot reads is the same position as that of an object on the grid, while the field component allows continuous moves but the position that a robot reads is an approximation of the position of the robot on the field. In the sequel, we use the grid component $G_\mu(I)$ instead of the field component.

Updates of components

Problem The interaction signature of a product operator on components restricts which pairs of behaviors are composable. As a consequence, some components may have *more behavior* than necessary, namely the elements that do not occur in any composable pair. An update is an operation that preserves the global behavior of a composite system while changing an operand of a product in the algebraic expression that models the composed system. The goal of such update, for instance, is to remove some behaviors that are not composable or prevent some possible runtime errors. We give an example of such update that replaces a robot component by a new version that removes some of its possibly blocking moves.

Scenario For each robot, we fix its behavior to consist of TESs that alternate between move and reading observations. Moreover, for a robot's period T, and arbitrary $n_i \in \mathbb{N}$, we let $T \times n_i, i \in \mathbb{N}$, represent the timestamp of the i^{th} observation of a TES in its behavior, so long as $n_i < n_{i+1}$. Table 2.4 displays elements of the behavior for each robot. For instance, the TES $\eta : R(1)$ captures the observations resulting from $R(1)$ moving west twice. Note that, in composition with the grid component, the readings may conflict with the actual position of the robot, as some moves may not be allowed due to obstacles on the path.

For instance, given the expression $(R(1) \bowtie R(2) \bowtie R(3)) \times_{\Sigma_{RF}} G_\mu(I)$, the TES η is not observable as it is not composable with any of the TESs τ or δ from $R(2)$ and $R(3)$ respectively. We show how to use division to remove all of such behaviors.

Update The replacement for $R(1)$ should preserve the global behavior. We use division to define an update $R'(1)$ of $R(1)$ that removes all elements from its behavior that are not composable with any element from the behavior of $R(2)$ and $R(3)$ under the constraints imposed by the grid.

As a result, the component

$$R'(1) = \Sigma_{sync} \frac{(R(1) \bowtie R(2) \bowtie R(3)) \times_{\Sigma_{RF}} G_\mu(I)}{(R(1) \bowtie R(2) \bowtie R(3)) \times_{\Sigma_{RF}} G_\mu(I)} (\leq, E_R(1)) \tag{2.3}$$

contains in its behavior all elements ranging over the interface $E_R(1)$ that are composable with elements in the behavior of the dividend component. Note that the set of quotients is filtered on the interface $E_R(1)$, and $R(1)$ trivially qualifies as a quotient. However, $R(1)$ is not minimal as η can be removed from its behavior.

Table 2.4: Prefixes of four TESs for $R(1)$, $R(2)$, and $R(3)$. For direction d and robot i, we write $d(i)$ instead of $d(i,p)$ since the power p is initially fixed. We omit the set notation as observations are all singletons. We consider $(n_i)_{i\in\mathbb{N}}$ as an increase sequence of natural numbers.

t/T	$\sigma : R(1)$	$\eta : R(1)$	$\tau : R(2)$	$\delta : R(3)$
n_0	$read(1,(3;0))$	$read(1,(3;0))$	$read(2,(2;0))$	$read(3,(1;0))$
n_1	$N(1)$	$W(1)$	$N(2)$	$E(3)$
n_2	$read(1,(3;1))$	$read(1,(2;0))$	$read(2,(2;1))$	$read(3,(2;0))$
n_3	$W(1)$	$W(1)$	$W(2)$	$E(3)$
n_4	$read(1,(2;1))$	$read(1,(1;0))$	$read(2,(1;1))$	$read(3,(3;0))$
n_5	$W(1)$	$\emptyset$	$S(2)$	$\emptyset$
n_6	$read(1,(1;1))$	$\emptyset$	$read(2,(1;0))$	$\emptyset$
n_7	$S(1)$	$\emptyset$	$E(2)$	$\emptyset$
n_8	$read(1,(1;0))$	$\emptyset$	$read(2,(2;0))$	$\emptyset$
n_9	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
...	...	...	...	...

Consequence As a consequence, we defined, using division, an update for component $R(1)$ that removes some elements of its behavior while preserving the global behavior of the composite expression.

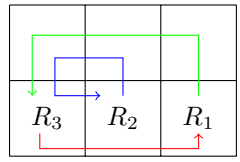
Note that the fact that in our example each robot alternates between a move and a read is crucial to remove, by composition, undesired behavior. Indeed, the readings of each robot must synchronize with the location displayed on the grid, and therefore implies that the robot successfully moved. The constraints imposed by the grid coordinate the robot by preventing two robots to share the same location.

Coordination and distribution

Problem Consider the scenario previously described with an additional modification: a robot no longer observes its location after every move, but only at the end of the sequence of moves. The TESs of each robot's behavior are described in Table 2.5, where (x_i, y_i) ranges over possible position readings for robot i. As a result, conflicts between robots may no longer be observable, and the timing of observations may render some incidents of robots blocking each other unobservable. We define a coordinator that makes the system conformant to a global property. As opposed to the division operation, a conformance coordinator may restrict the system behavior to a subset that conforms to a specified property. We consider the following property $P_{sorted}(I)$: "eventually, all the robots get sorted, i.e., every robot $R(i)$ eventually ends on the grid location $(i;0)$.".

Table 2.5: Prefixes of three TESs for $R(1)$, $R(2)$, and $R(3)$, graphically represented by some trajectories on a grid.

t/T	$\sigma : R(1)$	$\tau : R(2)$	$\delta : R(3)$
n_1	$N(1)$	$N(2)$	$E(3)$
n_2	$W(1)$	$W(2)$	$E(3)$
n_3	$W(1)$	$S(2)$	$read(3,(x_3;y_3))$
n_4	$S(1)$	$E(2)$	$\emptyset$
n_4	$read(1,(x_1;y_1))$	$read(2,(x_2;y_2))$	$\emptyset$
n_5	$\emptyset$	$\emptyset$	$\emptyset$
...	...	...	...



Global coordinator We can define, from the sort property, a component as $C_{sorted}(I) = (E_{sorted}(I), L_{sorted}(I))$ whose interface is the union of the interfaces of all robots and the grid, i.e., $E_{sorted}(I) = E_G(I) \cup \bigcup_{i \in I} E_R(i)$, and whose behavior $L_{sorted}(I) \subseteq TES(E_{sorted}(I))$ contains all sequences of moves that make the robots eventually end in their respective sorted grid positions, i.e., $\sigma \in L_{sorted}(I)$ if and only if there exists $t \in \mathbb{R}_+$ such that $(O,t) \in \sigma$ with $(i;0)_i \in O$ for all $i \in I$. Note that, by construction, the behavior of component $C_{sorted}(I)$ may contain some TESs from the behavior of component $(R(1) \bowtie R(2) \bowtie R(3)) \times_{\Sigma_{RF}} G_\mu(I)$, namely ever TESs that satisfies the property.

Consequently, the product of component $C_{sorted}(I)$ with $(R(1) \bowtie R(2) \bowtie R(3)) \times_{\Sigma_{RF}} G_\mu(I)$, under the signature Σ_{sync}, defines a component whose behavior contains all elements in the behavior of $(R(1) \bowtie R(2) \bowtie R(3)) \times_{\Sigma_{RF}} G_\mu(I)$ that are also in the behavior of $C_{sorted}(I)$. Therefore, if the behavior of component $((R(1) \bowtie R(2) \bowtie R(3)) \times_{\Sigma_{RF}} G_\mu(I)) \bowtie C_{sorted}(I)$ is not empty, $(R(1) \bowtie R(2) \bowtie R(3)) \times_{\Sigma_{RF}} G_\mu(I)$ is conformant to $C_{sorted}(I)$ and $C_{sorted}(I)$ is a principal coordinator. However, using $C_{sorted}(I)$ as a coordinator requires each component to synchronize, at each step, with every other component. We show how to define a different choice function on the set of coordinators, in order to identify a minimalist form of coordination.

Minimalist coordinator We define a coordinator whose interface is strictly contained in the interface of the global $C_{sorted}(I)$ coordinator. More precisely, we search for a coordinator over the interface of robot $R(1)$ that makes the system $(R(1) \bowtie$

$R(2) \bowtie R(3)) \times_{\Sigma_{RF}} G_\mu(I)$ conformant to the property component $C_{sorted}(I)$. First, observe that the set of coordinators

$$(R(1) \bowtie R(2) \bowtie R(3)) \times_{\Sigma_{RF}} G_\mu(I) \downarrow^*_{\Sigma_{sync}} C_{sorted}(I)$$

filtered on the interface $E_R(1)$ is empty. Indeed, for any set of timestamp factors n_1, n_2, n_3, and n_4 for the observables of $R(1)$ in Table 2.5, there exists an element from the behavior of $R(2)$ that delays its first action until after n_3, and eventually ends up in a blocking position. As a consequence, there is no coordinator restricted to the events of $R(1)$ that makes $(R(1) \bowtie R(2) \bowtie R(3)) \times_{\Sigma_{RF}} G_\mu(I)$ conformant to $C_{sorted}(I)$.

Instead, we consider filtering the set of coordinators with the interface $E_R(1) \cup \{N(2)\}$. In this case, one can find a simple coordinator that makes the set of robots to conform with the sort property. Indeed, every observation $N(1)$ of robot $R(1)$ must occur after an observable $N(2)$ of robot $R(2)$. As a result, such coordinator C_{12} restricts $R(1)$ to move only after $R(2)$ moves, which results in a composite system conformant to $C_{sorted}(I)$. Thus, the new coordinator C_{12}, in product with $R(1)$, $R(2)$, $R(3)$, and the grid $G_\mu(I)$ satisfies the sort property, i.e., $((R(1) \bowtie R(2) \bowtie R(3)) \times_{\Sigma_{RF}} G_\mu(I)) \bowtie C_{12} \sqsubseteq C_{sorted}(I)$.

Consequence We showed one global coordinator for the set of robots to satisfy the sort property, and one minimalist coordinator over the interfaces of $E_R(1)$ with an event from $E_R(2)$. The minimalist coordinator has an interface strictly included in the global coordinator, which therefore minimizes the amount of interaction among components. In the next section, we discuss the cost of coordination as a possible measure to order a set of conformance coordinators.

2.2.4 Discussion

The operations of division and conformance defined earlier characterize all possible updates and coordinators for a composite system. In general, the set of quotients or coordinators is not a singleton, which then necessitates a choice function to *pick* a component that suits the needs. We saw in Theorem 2 how such a choice function can be defined using an ordering on components, and choosing the least of such component. Intuitively, such choice function prefers a quotient with the least amount of observations. However, Theorem 2 assumes a fixed interface, and does not discuss how to rank components according to their interface. We discuss alternative rankings hereafter in the case of a synchronous product $\bowtie$ of Example 4.

Cost of coordination

Let $A = (E_A, L_A)$ and $B = (E_B, L_B)$ be two components such that the set of quotients of A divisible by B under $\bowtie$, the synchronous product, is non empty. Consider, as well, a component $C = (E_C, L_C)$ in the set of quotients. We discuss alternative scenarios based on the interface of component C.

Consider the case where $E_C \cap E_B \neq \emptyset$. Then, events in the intersection $E_C \cap E_B$ are events for which C and B must perform a simultaneous observation, i.e., with equal time stamp. The size of the intersection $E_C \cap E_B$ therefore characterizes how much coordination should take place among two implementations of components B and C to successfully achieve the synchronous observations. Alternatively, if a component $D = (E_D, L_D)$ is in the set of quotients such that $|E_D \cap E_B| < |E_C \cap E_B|$, the number of shared events is smaller, which hints at a smaller amount of coordination among components D and B.

In the case that $E_C \cap E_B = \emptyset$ and $E_C \neq \emptyset$, the family of quotients with interface E_C are particularly useful. The fact that the two interfaces are disjoint tells us that A can be decomposed in two components that do not need any coordination. Indeed, as $\bowtie$ only constraints occurrences of shared events, if B and C share no events then they can be run completely independently.

The two cases highlighted above give us some insight on how coordination can be used as a measure to rank components. Note that such ranking is contextual to the dividend and the divisor. Even though C may require more coordination than D to synchronize with B in order to form component A, in the context of A divisible by another component F, the component C may become preferable to D.

The cost of coordination discussed here is orthogonal to efficiency measures discussed in 2. For that reason, the two measures can be combined to first rank components in terms of their interface to minimize the amount of coordination required, and then rank components sharing the same interface in terms of the size of their behavior. More thorough analysis could be done as to compare, from the behavior of each component, how often coordinated event effectively occurs.

Series of division

We defined the operator of division on components. We saw that, under some criteria, division may return a 'better' description of a composite system, i.e., for $A = B \times C$, the division of A by B may return a D better than C while preserving the behavior of A.

Then, one question that follows is that of convergence. Consider the expression $A = B \times C$, and the division of A by B returning a component $C' \neq C$. Symmetrically, the division of A by the new component C' may return a component $B' \neq B$. Repeating the same process, dividing A by B' and so on, gives a sequence of components $C^{(n)}$ and $B^{(n)}$ for the respective n^{th} division. Does the sequence eventually converge to a fixed pair of components?

2.3 Linearization

Programming languages like Reo [42] and LUSTRE [24] describe concurrent systems in terms of sequences of *transactions* (or interactions). A transaction is a finite set of actions that occur atomically, which corresponds to a timeless observation, as introduced in Section 2.1. That is, a transactions *occurs* if and only if all the actions in the set occur, without the interference of any other action in between. Atomicity captures the notion of the *all-or-nothing* behavior and the possibility of interleaving of independent events only (see Section 2.1). A prime example of a transaction in concurrent systems is a barrier synchronization, which allows a group of processes to proceed only when all of them reach a particular local state. We refer to an infinite sequence of transactions as a transactional trace, and refer to a set of transactional traces as a transactional behavior. Note that a transactional behavior is independent of the time at which the transaction occurs, and records the order of occuring transactions only. We use $\mathbb{T}$ to denote the set of transactional behaviors. Transactional behaviors are particularly suitable to describe *what* behavior is acceptable in a concurrent system, by declaring all sequences of transactions that are allowed. A transactional component denotes a transactional behavior restricted to a set of actions that we refer to as its interface. Transactional components are therefore a subclass of components as introduced in Section 2.1.

A transactional component is *linear* if and only if every transaction in this component is a singleton set or the empty set. Programming languages without syntax for such transactions, like [23, 73, 11], describe concurrent systems as linear (transactional) components. Linear components describe sequential behaviors and are particularly suitable to describe *how* the behavior of a component is generated, since at most one action happens at any given time. We use $\mathbb{L}$ to denote the set of transactional behaviors.

When two processes that have shared actions in their interfaces are composed, the occurrence of a shared action in their behaviors must coincide. This coincidence is

understood as synchronous communication between the two processes. The concept of (a)synchronous communication is orthogonal to transactional or linear behaviors. In this section, we deal exclusively with synchronous communication. To construct complex concurrent systems out of simpler components, we equip components with a composition operator that captures synchronous communication.

The design of a specification of a concurrent system is easier using transactional components [4], while its execution is better captured by linear components. The problem is therefore to characterize what linearization from transactional components in $\mathbb{T}$ to linear components in $\mathbb{L}$ are deemed correct. Intuitively, the linearization of a transactional component is correct if the behavior of the resulting components preserves the intended semantics of the behavior of the initial transactional component.

Moreover, in the case of concurrent and distributed systems, compositionality is an important feature that allows one to run software in parts, and assemble the compiled parts at runtime. Besides static composition, dynamic composition is required for, e.g., dynamic updates, modular compilation, or reconfiguration. Semantically, runtime composition is captured by the composition operation $\otimes$ in $\mathbb{L}$. After defining transactional components in Section 2.3.2, we characterize valid linearizations in Section 2.3.4 and give two practical instances, one that runs every component in lockstep, and one that allows for interleaving of independent transactions. More particularly, we require that a sequence after linearization contains all events of a transactional behavior and preserve some ordering among dependent events.

Notation In this section, we consider *order sensitive components*, which are components for which the order of observations matters but not the exact time labels of observations. For that reason, the behavior of such a component is closed under stretching time, as long as the order of observations remains the same. We relax the notation throughout this section, and use a sequence $\sigma \in \mathcal{P}_f(\Sigma)^\omega$ to effectively reason about all the TESs $\tau \in \mathit{TES}(\Sigma)$ that have the same order of observations, i.e., $\sigma = \mathsf{pr}_1(\tau)$.

2.3.1 Dependency and concurrency

We fix a (possibly infinite) set Σ of *primitive actions*, and write $\tau \notin \Sigma$ to denote an *internal action*. We write $\Sigma_\tau = \Sigma \cup \{\tau\}$ to denote the set of actions. We do not make any assumptions on the structure of primitive actions. For instance, Σ may consist of actions a_c of the meaning "the traffic light indicates color c", or Σ may consist of actions $a_{p,d}$ of the meaning "port p fires with data d". We use regular expressions to

denote a set of sequences of actions. Given a set X, with $a, b \in X$, the expressions $a+b$, ab, a^* respectively denote the sets of sequences $\{a, b\}$, $\{ab\}$, and $\{a, aa, aaa, ...\}$. Later on, we use regular expressions over a carrier that is the finite power set of X: we should not confuse the set notation from the terms of the carrier and the set notation from the set of sequences that the expression denotes.

Some actions are dependent. For instance, the statement "port p fires with data d always happens before port p fires with data e" induces a dependency relation between $a_{p,d}$ and $a_{p,e}$. We define dependency as follows.

Definition 23. *Dependency is a partial order $\leq$ on Σ_τ, such that τ is not related to any primitive action in Σ.*

In the sequel, we consider a fixed but arbitrary dependency $\leq$ and say that a *happens before* b if $a \leq b$. Dependency induces a symmetric, reflexive dependency relation

$$D_\leq = \{(a, b) \in \Sigma^2 \mid a \leq b \text{ or } b \leq a\},$$

which brings us to the realm of Mazurkiewicz traces [62]. Actions a and b are *dependent* if $(a, b) \in D_\leq$ and are *independent* if $(a, b) \notin D_\leq$. Note that, without loss of generality, we can make the partial order strict and remove the elements (a, a) from the dependency relation.

Remark 7. *Consider a client that can send requests a and b. Suppose that $(a, b) \notin D_\leq$. Then, a and b are independent (e.g., one request does not require the other request) and the client may perform the requests in parallel.* ◊

The dependency relation allows us to permute independent actions. In the case of finite sequences, it suffices to define a trace equivalence as the smallest equivalence relation that allows commutation of consecutive independent actions. However, such a definition would allow only finitely many commutations, which means that $(ab)^\omega$ and $(ba)^\omega$ are not trace equivalent, whenever a and b are independent. Following Gastin [36], we define trace equivalence by means of (infinite) dependency graphs.

Definition 24. *The dependency graph $\Gamma_\leq(s)$ of a sequence $s \in \Sigma_\tau^\omega$ of actions is a graph (V, E) with vertices V and edges E defined as*

$$V = \{(a, i) \mid a \in \Sigma_\tau, 0 \leq i < |s|_a\}$$
$$E = \{(a, i) \to (b, j) \mid (a, b) \in D_\leq \text{ and the } i\text{th } a \text{ occurs before the } j\text{th } b \text{ in } s\}$$

where $|s|_a \in \mathbb{N} \cup \{\infty\}$ is the number of occurrences of action $a \in \Sigma_\tau$ in s.

We define trace equivalence (modulo dependency $\leq$) as the kernel of $\Gamma_\leq$:

$$\equiv_\leq \;=\; \ker(\Gamma_\leq) \;=\; \{(u,v) \mid \Gamma_\leq(u) = \Gamma_\leq(v)\}.$$

In the sequel, we drop the subscript $\leq$, if there is no danger of confusion. For a sequence $s \in \Sigma_\tau^\omega$, we write $[s] = \{r \in \Sigma_\tau^\omega \mid r \equiv s\}$ for the equivalence class of s. For a set $L \subseteq \Sigma_\tau^\omega$, we write $[L] = \{r \in \Sigma_\tau^\omega \mid s \in L, r \equiv s\}$ for the closure of L under trace equivalence.

Remark 8. *The most primitive verification tool of a software engineer is the print statement. Let Σ denote the set of all possible results of every print statement occurring in a program. If the program is sequential, its observable behavior can be accurately expressed as a set $L \subseteq \Sigma^\infty$ of possibly infinite sequences of observations. The set L contains all relevant observable information of a sequential system.*

If the software is concurrent, distinct statements may print simultaneously. To prevent unreadable output, the print statements acquire exclusive access to the log file or console before printing. While such linearization of observations restores the readability of the output, it hides the fact that the program is concurrent. Hence, the set of sequences of observations does not *contain all relevant information of a concurrent system.* ◊

2.3.2 Transactional and linear components

We fix a collection $T \subseteq \mathcal{P}_f(\Sigma)$ of finite sets of primitive actions, called *transactions*. A transaction $A \in T$ is atomic, as defined in Section 2.1, such that all primitive actions in A occur simultaneously, and no other dependent action can interleave. The empty transaction $\emptyset$ is related to the internal τ step as we will see later. Note that some actions in Σ are conflicting, such as $a_{p,d}$ and $a_{p,e}$ from Section 2.3.1, for distinct data $d \neq e$. Since conflicting actions never occur in the same transaction, we generally have $T \neq \mathcal{P}_f(\Sigma)$ and $\emptyset \in T$.

A *transactional trace* is an infinite sequence $A_0 A_1 A_2 \cdots \in T^\omega$ of transactions. A *transactional behavior* is a set $K \subseteq T^\omega$ of transactional traces. The composition operator on transactional behaviors $K \in \mathcal{P}(T^\omega)$ is intersection, $\cap$, of sets. Let $I \subseteq \Sigma$ be a set of primitive actions. Consider the largest[11] symmetric relation $\sim_I$ on T^ω that

[11] This largest relation exists, because the union of all $\sim_I$ that satisfy Equation 2.4 satisfies Equation 2.4.

satisfies

$$\forall r, s \in T^\omega \qquad r \sim_I s \quad \Rightarrow \quad r(0) \cap I = s(0) \cap I \text{ and } r' \sim_I s', \tag{2.4}$$

where $r'(i) = r(i+1)$ and $s'(i) = s(i+1)$, for all $i \geq 0$. A transactional behavior $K \subseteq T^\omega$ is $\sim_I$*-closed*, whenever $r \sim_I s \in K$ implies $r \in K$, for all $r, s \in T^\omega$.

Given a transactional behavior K, a *transactional component* is a pair (I, K) with $I \subseteq \Sigma$ and such that K is $\sim_I$-closed. We call I its *interface*.

Example 32 (Sync channel)**.** *A Sync channel is a transactional component with* $I = \{a, b\}$ *as interface containing an input port* a *and an output port* b*. Whenever the Sync fires its input port, it simultaneously fires its output port.*

Formally, let $\Sigma = \{a, b\}$ *consist of two actions* a *and* b *that correspond with firing of the input and output ports of the Sync channel, respectively. The behavior of the Sync is defined as the set* $(\{a, b\} + \emptyset)^\omega$*.*

In a larger context, such as $\Sigma = \{a, b, c\}$*, for some other port* c*, behavior for the Sync component extends to* $(\{a, b, c\} + \{a, b\} + \{c\} + \emptyset)^\omega$*, because we assume that components are closed under addition/removal of actions outside of their interface. That is, the set* $(\{a, b, c\} + \{a, b\} + \{c\} + \emptyset)^\omega$ *is* $\sim_I$*-closed.* ◊

Remark 9. *There is a subtle difference between concurrent actions and transaction. Concurrency is about independence of actions such that they* may *happen simultaneously. A transaction is about timing of actions such that they* will *happen atomically, which may also be simultaneous. Since the two terms are both referring to actions that may happen at the same instance, it is easy to confuse the terms. A transaction makes sense only in a concurrent setting: actions are atomic and are therefore concurrent. However, concurrent actions are not necessarily atomic.* ◊

Example 33. *Let* C_1 *and* C_2 *be two Sync components, with interface* $\{a, b\}$ *and* $\{b, c\}$*, respectively. The behavior of the composition of* C_1 *and* C_2 *consists of all behaviors in* C_1 *that are also behaviors in* C_2 *(composition is intersection). By modelling atomicity of actions with transactions one gets transitivity of atomicity for free. Indeed, if* a *and* b *are atomic and* b *and* c *are atomic, then* a *and* b *occur within the same transaction and* b *and* c *occur within the same transaction. Hence,* a *and* c *eventually occur within the same transaction in the composition of* C_1 *and* C_2*, which means that they are atomic.* ◊

A *linear trace* is a special kind of transactional trace, where every element is either a singleton action, or the empty set. In order to simplify notation, we identify a

sequence of actions $a_0a_1a_2\cdots \in \Sigma_\tau^\omega$ with the linear trace $A_1A_2A_3\cdots \in \mathcal{P}(\Sigma)^\omega$, where each action a_i corresponds to the singleton transaction $\{a_i\}$ if $a_i \neq \tau$, and to the transaction $\emptyset$ otherwise. A *linear behavior* is a set $L \subseteq \Sigma_\tau^\omega$ of linear traces, and a *linear component* is a pair (I, K) of an interface I and a linear behavior K that is $\sim_I$ closed.

We consider an arbitrary associative, commutative, idempotent composition operator $\otimes$ on linear components. Typically, intersection of sets, $\cap$, is used as composition operator but see Section 2.3.4 for an example of a different composition operator.

Linearization Every transactional behavior has one or more equivalent linear behaviors that are related to it via linearization. The linearization is defined hierarchically, from transactions to traces and behaviors. Each level "lifts" the definition of linearization to sequences of transactions, and to sets of sequences of transactions.

The linearization of a transaction results in a set of sequences of actions. Our fixed dependency of actions, $\leq$, restricts the order in which the actions of a transaction can linearize. Let $A \subseteq \Sigma$ be a finite set of primitive actions. An element $a \in A$ is *minimal* in A iff $x \leq a$ implies $x = a$, for all $x \in A$. We write $\min(A)$ for the set of all minimal elements of A.

Let λ denote the empty sequence. The set of linearizations $\ell(A) \subseteq \Sigma_\tau^*$ of a transaction $A \in T$ with respect to dependency is defined inductively on the size of A as $\lambda \in \ell(\emptyset)$, and if $u \in \ell(A)$ and $a \in \min(A \cup \{a\}) \setminus A$, then $\tau u \in \ell(A)$ and $au \in \ell(A \cup \{a\})$. One reason to allow arbitrary interleaving of τ-actions is that it allows us to encode transactions with an explicit terminating τ-step.

Example 34. *We have* $\ell(\emptyset) = \tau^*$, $\ell(\{a\}) = \tau^* a \tau^*$, *and*

$$\ell(\{a,b\}) = \begin{cases} \tau^* a \tau^* b \tau^* & \text{if } a < b, \\ \tau^* b \tau^* a \tau^* & \text{if } b < a, \\ \tau^*(a\tau^* b + b\tau^* a)\tau^* & \text{otherwise,} \end{cases}$$

where $a \neq b$ *and* τ^* *is the Kleene star operator applied on* τ, *and* $+$ *is union.* $\Diamond$

Definition 25 (Linearization)**.** *The linearization relation is coinductively defined as the largest*[12] *relation* $\rightsquigarrow \subseteq T^\omega \times \Sigma_\tau^\omega$ *that satisfies one of the following conditions. If* $A_0A_1A_2\cdots \rightsquigarrow a_0a_1a_2\cdots$, *then either*

[12]The largest relation exists, because the union of all $\rightsquigarrow$ that satisfy Definition 25 satisfies Definition 25

1. *for all $i \geq 0$, we have $A_i = \emptyset$ and $a_i = \tau$; or*

2. *for some $i, j \geq 0$, $A_i \neq \emptyset$, $a_0 \cdots a_j \in \ell(A_0)$, and $A_1 A_2 \cdots \rightsquigarrow a_{j+1} a_{j+2} \cdots$.*

The first item of Definition 25 considers the case of a sequence with the empty transaction only. In this case, the resulting linear behavior is the singleton set with τ actions only.

We explain informally the second item in Definition 25. The index i and the condition $A_i \neq \emptyset$ is to exclude item (1). The index j and the sequence $a(0)...a(j)$ is a linearization of A_0, which is the head of the sequence of transactions. The last part (i.e., $A_1 A_2 ... \rightsquigarrow a(j+1)a(j+2)...$) is the coinductive definition.

An empty transaction $A_i = \emptyset$, for $i \geq 0$ of a sequence $A_0 A_1 A_2 \cdots \in T^\omega$ is trailing iff $A_j = \emptyset$, for all $j \geq i$. Linearization ignores non-trailing empty transactions, as they can match with the empty sequence of actions. However, linearization recognizes trailing empty transactions and relates them to τ^ω.

Example 35. *We have $\emptyset^\omega \rightsquigarrow \tau^\omega$. Suppose that $a < b$. Then, $\{a, b\}^\omega \rightsquigarrow s$ if and only if $s \in (\tau^* a \tau^* b)^\omega$. Also, $(\{a, b\}\emptyset)^\omega \rightsquigarrow s$ if and only if $s \in (\tau^* a \tau^* b)^\omega$. Thus, linearization ignores non-trailing empty transactions. We also have $(\{a\}\{b\})^\omega \rightsquigarrow s$ if and only if $s \in (\tau^* a \tau^* b)^\omega$. Hence, linearization also confuses the transaction $\{a, b\}$ with the sequence of transactions $\{a\}\{b\}$.* $\Diamond$

For a transactional behavior $K \subseteq T^\omega$, we define $K\rightsquigarrow = \{s \in \Sigma_\tau^\omega \mid \exists \alpha \in K, \alpha \rightsquigarrow s\}$ to be the linear component that contains all linearizations of sequences from K. The set of linearizations $K\rightsquigarrow$ of K is generally not closed under trace equivalence.

Example 36. *Consider the actions $\Sigma = \{a, b, c, d\}$, with dependency $a < b$ and $c < d$. Let $\alpha = \{a, b\}\{c, d\}\emptyset^\omega$, $s = abcd\tau^\omega$, and $s' = ac\tau bd\tau^\omega$. Since τ is not related to b or d by Definition 23, we have $s \equiv s'$ (see Definition 24). Furthermore, we have $\alpha \rightsquigarrow s$, while $\alpha \not\rightsquigarrow s'$. Therefore, $\{\alpha\}\rightsquigarrow$ is not closed under trace equivalence.* $\Diamond$

Definition 26 (Weak linearization)**.** *The weak linearization relation is the composition $\rightsquigarrow\equiv$ of linearization and trace equivalence.*

That is, $\alpha \rightsquigarrow\equiv s$ if and only if $\alpha \rightsquigarrow s' \equiv s$, for some $s' \in \Sigma_\tau^\omega$. By construction, $K\rightsquigarrow\equiv = \{s \in \Sigma_\tau^\omega \mid \exists \alpha \in K, \alpha \rightsquigarrow\equiv s\} = [K\rightsquigarrow]$ is closed under trace equivalence.

2.3.3 Problem statement: compositional linearization

Let $T \subseteq \mathcal{P}_f(\Sigma)$ be a set of transactions over the set of actions Σ. Let $\mathbb{T} = \mathcal{P}(T^\omega)$ be the space of transactional behaviors with $\cap$ as composition operator. Let $\mathbb{L} \subseteq \mathbb{T}$

be the space of linear behaviors with an arbitrary commutative, associative, idempotent product $\otimes$ as composition operator. As introduced in previous subsection, the linearization relates transactional behaviors to linear behaviors.

Let $\varphi : \mathbb{T} \to \mathbb{L}$ be a linearization function. Then, φ is a compositional linearization if and only if φ is a homomorphism, i.e., for all behaviors $K_1, K_2 \in \mathbb{T}$, $\varphi(K_1 \cap K_2) = \varphi(K_1) \otimes \phi(K_2)$. Not all compositional linearization give rise to a *useful* linearization. As an example, take the trivial linearization that maps every transactional behaviors to the singleton set containing the sequence of empty transaction. While being compositional, such linearization does not preserve the intended semantics. Instead, we consider *valid* linearizations.

Definition 27 (Valid linearization)**.** *Let $\varphi : \mathbb{T} \to \mathbb{L}$ be a linearization. Then, φ is* valid *if:*

1. *φ is compositional, i.e., for all $K, K' \in \mathbb{T}$, $\varphi(K \cap K') = \varphi(K) \otimes \varphi(K')$; and*
2. *$\rightsquigarrow\!\equiv$ is total and surjective on $K \times \varphi(K)$ for all $K \in \mathbb{T}$.*

The first item in Definition 27 implies that φ preserves composition, which allows us to linearize in parts and assemble later (possibly at run time). This is particularly useful for the inclusion of closed-source third-party software, which is precompiled by its vendor. Furthermore, first item in Definition 27 can be used to speed up application of updates, since parts that are not changed do not need to be linearized again.

The second item in Definition 27 means that, for every behavior $K \in \mathbb{T}$, every trace in $\varphi(K)$ is a linearization of some trace in K, and every trace in K has some linearization in $\varphi(K)$. In other words, the second item in Definition 27 asserts that a valid linearization does actually 'linearize'. Moreover, the equivalence relation $\equiv$ enables commutation of events that are independent, which corresponds to the second clause of the definition of atomicity of a transaction, as defined in Section 2.1.

Remark 10. *Delinearization is the translation of a linear behavior to a transactional behavior. If a linearization φ is injective, it naturally comes with a delinearization φ^{-1} that, for those linear behaviors in the image of φ, gives back a transactional behavior.* $\Diamond$

Note that $\rightsquigarrow$ as defined in Section 2.3.2 is not compositional for $\otimes = \cap$, and therefore not valid. As an example, let $K_1 = (\emptyset\{a\})^\omega$ and $K_2 = (\{a\}\emptyset)^\omega$. Then, $K_1 \cap K_2 = \emptyset$ while $K_1{\rightsquigarrow} \cap K_2{\rightsquigarrow} = (\tau^* a \tau^\omega) \cap (\tau^* a \tau^\omega) = \tau^* a \tau^\omega$. In the next section, we characterize all valid linearizations, and we give two practical instances.

2.3.4 Valid linearizations: lock step and interleaving

As defined in Definition 27, a *valid linearization* is a function that maps a transactional behavior to a linear behavior while preserving the compositional structure. We seek a morphism whose co-domain is a linear behavior from the set of transactions of its domain.

The lock-step linearization is a valid linearization which we present first. Each linearization is delineated with a special τ symbol. The resulting linear behavior has, for each trace, a τ symbols that delineates every transaction, and is therefore homomorphic with set intersection as the composition operation. While intuitive, such linearization is not efficient and requires every component in the intersection to run in lock-step.

We give another instance of a valid linearization, where we relax the lock-step behavior to tolerate some interleaving of independent actions.

Lock-step linearization A straightforward example of a valid linearization is based on synchronous rounds. Here, we use the τ action (also the empty transaction) to indicate the end of a round. This allows us to reconstruct the transactions from a sequence of actions. To be precise, we define *grouping* coinductively as follows.

Definition 28 (Grouping)**.** *The grouping relation is the largest*[13] *subset* $G \subseteq \Sigma_\tau^\omega \times T^\omega$*, such that* $(a_0 a_1 \cdots, A_0 A_1 \cdots) \in G$ *implies*

1. $a_0 = \tau$*,* $A_0 = \emptyset$*, and* $(a_1 a_2 \cdots, A_1 A_2 \cdots) \in G$*; or*
2. $a_0 \in \min(A_0)$ *and* $(a_1 a_2 \cdots, (A_0 \setminus \{a_0\}) A_1 \cdots) \in G$

Example 37. *If* a *and* b *are independent, then* $(ab\tau\tau ba\tau^\omega, \{a,b\}\emptyset\{a,b\}\emptyset^\omega) \in G$*. If* a *and* b *are such that* $a < b$*, then* $(ab\tau^\omega, \{a,b\}\emptyset^\omega) \in G$ *but* $(ba\tau^\omega, \{a,b\}\emptyset^\omega) \notin G$*.* ◊

Lemma 17. *G is a functional relation.*

Hence, we find a partial function $g : \Sigma_\tau^\omega \rightharpoonup T^\omega$ with $g(a_0 a_1 \cdots) = A_0 A_1 \cdots$ if and only if $(a_0 a_1 \cdots, A_0 A_1 \cdots) \in G$. The domain of g consists of all sequences $s \in \Sigma_\tau^\omega$, such that τ occurs infinitely often in s, and every primitive action occurs at most once between consecutive τ actions. We define the linearization based on synchronous rounds as the preimage of grouping g.

[13] Again, the largest relation exist.

Definition 29. *For every transactional behavior $K \subseteq T^\omega$, we define*

$$g^{-1}(K) = \{s \in dom(g) \mid g(s) \in K\}$$

It is straightforward to check that g^{-1} is a valid linearization, if we take intersection $\cap$ as the composition operator $\otimes$ on $\mathcal{P}(\Sigma_\tau^\omega)$.

Theorem 4. *g^{-1} is a valid linearization, for $\otimes = \cap$.*

Although Theorem 4 shows that the linearization based on synchronous rounds is valid, the resulting linear behaviors are locking the execution into strict rounds. As a result, independent actions can only swap within a transaction but not over sequences of transactions.

Example 38 (Parallel Syncs)**.** *Consider a Sync channel K_1 from a to b and a Sync channel K_2 from c to d. Set $\Sigma = \{a, b, c, d\}$ with $a < b$ and $c < d$. The composition of K_1 and K_2 is the behavior*

$$K_1 \cap K_2 = (\{a, b, c, d\} + \{a, b\} + \{c, d\} + \emptyset)^\omega.$$

Then, $g^{-1}(K_1 \cap K_2)$ equals

$$(a(bcd + c(bd + db))\tau + c(dab + a(bd + db))\tau + ab\tau + cd\tau + \tau)^\omega$$

Now consider the prefix acb of a behavior in $g^{-1}(K_1 \cap K_2)$. Ignoring the c from the second Sync K_2, we see that acb completes an $\{a, b\}$ transaction of the first Sync K_1. One would expect that the transactions $\{a, b\}$ and $\{c, d\}$ are independent, because they are disjoint and their respective actions are not related by the partial order. However, $g^{-1}(K_1 \cap K_2)$ dictates that the second Sync K_2 must complete its $\{c, d\}$ transaction before anything else can happen. Hence, first Sync K_1 can accept new input only after $acbd\tau$. Consequently, the concurrency of the transaction $\{a, b\}$ and $\{c, d\}$ is weaker than expected: while actions a and b are independent with actions c and d, the two transaction $\{a, b\}$ and $\{c, d\}$ cannot interleave arbitrarily.

We want to design a valid linearization φ, such that, for instance, the sequence $(acdcdb\tau)^\omega$ is part of $\varphi(K_1 \cap K_2)$. ◊

Interleaving tolerant linearization To avoid the oversynchronization in Theorem 38, we must allow traces that minimize the explicit τ action, i.e., the τ inserted after every round in the lock-step linearization. Observe that $g^{-1}(K) \subseteq dom(g)$, for

every behavior $K \subseteq T^\omega$. The set $dom(g)$ consists of sequences over Σ_τ that contain infinitely many τ actions and between any two τ actions, every primitive action happens at most once.

Recall the linearization relation $\rightsquigarrow$ from Section 2.3.2 and from Section 2.3.1 that $[L]$ is the closure of $L \subseteq \Sigma_\tau^\omega$ with respect to trace equivalence $\equiv$. Consider the map φ defined, for all behavior $K \subseteq T^\omega$, as

$$\varphi(K) = g^{-1}(K) \cup ([K\rightsquigarrow] \setminus dom(g)),$$

where $[K\rightsquigarrow]$ is the closure modulo trace equivalence of the set of all linearizations of behaviors from K. Intuitively, the linearization φ adds to the image of g^{-1} some linear behaviors that are not in the domain of the grouping. Observe that φ is injective, since

$$g(\varphi(K) \cap dom(g)) = g(g^{-1}(K)) = K.$$

Consider the composition operator $\otimes$ defined, for behaviors $L_1, L_2 \subseteq \Sigma_\tau^\omega$, as

$$L_1 \otimes L_2 = L_1 \cap L_2 \cap [L_1 \cap L_2 \cap dom(g)]. \tag{2.5}$$

Lemma 18. *For $L \subseteq dom(g)$, we have $[L] = [g(L)\rightsquigarrow]$.*

Theorem 5. *φ is valid, with $\otimes$ defined in Equation 2.5.*

Example 39 (Parallel Syncs with concurrency)**.** *Consider the Sync channels as in Example 38. Then, the linear behavior $\varphi(K_1 \cap K_2)$ contains runs like $acbababd\tau^\omega$. Although actions a and b are independent to actions c and d, the linearization g^{-1} forbid arbitrary interleaving of transactions $\{a, b\}$ and $\{c, d\}$. As a result, the linearization φ allows for such arbitrary interleaving.* ◊

2.4 Related work and future work

(De)composition. In [27] , the authors present a declarative and an operational theory of components, for which they define a refinement relation and compositionality results for some composition operators. Our work is related as it aims for similar results, but for the case of Cyber-Physical systems. Thus, instead of having input and output actions, components have timed observations, and composability relations. We present, as well, quotient operation on components, and show how it can be used to synthesize coordinating CPSs.

In [69], the authors consider the problem of decomposition of constraint automata. This work provides a semantic foundation to prove that the construction in [69] is a valid division.

Algebra, co-algebra The algebra of components described in this paper is an extension of [57]. Algebra of communicating processes [35] (ACP) achieves similar objectives as decoupling processes from their interaction. For instance, the encapsulation operator in process algebra is a unary operator that restricts which action to occur, i.e., $\delta_H(t \parallel s)$ prevent t and s to perform actions in H. Moreover, composition of actions is expressed using communication functions, i.e., $\gamma(a, b) = c$ means that actions a and b, if performed together, form the new action c. Different types of coordination over communicating processes are studied in [20]. In [12], the authors present an extension of ACP to include time sensitive processes.

The modeling of component's interaction using co-algebraic primitives is at the foundation of the Reo language [7]. In [18], the question of separation of components into two sub-components is addressed from a co-algebraic perspective.

Discrete Event Systems Our work represents both cyber and physical aspects of systems with a unified model of discrete event systems. In [51], the author lists the current challenges in modelling cyber-physical in such a way. The author points to the problem of modular control, where even though two modules run without problems in isolation, the same two modules may block when they are used in conjunction. In [74], the authors present procedures to synthesize supervisors that control a set of interacting processes and, in the case of failure, report a diagnosis. An application for large scale controller synthesis is given in [66].

Coordination In [67], the author describes infinite behaviors of process and their synchronization. Notably, the problem of non-blockingness is stated: if two processes eventually interact on some actions, how to make sure that both processes will not block each others. The concept of centrality of a process is introduced.

Transactional components Transactional components such has in Reo have been extensively studied from a formal perspective. In [42], Jongmans presents over 30 semantics for Reo. Current work adds an intermediate linear semantics for Reo, for which proving the correctness of an implementation as given in [70, 71, 41, 32] would

be possible. We also believe that the results presented in this paper may be of benefit to other similar semantics, such as [21].

From a trace theoretical perspective, the notion of transaction has also been studied. In [39], elements in $\mathcal{P}_f(\Sigma)^*$ are also called step sequences. However, the axiom $C = DE$ mentioned in [39], for $C, D, E \subseteq \Sigma$ with $D \cap E = \emptyset$, allows one to split a step into two consecutive substeps, which does not apply in our case, because our transactions are atomic and cannot be split. So, we consider the case of traces, where actions are sets. In [36], Gastin investigates the problem of reconstructing sequences of transactions from a sequence of actions investigated. The Foata normal form, defined in [36], Definition 2.10, partitions a trace into sets of mutually independent actions, and is used for this purpose. Unfortunately, not every sequence of transactions emerges as a Foata normal form, which makes the normal form not suitable for our purpose.

Linearization The linearization of transactional behaviors to linear behaviors has been approached in the context of databases queries and transactions. In [34], the authors consider the problem of serialization, which aims at reordering a sequence of events into a sequence of individual transactions. In their work, a transaction is a sequence of events that starts with a unique beginning symbol and ends with a unique final symbol. Our work relaxes the assumption that each transaction needs two delimiters.

Synchronous and asynchronous The composition of synchronous systems with asynchronous systems has been investigated in the context of interconnecting machines on a network. Globally asynchronous locally synchronous (GALS) is an architecture for designing electronic circuits which addresses the problem of safe and reliable data transfer between independent clock domains [26]. In our paper, we do the opposite: locally asynchronous globally synchronous (LAGS). For example, in the Sync channel, we split the locally synchronous $\{a, b\}$ transaction into asynchronous a and b steps, and recover the $\{a, b\}$ transaction via global synchronization.

Chapter 3

Reo as an algebra of order sensitive components

In this Chapter, we instantiate the algebra of components introduced in Chapter 2 to compositionally design, compile, and analyse order sensitive components. Order sensitive components are components for which only the fact that an observation *happens before* another observation matters, but not the absolute time at which the observation occurs nor the interval of time between two observations. Many real world applications fall in the class of interacting order sensitive components, and dedicated design languages, such as the Reo coordination language [6, 3, 43], focus on formally specifying the interaction occurring among such components.

Reo protocols coordinate components by means of elementary and composite connectors. The behavior of a whole system based on these components is mainly defined by its Reo protocol, which makes the reliability of the coordination protocol a central part of the verification. We list in Table 3.1 few properties of interest to study the temporal behavior of connectors. We note a port p in italic, together with some key words such as *silent* and *always*. Reo semantics is at the level of transactional components: ports fire within transactions. An executable of a Reo specification (e.g., an implementation) would however use sequential (yet concurrent) primitives, and falls into the category of linear components. This Chapter discusses a possible implementation of the linearization introduced in Section 2.3. The property of *synchrony* captures that two ports a and b fire together. In a sequential setting, we allow, as shown in Section 2.3, silent steps between firing of a and firing of b: we call this relation *a then*

b.

Table 3.1: Properties on firing of ports and their meaning

Properties	Relations
a fires	exchange of data at a
silent	no firings
a before b	a fires then eventually b fires
a then b	a fires then *silent* until b fires
synchrony (sync)	*always* a fires iff b fires
asynchrony (async)	*always* a before b or b before a

We show that the property of synchrony on transactional components is then reflected by the property a then b on a sequential setting. To experiment and verify temporal properties, we give a translation from Reo to Promela. Promela, the specification language used for the LTL model checker SPIN, is sequential and relies on message passing through channels. Some tools currently exist to translate a Reo circuit into a language used by a model checker [50]. However, no general mechanism is described to deal with the translation of synchronous properties on Reo circuits to asynchronous properties on the target language. This section extends a prior work[58].

We express, in Section 3.1, the formal semantics of Reo as an algebra of order sensitive components. We show that this algebra can express Reo connectors using the primitive notion of a port component, and a set of interaction signatures. We use the results of Chapter 2 to show algebraic properties of the Reo composition operator, and discuss some connector equivalences. Section 3.2 introduces a logical specification of Reo connector as guarded commands. The semantics of such connector is linked to the Reo semantics by considering sequences of observations that satisfy the connector's constraint. Section 3.3 is motivated by the extension [55] of our Reo compiler to generate verifiable specification using the SPIN model checker [38]. We implement a translation from a logical specification of Reo connectors to Promela, an executable language, and show that properties of synchrony are preserved during translation. We use the SPIN model checker to investigate some temporal properties of basic Reo connectors, and give a general framework and domain specific language for verifying temporal properties of Reo protocols.

3.1 Reo

In Chapter 2, we introduced an algebra of components and their parametrized products. We also presented properties of components, such as being order sensitive. The order sensitivity property manifests the fact that a component behavior is independent to the concrete time value of its observations, and only the order of observations matters. For instance, the time at which a packet is sent on a network is usually irrelevant, but the order at which each packet arrives at its destination is important.

We show in this section how to define Reo connectors (and Reo primitives) using our algebra of components. More precisely, we give a description of Reo primitives (like a merger, a replicator, a fifo1, etc.) as product of port components under some interaction signatures. This way, we open new reasoning about circuit equivalences based on the underlying algebraic laws of such operators (e.g., associativity, commutativity, and distributivity).

A *component* in Reo has an *interface* consisting of a set of ports, and a *behavior* specifying the sequence of data flowing at each port. Components interact with other components via shared ports by *agreeing* on the data flowing at that port.

Current work on Reo focuses particularly on a specific class called *connector*. A connector specifies the exchange or transformation of data only, but not its creation nor its deletion. Therefore, a connector has open ended input and outputs ports, to which consumer and producers are eventually placed. Typically, for a connector to have observable behaviors, components plugged at the input ports must *feed* data into that port, while components at the output ports must *consume* data through that port. As an example, the *sync* connector in Reo has an input and output open ended points, and synchronously forwards input data to its output. If connected to two components, one at each endpoint, a sync connector essentially models a simultaneous send and receive operation between those two components. Connectors may be connected to each others, forming of more complex input/output relations [6].

Formally, each port denotes a set of Timed-Data Streams (TDS), which are infinite sequences of a datum paired with a time stamp. Intuitively, a Timed-Data Stream tells what data flows at what time through a port, and faithfully transcribes the observable behavior of that port. A component denotes a set of tuples of TDSs, where each port from the component's interface is uniquely represented by a TDS in each tuple. A component therefore specifies which of the TDS tuples are accepted or rejected. We study properties of Reo connectors expressed as a fragment of the component framework of Chapter 2.

The first model for Reo was introduced in [3], and made use of timed-data streams where each port labels its firing with a time stamp. The representation of time as a real value is motivated by the necessity to allow arbitrarily many finite interleavings between two observations. Most of Reo connectors, however, are order sensitive, which implies that the precise value of the time stamp of an observation does not matter, only the order of observations matters.

Port as a component In Reo, the most primitive form of computations take place at a port. A port denotes events that occur over time at a unique location. Often, a port does not restrict which sequence of events may occur, and captures all possible such sequences.

We therefore model a port as a unary component from the model introduced in Chapter 2. We use $P_a(D) = (E_a(D), L_a)$ to denote the port a with domain D where $E_a(D) = \{(a, d) \mid d \in D\}$ that contains all events for port a, and its behavior $L_a \subseteq TES(E_a(D))$ contains all TESs with singleton or empty observations, i.e., such that $\sigma \in L_a$ implies that for all $i \in \mathbb{N}$, $|\sigma(i)| \leq 1$. When the context is clear, we drop the domain of a port and simply write $P_a = (E_a, L_a)$. Note that a port is an order sensitive component, as only the order between occurrences of events matter, but not the exact time nor the time interval between two observations.

Example 40 (Alternating port)**.** *We define $P_m = (E_m, L_m)$ where $E_m = \{(m, 0), (m, 1)\}$ and $\sigma \in L_m \subseteq TES(E_m)$ if and only if, for all $i \in \mathbb{N}$, $\sigma(2i) = (\{(m, 0)\}, t_i)$ and $\sigma(2i + 1) = (\{(m, 1)\}, t_{i+1})$, which consists of a stream of alternating bits at port m.*

Interaction signature Following Example 11, we define three main interaction signatures that are used to form binary and n-ary Reo components.

The *synchronous* signature $\Sigma^{sync}_{(a,b)} = ([\kappa^{sync}_{(a,b)}], [\cup])$ enforces events at port a to occur at the same time as the same event at port b, i.e., $((O_1, t_1), (O_2, t_2)) \in \kappa^{sync}_{(a,b)}(E_a, E_b)$ if and only if

$$
\begin{aligned}
t_1 < t_2 &\implies O_1 \cap E_b = \emptyset \wedge \\
t_2 < t_1 &\implies O_2 \cap E_a = \emptyset \wedge \\
t_2 = t_1 &\implies \forall d.((a, d) \in O_1 \iff (b, d) \in O_2)
\end{aligned}
$$

Note that κ^{sync} from Example 11, when restricted to observations occurring at ports, is the symmetric and reflexive relation that synchronizes the occurrences of events at every shared port, i.e., κ^{sync} is the union of all $\kappa^{sync}_{(x,x)}$ with x a port name.

The *asynchronous* signature $\Sigma^{async}_{(a,b)} = ([\kappa^{async}_{(a,b)}], [\cup])$ prevents events from port a and b to occur at the same time, i.e., $((O_1, t_1), (O_2, t_2)) \in \kappa^{async}_{(a,b)}(E_a, E_b)$ if and only if

$$\forall d_1, d_2.((a, d_1) \in O_1 \wedge (b, d_2) \in O_2) \implies t_1 \neq t_2$$

The *relational* signature $\Sigma^{rel}_{(a,b,\sqcap)} = ([\kappa^{rel}_{(a,b,\sqcap)}], [\cup])$, for $\sqcap \subseteq E_a \times E_b$ a relation, relates an event (a, d_1) from port a to a simultaneous event (b, d_2) at port b such that $((a, d_1), (b, d_2)) \in \sqcap$, i.e., $((O_1, t_1), (O_2, t_2)) \in \kappa^{rel}_{(a,b,f)}(E_a, E_b)$ if and only if

$$\begin{aligned}
t_1 < t_2 &\implies O_1 \cap dom(\sqcap) = \emptyset \wedge \\
t_2 < t_1 &\implies O_2 \cap codom(\sqcap) = \emptyset \wedge \\
t_2 = t_1 &\implies (\forall d_1.(a, d_1) \in O_1 \cap dom(\sqcap) \\
&\qquad \implies (\forall d_2.(b, d_2) \in O_2 \cap codom(\sqcap) \\
&\qquad\quad \implies ((a, d_1), (b, d_2)) \in \sqcap)
\end{aligned}$$

where $dom(\sqcap) \subseteq E_a$ and $codom(\sqcap) \subseteq E_b$ are the sets of events in the domain and co-domain of the relation $\sqcap$. Every event that is not in the related by $\sqcap$ can therefore freely occur at anytime. Note that if $\sqcap$ is the identity relation on the data element, i.e., $(a, d_1), (b, d_2) \in \sqcap$ implies $d_1 = d_2$, then $\Sigma^{rel}_{(a,b,\sqcap)}$ is equal to the signature $\Sigma^{sync}_{(a,b)}$.

Example 41 (Binary component)**.** *Let a be a port, and m an alternating port. Let f_0 be the functional relation such that $f_0(a, v) = (m, 0)$ for all $(a, v) \in E_a$, the product $P_a \times_{\Sigma^{rel}_{(a,m,f_0)}} P_m$ represents the component that synchronizes all values at port a with the value 0 at port m. Reciprocally, fixing the functional relation f_1 to be such that $f_1(b, v) = (m, 1)$ for all $(b, v) \in E_b$, the product $P_b \times_{\Sigma^{rel}_{(b,m,f_1)}} P_m$ represents the component that synchronizes all values at port b with the value 1 at port m.*

The *delay* signature $\Sigma^{delay}_{(a,b,\sqcap)} = ([\kappa^{delay}_{(a,b,\sqcap)}], [\cup])$, for port a, b, and a relation $\sqcap \subseteq E_a \times E_b$, restricts every occurrence of data at port a to be related to a later observable at port b, i.e., $((O_1, t_1), (O_2, t_2)) \in \kappa^{delay}_{(a,b,\sqcap)}$ if and only if

$$t_1 < t_2 \implies (\forall d_1.(a, d_1) \in O_1 \implies (\forall d_2.(b, d_2) \in O_2 \implies ((a, d_1), (b, d_2)) \in \sqcap))$$

When $\sqcap$ is the identity relation on the data element, i.e., $((a, d_1), (b, d_2)) \in \sqcap$ implies $d_1 = d_2$, we simplify the notation to write $\Sigma^{delay}_{(a,b)}$.

Reo syntax Reo has a graphical syntax that visualizes composition of components. Figure 3.1 shows three kinds of components: *synchronous channels*, *asynchronous channels* and *nodes*. In this section, we describe the semantics of these kinds only intuitively. Typically, we call a binary component a channel and certain kinds of other components nodes.

First, we consider two synchronous channels. The *sync* channel in Figure 3.1 represents a synchronous transfer of data from port a to port b. The *syncdrain* in Figure 3.1 models a synchronous firing of port a and b without necessarily equating data at those ports. There are also asynchronous channels. A *fifo1* channel (depicted in Figure 3.1) has an internal buffer with the capacity to hold one data item. This buffer is initially empty. When its buffer is empty, a *fifo1* channel accepts a data item through its input port a, places it in its buffer, which then becomes full. When its buffer is full, a *fifo1* channel no longer accepts any intput. When the buffer of a *fifo1* channel is full, the channel delivers the content of its buffer to a get operation performed by the environment on its output port b, and its buffer becomes empty. A get on the output port of a *fifo1* channel with an empty buffer blocks until after its buffer becomes full. Thus, the get and put operations on the ports of a *fifo1* channel succeed only asynchronously: never together. The *asyncdrain* channel in Figure 3.1 never allows a pair of put operations on its boundary ports to succeed synchronously. Finally, Figure 3.1 has two ternary components: a *merger* and a replicator. A merger synchronizes data transfer through at most one of its input boundary ports with data transfer through its output port. If data is available at both input ports, a *merger* non-deterministically chooses one to synchronize with its output port. A *replicator* forwards the data from its input port to both of its output ports. All ports must be ready for the replicator to proceed. A *filter* forwards the data from its input to its output if the predicate ϕ labeling the filter holds on the input data. In the case where the predicate ϕ does not hold on the data at its input, the data is lost. Oppositely, the *b*locking *filter*, written *bfilter*, blocks when the data at its input violates the constraint ϕ. See Section 4.3 for the use of Reo primitives to construct connectors.

Reo semantics Typically, the composition operator is fixed in Reo to be $\times^{sync}$ that joins behaviors of components on shared port names. We use the notation $\bowtie$ to denote such operation. Reo fixes the semantics of a node [6], whereas all channels are user defined. There is, however, a commonly useful set of channels (see Figure 3.1) that we use in this section. We define algebraically some common channels out of the port component introduced earlier and few interaction signatures that we listed.

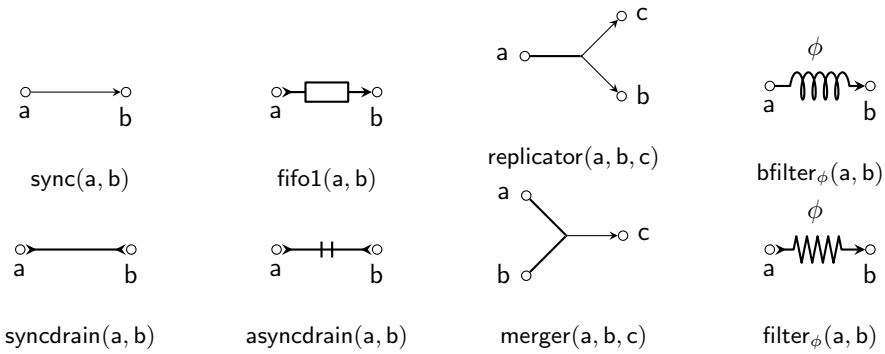


Figure 3.1: Graphical syntax for some primitives.

More generally, we define Reo components as a fragment of our component algebra:

$$C := C \times_\Sigma C \mid P_x$$

where $\Sigma \in \{\Sigma^{sync}, \Sigma^{sync}_{(a,b)}, \Sigma^{async}_{(a,b)}, \Sigma^{delay}_{(a,b,\sqcap)}, \Sigma^{rel}_{(a,b,\sqcap)}\}$ for some port names a, b, x, for some relation on events $\sqcap$.

We define the following Reo channels and nodes:

$$\begin{aligned}
\mathit{sync}(a,b) &= P_a \times_{\Sigma^{sync}_{(a,b)}} P_b \\
\mathit{syncdrain}(a,b) &= P_a \times_{\Sigma^{rel}_{(a,b,\sqcap)}} P_b \ \mathit{with}\ \sqcap = (E_a \times E_b) \\
\mathit{filter}(a,b,f) &= P_a \times_{\Sigma^{rel}_{(a,b,\sqcap_f)}} P_b \ \mathit{with}\ \sqcap_f = \{((a,d),(b,f(d)) \mid d \in \mathit{dom}(f)\} \\
\mathit{fifo1}(a,b,M) &= (P_a \times_{\Sigma^{rel}_{(a,m,f_0)}} P_m) \times_{\Sigma^{delay}_{(a,b)} \cup \Sigma^{sync}_{(m,m)}} (P_b \times_{\Sigma^{rel}_{(b,m,f_1)}} P_m) \\
\mathit{merger}(a,b,c) &= (P_a \times_{\Sigma^{excl}_{(a,b)}} P_b) \times_{\Sigma^{sync}_{(a,c)} \cup \Sigma^{sync}_{(b,c)}} P_c
\end{aligned}$$

with $\Sigma^{delay}_{(a,b)} \cup \Sigma^{sync}_{(m,m)} = ([\kappa^{delay}_{(a,b)}] \cup [\kappa^{sync}_{(m,m)}], [\cup])$ and $\Sigma^{sync}_{(a,c)} \cup \Sigma^{sync}_{(b,c)} = ([\kappa^{sync}_{(a,b)}] \cup [\kappa^{sync}_{(b,c)}], [\cup])$ and $\sqcap_f = \{(\{(a,d_1)\}, \{(b,d_2)\}) \mid d_2 = f(d_1)\ \mathit{or}\ \}$. The $\mathit{sync}(a,b)$ component is such that the data observed at port a and b are equal and synchronous, i.e., occurs at the same time. The $\mathit{syncdrain}(a,b)$ component ensures that both the data of a and b are observed at the same time, but does not restric their data to be equal. The component $\mathit{fifo1}(a,b,M)$ synchronizes the observation of a data at a with the change of the memory state M, and then outputs the same data at b. As defined here, the $\mathit{fifo1}(a,b,M)$ component is infinitely productive, i.e., always eventually has an input at a and an output at b. The component $\mathit{filter}(a,b,f)$ synchronizes events from a with

event from b related by the function f. Any unrelated event at a or b can freely happen. Note that, in the case that f is the identity, we recover the Reo filter behavior where the condition $d \in dom(f)$ denotes some predicate ϕ. The blocking filter behavior can be encoded by composition of a non-blocking filter and other Reo primitives. The $merger(a, b, c)$ component either synchronizes a with c or b with c but never all ports together.

A strength of Reo is its compositional nature: protocols are built out of primitives. We use the join operation defined in Example 7 (see Theorem 1) for the proof of associativity and commutativity of $\bowtie$) to define two Reo connectors:

$$\begin{aligned} \mathit{alternator}(a, b, c) = \mathit{sync}(a, c_1) \bowtie \mathit{fifo1}(x, c_2) \bowtie \mathit{syncdrain}(a, b) \bowtie \\ \mathit{sync}(b, x) \bowtie \mathit{merger}(c_1, c_2, c) \\ \mathit{fifo2}(a, b) = \mathit{fifo1}(a, x, M_1) \bowtie \mathit{fifo1}(x, b, M_2) \end{aligned}$$

3.2 Logical specification of connector components

Language of constraints We formally characterize the behavior of a component as a predicate relating the data flow through its ports. Note that we first study the behavior of a component in its ideal environment, i.e. all input and output sequences are possible. The resulting behavior that a component describes is a set of tuples of data streams representing the synchronous flowing of data through the ports of its interface.

Data elements that flow through ports and get stored in memory belong to a domain that we call D. In this work, we use a unique domain for all port and memory, since we do not use any algebraic operations on data. Note that Reo allows more structured data elements exchanged through ports and memories. As we will later see in the characterization of components' behavior, the need of talking about the case where no data is observed at a port or memory is of importance. The item $*$ is added to the data domain D, and D_* denotes the resulting data domain.

Ports and memories appear as variables in the logical characterization. Port and memory variables take values in the domain D_*. The set of port variables is denoted as P, and M denotes the set of memory variables. While ports do not have any memory (as mentioned in the previous paragraph), memories always store the previous data item. For each memory variable $m \in M$, there is a memory variable $m' \in M'$. Their interpretation becomes clear in the next paragraph.

User defined components are characterized by a user supplied predicate or function

among the ports of its interface. The sets Q and F respectively denote the set of n-ary predicate symbols and n-ary function symbols.

A *term* is either a variable $p \in P$, $m \in M$, or $m' \in M'$, an n-ary function application $f(t_1, ..., t_n)$ where $f \in F$ is an n-ary function symbol, or a constant $d \in C$.

A *formula* is built inductively by:

$$\phi \ ::= t_1 = t_2 \mid B(t_1, ..., t_n) \mid \phi_1 \wedge \phi_2 \mid \neg\phi$$

Where $B \in Q$ is a predicate symbol. The set of formula expressions is denoted by $\mathcal{F}$. We use the shorthand notation $t_1 \neq t_2$ for $\neg(t_1 = t_2)$, $\bot$ for $t_1 \neq t_1$, and we get $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$ as well as $\phi \implies \psi = \neg\phi \vee \psi$.

We allow existential quantifier at the outermost left position of a formula. Quantifiers range over port variables only. A port occurring in ϕ but not existentially quantified is called a *free* port variable. We call *atomic* a formula that is either an equality, an inequality, or a predicate. We call V_ϕ the set of free variables occurring in ϕ, and similarly $P_\phi \subseteq V_\phi$ and $M_\phi \subseteq V_\phi$ for the set of free port and memory variables. When ϕ is clear from the context, we drop the subscript notation.

Example 42. *A Reo component, as introduced earlier, constrains the flow of data through its boundary ports. We call constraint of the component the logical formula used to relate the data flowing through the ports. We usually write* $\mathsf{comp}(a, b)$ *for the formula of the component comp having as free port variables* a *and* b*. We give below two examples for components sync and fifo1.*

Given a port $p \in P$*, the proposition of whether or not* p *fires is encoded as an equality between* p *and elements of* D_**. We say that* p *fires if* $p \neq *$*. On the other hand,* p *does not fire if* $p = *$*. Taking* $a, b \in P$*, we can now express that* a *and* b *fire synchronously with the same datum, as the formula* $a = b$*.*

$$\mathsf{sync}(a, b) = a = b \tag{3.1}$$

$$\begin{aligned}\mathsf{fifo1}(a, b) = {} & (m' = a \wedge a \neq * \wedge b = * \wedge m = b) \vee \\ & (m' = a \wedge a = * \wedge b \neq * \wedge m = b) \vee \\ & (m' = m \wedge a = * \wedge b = *)\end{aligned} \tag{3.2}$$

The constraint for a fifo1 channel has three clauses. The first one corresponds to filling the buffer with the data item observed at port a*; the second one empties the buffer through port* b*; and the last one corresponds to the case where no port fires, in which case the value in the buffer must remain unchanged.*

As hinted previously, protocols can be built by composing primitives. In the case of a composite component, the resulting constraint is defined as the conjunction of the constraints of the underlying components.

The logic is agnostic regarding the direction of data flow and merely represents the constraint on the data observed at each port.

Solution of constraints Every constant element of D_* get mapped to an element of the homonym domain D_*. Let γ be the map for every n-ary function symbol $f \in F$ to an element of $D_*^n \to D$. Let $\mathcal{I}$ be the map for every n-ary predicate symbol $B \in Q$ to an element of $D_*^n \to 2$. Let $\Gamma : V \to D_*$ be the interpretation function for variable symbols where $V = P \cup M \cup M'$. The interpretation of a term t, noted $[\![t]\!]_{(\Gamma,\gamma)}$, is the standard inductive interpretation providing the signature (Γ, γ)

A *solution* to a formula ϕ is an assignment Γ such that Γ *satisfies* ϕ, written as $\Gamma \models \phi$, defined inductively on ϕ as:

$$\Gamma \models \top \text{ always}$$
$$\Gamma \models t_1 = t_2 \text{ iff } [\![t_1]\!]_{(\Gamma,\gamma)} = [\![t_2]\!]_{(\Gamma,\gamma)}$$
$$\Gamma \models \phi_1 \wedge \phi_2 \text{ iff } \Gamma \models \phi_1 \text{ and } \Gamma \models \phi_2$$
$$\Gamma \models \neg\phi \text{ iff } \Gamma \not\models \phi$$
$$\Gamma \models \exists p\phi \text{ iff there exists } d \in D_* \text{ such that } \Gamma \models \phi[d/p]$$
$$\Gamma \models B(t_1, ..., t_n) \text{ iff } ([\![t_1]\!]_{(\Gamma,\gamma)}, ..., [\![t_n]\!]_{(\Gamma,\gamma)}) \in \mathcal{I}(B)$$

We extend the domain definition of an n-ary function from D^n to D_*^n by defining $f(t_1, ..., t_n) = * \iff t_1 = * \vee ... \vee t_n = *$.

Example 43. *Following the Example 42, the set of solutions for a sync and a fifo1 channel corresponds to the assignments for all free variables in the formula* $\mathsf{sync}(a, b)$ *and* $\mathsf{fifo1}(a, b)$ *that satisfy the formula. We use the data domain* $D = \{0, 1, *\}$*, and for clarity, we write* $(0, 1)$ *as the assignment that maps* $a \mapsto 0$ *and* $b \mapsto 1$*. In this context, the assignments* $(0, 0)$*,* $(1, 1)$*, and* $(*, *)$ *are the only assignments that satisfy the constraint* $\mathsf{sync}(a, b)$*. We write* $(0, 1, 0, 1)$ *the assignment that maps* a *to the first element,* b *to the second element,* m *to the third, and* m' *to the last element. Thus, considering the constraint* $\mathsf{fifo1}(a, b)$*, the assignment* $(0, *, *, 0)$ *and* $(1, *, *, 1)$ *both satisfy the constraint. In the case of the latter, the next value of the memory should now be equal to* 1*, and only assignment mapping the memory to* 1 *would be allowed. We explain in the next paragraph the behavior of a component as all the infinite sequences*

of assignments that satisfy its internal constraint.

3.2.1 Connector as guarded commands: an intermediate form

A component, if it has some open ports, is not in isolation but must co-evolve with its environment. The solution of its internal constraints must conform with the data provided or requested by the environment.

Guarded commands We propose a method based on guards and command to implement and simulate transactional behavior. The guard is a predicate on the state of the ports and the memory. If the guard is true, the update describes a constraint that both the component and its environment must satisfy. In an implementation language, updates are themselves sequential, which can introduce some interleaving in a concurrent setting. We later show, by making use of temporal properties, that the translation to an sequential (yet concurrent) implementation preserves the atomicity property.

We first refine the language of constraint and impose some requirement on ϕ to be a guarded command. We show some properties if ϕ satisfied those requirements.

We denote by v^{in} an input variable and v^{out} an output variable. An input term t^{in} refers to either a data element $d \in D_*$, an n-ary function $f(t_1^{in}, ..., t_n^{in})$ whose arguments are input terms $t_1^{in}, ..., t_n^{in}$.

A guards g is a conjunction of literals l defined as follows:

$$l ::= \ B(t_1^{in}, ..., t_n^{in}) \mid t_1^{in} \sim t_2^{in} \mid v^{out} \sim d$$

where $\sim \in \{=, \neq\}$.

A commands c is a conjunction of equalities of the kind $v^{out} = t^{in}$. We say that a formula ϕ is in *guarded command form* if

$$\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_j g_j)$$

where g_i are formulas in the language of guards and c_i are formulas in the language of commands.

Given ϕ in guarded command form, we call an implication of the type $g \implies c$ in ϕ, a guarded command of ϕ. We call the number of guarded commands in such a formula, the size of that formula. We denote the set of guards by $\mathcal{G}$, and the set of commands by $\mathcal{C}$. Note that guarded commands are quantifier free formulas.

We say that a quantifier free formula $\phi = \phi_1 \vee ... \vee \phi_n$ in disjunctive normal form and expressed in the language of constraints is *deterministic* if ϕ can be written with the grammar of guarded commands such that $\phi = g_1 \wedge c_1 \vee ... \vee g_n \wedge c_n$, where $\wedge$ has precedence over $\vee$, g_i are guards, c_i are commands, and $g_i \wedge g_j \equiv \bot$ for all i, j where $i \neq j$.

We assume that if a v^{out} is involved in an equality, then the other term is either $*$, or an input term t^{in}. In other words, if an equality $v_1^{out} = v_2^{out}$ appears in the constraint ϕ, there must exist an input term t^{in} such that $v_1^{out} = t^{in}$ and $v_2^{out} = t^{in}$ in order for ϕ to be written in the guarded command form. Moreover,

Proposition 1. *A deterministic formula $\phi = \bigvee_i (g_i \wedge c_i)$ can be written as a guarded command where:*

$$\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_i g_i)$$

where i ranges over the number of disjuncts of ϕ.

Proof. By induction on the structure of ϕ. We assume ϕ is in disjunctive normal form, and negation is pushed to the literals. We can write $\phi = \phi_1 \vee ... \vee \phi_n$ where ϕ_i are conjunctions of equalities, inequalities, or predicates.
Step 1 (identification): We syntactically partition each ϕ_i with its corresponding guard g_i and command c_i such that $\phi_i = g_i \wedge c_i$. We get the resulting formula $\phi = g_1 \wedge c_1 \vee ... \vee g_n \wedge c_n$. We syntactically substitute $g_i \wedge c_i$ by $(g_i \implies c_i) \wedge g_i$. We then get $\phi = (g_1 \implies c_1) \wedge g_1 \vee ... \vee (g_n \implies c_n) \wedge g_n$.
Step 2 (factorization): Because the formula ϕ is deterministic, for all $i \neq j$ we have $g_i \wedge g_j \equiv \bot$, which equivalently gives $g_i \wedge \neg g_j \equiv g_i$. We can then rewrite $(g_1 \implies c_1) \wedge g_1$ as $(g_1 \implies c_1) \wedge (g_2 \implies c_2) \wedge g_1$, since

$$\begin{aligned}(g_1 \implies c_1) \wedge (g_2 \implies c_2) \wedge g_1 &= g_1 \wedge c_1 \wedge (\neg g_2 \vee c_2) \\ &= g_1 \wedge c_1 \wedge \neg g_2 \vee g_1 \wedge c_1 \wedge c_2 \\ &\equiv g_1 \wedge c_1 \vee g_1 \wedge c_1 \wedge c_2 \\ &\equiv g_1 \wedge c_1 \\ &\equiv (g_1 \implies c_1) \wedge g_1\end{aligned}$$

By induction on the number of implications, we conclude that $(g_j \implies c_j) \wedge g_j$ is equivalent to $\bigwedge_i (g_i \implies c_i) \wedge g_j$ for all j, and thus:

$$\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_i g_i)$$

□

Given two guarded commands $\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_j g_j)$ and $\psi = \bigwedge_i (g'_i \implies c'_i) \wedge (\bigvee_j g'_j)$, we write the composition $\phi \wedge \psi$ as the formula:

$$\phi \wedge \psi = \bigwedge_i (g_i \implies c_i) \bigwedge_i (g'_i \implies c'_i) \wedge (\bigvee_{i,j} g'_i \wedge g_j)$$

Proposition 2. *Given two deterministic formulas ϕ and ψ, their product $\phi \wedge \psi$ is also deterministic.*

Proof. Since ϕ and ψ are deterministic, we can write $\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_j g_j)$ and $\psi = \bigwedge_i (g'_i \implies c'_i) \wedge (\bigvee_j g'_j)$ where for $i \neq j$, $g_i \wedge g_j \equiv \bot$ and $g'_i \wedge g'_j \equiv \bot$. The product $\phi \wedge \psi$ can be seen as:

$$\begin{aligned} \phi \wedge \psi &= (g_1 \wedge c_1 \vee ... \vee g_n \wedge c_n) \wedge (g'_1 \wedge c'_1 \vee ... \vee g'_n \wedge c'_n) \\ &= (g_1 \wedge g'_1 \wedge c_1 \wedge c'_1 \vee ... \vee g_n \wedge g'_n \wedge c_n \wedge c'_n) \end{aligned}$$

The new formula $\phi \wedge \psi$ inherits from $\phi \wedge \psi$ that for any i, j, k and l such that $i \neq j \vee k \neq l$, we have $g_i \wedge g'_k \wedge g_j \wedge g'_l \equiv \bot$. Therefore, $\phi \wedge \psi$ is also deterministic and can be written as a guarded command:

$$\phi \wedge \psi = \bigwedge_{i,j} ((g_i \wedge g'_j) \implies (c_i \wedge c'_j)) \wedge \bigvee_{i,j} (g_i \wedge g'_j)$$

□

As the construction in the proof of Proposition 2 shows, writing the composition of two deterministic formulas as a deterministic formula may increase the size of the resulting formula.

We present some optimizations that can be applied to formulas before forming their product. We use the formula $\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_i g_i)$ and $\psi = \bigwedge_j (g'_j \implies c'_j) \wedge (\bigvee_j g'_j)$ to illustrate these transformations, where i and j range over a finite set of natural numbers.

In general, the disjunction of guards $\bigvee_i g_i$ constituting the last part of the guarded command can imply a relation on the literals of the guards, and simplify the guards

themselves. We consider the example of the sync channel:

$$\begin{aligned}\phi_{sync} =&(a = * \wedge b = * \wedge a = b) \vee (a \neq * \wedge b \neq * \wedge a = b)\\ =&((a \neq * \wedge b \neq *) \implies a = b) \wedge\\ &((a = * \wedge b \neq *) \implies a = b) \wedge\\ &((a = * \wedge b = *) \vee (a \neq * \wedge b \neq *))\\ =&(a \neq * \implies a = b) \wedge (a \neq * \iff b \neq *)\end{aligned}$$

In this example, the guarded command form of the formula for *sync* induces a relation on a fires and b fires that simplifies the formula. It also means that in subsequent compositions, we can consider literals $a \neq *$ and $b \neq *$ as interchangeable. We consider as future work the exploitation of such relations that emerge from the disjunction of guards.

Given the product $\phi \wedge \psi$, if we find g_k and g'_l such that $g_k \iff g'_l$, we can equivalently consider $\phi \wedge \psi$ as the following formula:

$$\begin{aligned}\phi \wedge \psi =&((g_k \wedge g'_l) \implies (c_k \wedge c'_l)) \wedge\\ &\bigwedge_{i \neq k}(g_i \implies c_i) \bigwedge_{i \neq l}(g'_i \implies c'_i) \wedge (\bigvee_{i,j} g_i \wedge g'_j)\end{aligned}$$

In other words, if ϕ is of size N and ψ of size M, we decrease the size of $\phi \wedge \psi$ by 1 by identifying two guards, i.e., the size of $\phi \wedge \psi$ is $M + N - 1$. If we compare the resulting size of the disjunctive normal form, since $g_k \wedge g'_l$ is exclusive of all other guards g_i and g'_j for all $i \neq k$ and $j \neq l$, identifying two guards remove $M + N$ clauses from $M \times N$.

Example 44. *We now consider an example of guarded commands for the fifo1 primitive. The fifo1 primitive has a permanent constraint written in Fig. 3.2, with the io designation of $io(a) = 1$ and $io(b) = 0$, i.e., a is an input port and b is an output port. The formula of a fifo1 is deterministic, and with the result of Proposition 1, we can write its permanent constraint as a guarded command:*

$$\begin{aligned}\phi_{fifo1} = \ &(g_1 \implies c_1) \wedge (g_2 \implies c_1) \wedge (g_3 \implies c_2) \wedge\\ &(g_1 \vee g_2 \vee g_3)\end{aligned}$$

where we have the following guards $g_1 := (a \neq * \wedge b = * \wedge m = *)$, $g_2 := (a =$

$* \wedge b \neq * \wedge m \neq *$), *and* $g_3 := (a = * \wedge b = *)$; *and the following commands* $c_1 := (m' = a \wedge m = b)$, *and* $c_2 := m' = m$.

The formula for a fifo1 channel has three different guards. The first guard g_1 *checks whether its source port* a *is active, in which case the command* c_1 *fills the buffer with the data observed at port* a*; the second guard* g_2 *checks whether the channel's sink port* b *is active, in which case its command* c_1 *empties the buffer through port* b*. The last guard* g_3 *checks if the two ports* a *and* b *are inactive, and if true, triggers the command* c_2 *where memories are copied over.*

Proposition 3. *Given* ϕ *a deterministic formula written in guarded command form,* Γ *is a solution of* ϕ *if and only if there exists a unique guarded command* $g \implies c$ *of* ϕ *such that:*

$$\Gamma \models g \wedge c$$

Proof. We assume $\phi = \bigwedge_i (g_i \implies c_i)$ of size $n \in \mathbb{N}$ where g_i are guards and c_i are commands for all $1 \leq i \leq n$.

$$\begin{aligned} \Gamma \models \phi \text{ iff } \Gamma \models \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_i g_i) \\ \text{iff } \forall\ 1 \leq i \leq n,\ \Gamma \models g_i \implies c_i \text{ and } \Gamma \models \bigvee_{j \leq n} g_j \end{aligned}$$

Because guards are exclusive to each others (ϕ is a deterministic formula), if there exists $1 \leq i \leq n$ such that $\Gamma \models g_i$, then $\Gamma \not\models g_j$ for all $j \neq i$. Therefore, there exists a unique $1 \leq j \leq n$ such that $\Gamma \models g_j$.

$$\begin{aligned} \Gamma \models \phi \text{ iff } & \text{ for all } 1 \leq i \leq n,\ \Gamma \models g_i \implies c_i \text{ and} \\ & \text{there exists a unique } 1 \leq j \leq n,\ \Gamma \models g_j \end{aligned}$$

Since there exists a unique guard that Γ can satisfy, for all $1 \leq i \leq n$ such that $i \neq j$, $\Gamma \models g_i \implies c_i$ since $\Gamma \models \neg g_i$. Thus:

$$\begin{aligned} \Gamma \models \phi \text{ iff } & \text{ there exists a unique } 1 \leq j \leq n,\ \Gamma \models g_j \\ & \text{and } \Gamma \models g_j \implies c_j \end{aligned}$$

Which is equivalent to:

$$\Gamma \models \phi \text{ iff } \text{ there exists a unique } 1 \leq j \leq n,\ \Gamma \models g_j \text{ and } \Gamma \models c_j$$

□

3.2.2 Behavior of connectors

Operational semantics. We present in this section the operational semantics of constraint ϕ specifying a protocol as a labeled transition system, where we consider memory assignment as labels for *states*, and port assignments as labels for *transitions* between states. We call Γ_{V_ϕ} the set of assignment functions $\Gamma : P \cup M \cup M' \to D_*$ that satisfy the constraint ϕ. Given an assignment $\Gamma \in \Gamma_{V_\phi}$, we denote by Γ_P, Γ_M, and $\Gamma_{M'}$ the restrictions of Γ to respectively the port variables, un-primed memory variables, and primed memory variables assignment.

The operational semantics of a connector characterized by an internal constraint ϕ, where $V_\phi = P \cup M \cup M'$, is defined in terms of a labeled transition system $(S, L, s_0, \Gamma_{V_\phi}, \to)$, where:

- S the set of states
- s_0 is the initial state
- $L : S \to (M \to D_*)$ is a labeling function
- $\Gamma_{V_\phi} = \{\Gamma \mid \Gamma \models \phi\}$ is the set of solutions of ϕ
- $\to \subseteq S \times (P \to D_*) \times S$ the transition relation s.t.: $(s_0, \Gamma_P, s_1) \in \to$ iff there exists $\Delta \in \Gamma_{V_\phi}$ such that $\Delta_P = \Gamma_P$, $\Delta_M = L(s_0)$, and $\Delta_{M'} = L(s_1)$.

According to this definition, each state in $s_i \in S$ represents a data assignment for memories of a component (free variables in ϕ) at an instant i.

In the case where $M = \emptyset$, which means no memories are defined in the protocol, then the set of states S is composed only of one state s_0, such that $L(s_0) = \emptyset$.

By Proposition 3, the label transition system of a deterministic constraint is deterministic with respect to its label: for any state $s \in S$ and pair of distinct states $s', s'' \in S$, if $(s, \Gamma, s') \in \to$ and $(s, \Gamma', s'') \in \to$, then $\Gamma \neq \Gamma'$. Indeed, the two solutions Γ and Γ' must satisfy two different guarded commands, and since ϕ is deterministic, the two guards cannot be satisfied at the same time. Then, Γ and Γ' must differ in

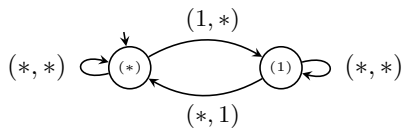


Figure 3.2: LTS of a *fifo1* channel with domain $D = \{*, 1\}$

at least one port assignment. It makes sense therefore to use such a label transition system to define the operational semantics of a connector specified by a deterministic constraint, as the resulting LTS is deterministic.

Example 45. *This example prolongs the Example 43, and use the data domain* $D = \{1, *\}$. *States are labeled by memory assignment. A fifo1 has only one memory, therefore a state labeled by the element* (d) *represents the assignment of value* d *to the memory. Transitions are labeled by port assignment. A fifo1 has two ports in its interface, therefore we express the assignment* a *maps to* d_a *and* b *maps to* d_b *as the tuple* (d_a, d_b). *Initially, the fifo1 start with an empty memory.*

The set Γ_{V_ϕ} *for the fifo1 channel corresponds to all solution of the constraint* $\mathsf{fifo1}(a, b)$, *that could be listed in the same manner as explained in Example 43. The resulting LTS for a fifo1 channel is shown in Figure 3.2.*

Execution path We define an infinite execution path σ of a transition system LTS as a sequence of transitions, i.e. $\sigma = s_0, \Gamma_{P0}, s_1, \Gamma_{P1}, s_2, ..., s_i, \Gamma_{Pi}, s_{i+1}, ...,$ where $(s_i, \Gamma_{Pi}, s_{i+1}) \in \rightarrow$.

We denote by w_σ the word induced by the path σ consisting of the sequence of the assignments Γ, i.e. $w_\sigma = (\Gamma_i)_{i \in \mathbb{N}}$ where $\Gamma_i(p) = \Gamma_{Pi}(p)$ and $\Gamma_i(m) = L(s_i)(m)$ for all $p \in P$, and $m \in M$ where L is the labeling function of the LTS. We write $\mathcal{T}$ for the set of infinite words, and $w \in \mathcal{T}$ is accepted if there exists an infinite execution path σ of LTS such that $w = w_\sigma$. Given $w \in \mathcal{T}$, the element $w(i)$ designates the i-th port and memory assignment in w, and $w(i)(v)$ gives the specific assignment for the port variable if $v \in P$ or memory variable if $v \in M$. The n-th derivation of a word w is noted as $w^{(n)}$ and defined such that $w^{(n)}(i) = w(n + i)$ for all $i \in \mathbb{N}$. Based on this definition, we have $w^{(0)}(i) = w(i)$. For example, in the first table (from the left) in Table 3.2, we have: $w^{(0)}(0) = (*, *, *)$, $w^{(0)}(1) = (1, *, *)$, and $w^{(1)}(0) = (1, *, *)$, $w^{(1)}(1) = (*, 1, *)$, where the values of the first, the second, and the third elements in the tuples are associated respectively to the ports and memory, a, m, and b.

Behavior of components We define the behavior of a component whose specification is given by a formula ϕ as the set of infinite words accepted by the LTS, i.e.:

$$L_\phi = \bigcup_{w \in \mathcal{T}} \{w \mid w \textit{ is an accepted word } \}$$

We refer to the behavior of a port as the restriction of the behavior of a component to a single variable. We note $w_{|a}$ the restriction of the word w to the port variable $a \in P$. The restriction $w_{|a}$ thus denotes the stream of values observed at port a, and is such that $w_{|a}(i) = w(i)(a)$ for all $i \in \mathbb{N}$.

Example 46. *The example of an execution of the protocol corresponding to the connector fifo1 is represented by Table 3.2.*

a	m	b
$*$	$*$	$*$
1	$*$	$*$
$*$	1	$*$
$*$	1	1
...	...	...

a	m	b
1	$*$	$*$
$*$	1	$*$
$*$	1	$*$
$*$	1	1
...	...	...

a	m	b
1	$*$	$*$
$*$	1	1
1	$*$	$*$
$*$	1	1
...	...	...

Table 3.2: Three words in the behavior set of a *fifo1* channel with domain $D = \{1, *\}$

We use the same convention as in Section 2.3, namely that a behavior of a order sensitive component is described without time labels. Moreover, using the set notation for an assignment Γ of ports to values, a word labeling an LTS is a sequence of sets of assignments, which denotes the representative of a set of equivalent behaviors under stretching. Thus, the semantics of a constraint ϕ is given as a component (E_ϕ, L_ϕ) where E_ϕ contains all possible assignments for all free ports and memories occurring in ϕ.

3.3 Verification of temporal properties on connectors

To specify the properties of the executions of Reo protocols, we define in t this section the LTL formulas semantics on Reo connector behaviors.

Let C be a Reo protocol specified with the formula ϕ, such that its operational semantics is specified with LTS. Let σ be an execution path of LTS. We refer by w_σ a word accepted by LTS over σ, and denote $\mathcal{T}$ the set of accepted words.

An LTL formula is expressed using the following syntax:

$$\varphi ::= \ v = d \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathsf{X}\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi_1 \mathsf{U} \varphi_2$$

where v is a port or memory variable in $P \cup M$ and d is an element of the data domain D_*.

We interpret the LTL formulas over the accepted word w_σ and define the satisfaction relation $\models$ such as,

- $w_\sigma \models v = d$ iff $w_\sigma^{(0)}(0)(v) = d$
- $w_\sigma \models \varphi_1 \wedge \varphi_2$ if and only if $w_\sigma \models \varphi_1$ and $w_\sigma \models \varphi_2$
- $w_\sigma \models \neg\varphi$ if and only if $w_\sigma \not\models \varphi$
- $w_\sigma \models \mathsf{X}\phi$ if and only if $w_\sigma^{(1)} \models \phi$
- $w_\sigma \models \Box\varphi$ if and only if $w_\sigma^{(i)} \models \varphi$ for all $i \in \mathbb{N}$
- $w_\sigma \models \Diamond\varphi$ if and only if $w_\sigma^{(i)} \models \varphi$ for some $i \in \mathbb{N}$
- $w_\sigma \models \varphi_1 \mathsf{U} \varphi_2$ if and only if there exists $j \in \mathbb{N}$ such that $w_\sigma^{(j)} \models \varphi_2$, and $w_\sigma^{(i)} \models \varphi_1$ for all $0 \leq i < j$.

We introduce several properties of interest on Reo circuit. Given p, m, and m' respectively port, memory, and next memory variables, we denote by $\mathbf{p}$ and $\mathbf{m}$ the atomic propositions $p \neq *$ and $m' \neq m$ which represent that p is firing and m is changing value. In both cases, we say that p or m fires.

Properties	Temporal formulas
a fires	$\mathbf{a}$
a silent	$\neg\mathbf{a}$
a before b	$\mathbf{a} \implies \Diamond\mathbf{b}$
a then b	$\mathbf{a} \implies \mathsf{X}(\ \mathit{silent}\ \mathsf{U}\ \mathbf{b})$
a sync b	$\Box(\mathbf{a} \iff \mathbf{b})$
a async b	$\Box((\mathbf{a} \implies \Diamond\mathbf{b}) \vee (\mathbf{b} \implies \Diamond\mathbf{a}))$

Example 47. *As an example of LTL formula for a sync channel, we have* $\Box(\mathbf{a} \iff \mathbf{b})$ *where* a *and* b *are port variables. For a fifo1 channel, we have* $\Box(\mathbf{a} \implies \mathsf{X}(\neg\mathbf{a}\mathsf{U}\mathbf{b}))$ *and also* $\Box(\mathbf{b} \implies \mathsf{X}(\neg\mathbf{b}\mathsf{U}\mathbf{a}))$.

We now look into the implementation and verification of a protocol described by a formula in guarded command form. As Proposition 3 shows each solution is described by a unique guarded command. Therefore a program implementing a protocol checks the set of guards that are satisfied by the state of the environment and the internal state, and nondeterministically satisfies one and only one corresponding command. We develop in the next section the steps leading from a protocol specification to a program written in Promela.

3.3.1 From synchronous protocol to asynchronous implementation

In the previous section, we detailed the requirement and the procedure to write a protocol as a formula in the guarded command form. The implication of having such a form for a protocol makes it possible and easier to translate it into a program. In this section, we define a translation from a formula written as a guarded command to a Promela program. We show the correctness of the translation, by comparing the semantic of a Reo specification, and that of its target program in Promela.

Translation of Reo to Promela Throughout this section, we assume a generic data type denoted as Data for data flowing in the protocol, since data type is specific in the application that employs the protocol. We go through the main constructs in Reo, being ports, components, and connectors.

Ports In Reo, as defined in the previous section, a port is a location where two components synchronize and exchange data. In Promela, we implement a Reo port as a pair of two Promela channels, each with a buffer size of one. We show that our Promela implementation of a port simulates Reo's synchronized message passing between components.

```
typedef port {
        chan data = [1] of {Data};
        chan trig = [1] of {int}; }
```

Listing 3.1: definition of a Reo port in Promela

As expressed in Listing 3.1, a port has a data channel and a synchronization channel. The data channel is responsible of the data flow between input and output ends of a port. The Promela synch channel ensures synchronous exchange between these two ends.

As described in Listing 3.2, two actions can be performed on such a constructed port: put and take.

```
inline put(q,a) {
  int x;
  q.data!a;
  q.trig?x }

inline take(q,a) {
  q.trig!-1; q.data?a }
```

Listing 3.2: put and take functions

The action put has two arguments: a port q and a datum a. The function call put(q, a) atomically fills the data channel of q with the datum a, and blocks on the trig channel, waiting to synchronize with the component on the output side of q. The integer x is used to empty the trig channel, but its value does not matter.

The action take has a port q and a variable a as arguments. The function call take(q, a) atomically notifies, by filling the trig channel, that there is a component willing to take data, and blocks on the data channel, until a datum can be taken into the variable a. The integer value of -1 written into the synchronization channel is arbitrary, as trig is used only for signaling.

We describe two temporal properties that reflect the synchronous behavior of a port in an asynchronous implementation. We say that a port fires whenever a data is exchanged between the input party and the output party. If a port does not fire, it is silent. In the case of an implementation of a port with two buffers, the firing property occurs whenever a port has both an input and an output request: both buffers are full. We say the a buffer (or a memory) is full whenever it contains a data, and is empty otherwise. We define some macros in Listing 3.3 for firing and silent property of port p, and for full and empty buffer.

Listing 3.3: Macros for firing of ports

```
#define p_fires (
        !(len(p.data) == 0) && !(len(p.trig) == 0) &&
        X((len(p.data) == 0) || (len(p.trig) == 0)))

#define p_silent        (! p_fires)


#define m_full          (!(len(m)==0))
#define m_empty         ((len(m)==0))
```

Components We call an external component that interacts with the main protocol, an *agent*. For each port q in the protocol's interface, we assume an *agent* connected to that port. More precisely, if q is used as an input port by the protocol, the *agent* connected to q must use q as an output port, and vice versa. We give in Listing 3.4 an example of a definition of an agent with two ports as its arguments, one input and one output.

```
proctype agent(port p1; port p2){
        /*      p1: input, p2: output   */
        do
          :: /* action */
        od }
```

Listing 3.4: A generic structure for an agent

Each agent is defined as a proctype in Promela, and runs concurrently with the main protocol. We represent the generic behavior of an agent as an infinite sequence of non deterministic actions (with the do − od loop), but the definition of the precise behavior of an agent is left for the user. Since agents and protocol share ports, it is possible that an agent blocks until a datum is delivered at its port.

We assume a set of agents given by the user. Our compiler generates only the skeleton of an agent including the set of ports in its interface, with the direction of each port (either input or output). As an extension, we intend to make the input/output restriction of ports direction more strict, using the Promela assertion xr and xs to prevent misuse of port directionality. We later specify some properties of the desired observable behavior.

Connectors As introduced previously, the main difference between a component and a connector is the ability for the component to block on some put or get operations on its port. We showed in the previous section a guarded command transformation for a connector, where the guard plays the role of a safety check on the state of the input and output ports, and the command gives the values exchanged at the output ports in terms of the value taken at the input ports.

A connector is a process running concurrently to the components. We define a connector as a proctype, taking as argument all ports in the interface of the connectors. The internal operation in the connector proctype are defined based on the definition of its internal constraint. We call P the set of free port variables used in the connector's constraint, and M the set of memory variables.

We first instantiate every variable occurring free in the connector's internal con-

straint as a channel structure in Promela. For every port variable $p \in P$, we define a global instance of the port structure defined in previous paragraph, with the same name as the variable. Since variable have unique names, the structure is also unique for every ports. The structure for a port is global, and will later be used for checking some temporal properties. For every memory variable $m \in M$, we define a local channel of size 1. Constant symbols are mapped to their corresponding domain. Promela only supports few data types, we assume that the constants get an interpretation in one of those data type (integer, boolean, float, characters). Function symbols are mapped to inline procedures in Promela. For each function used in Reo, we assume that an inline procedure with the same name will be provided in Promela. We use, for every port or memory variables, an additional variable in Promela that will temporary store the data occurring at a port or memory. This additional variable is typically used when multiple output variables take the value of a single input variable: in this case, the value of the input variable is temporary stored, and replicated to every outputs.

Based on the result of Proposition 1 we take the connector in its guarded command form. We use $\mathcal{GC}$ for the set of guarded commands. A guarded command $g \to c \in \mathcal{GC}$ has a natural interpretation in Promela as a conditional update. The guard g is translated as a condition on the status of the port and memory channels, while the command c is an update of the port and memory status. The most common statements in the guards are `full(p.data)`, `full(p.trig)` and `full(m)`, which respectively checks whether a port `p` has an incoming data request, an outgoing data request, or if the memory `m` is full. The negation of those statement are also commonly used in the guards. In the command, we proceed for the update of ports and memories, which corresponds to statements of the kind `take(p,d)`, `put(p,d)`, `m?d`, or `m!d`, where `d` is a local variable.

The generic structure of a Promela program obtained from a deterministic formula with n guarded commands is shown in Listing 3.5.

```
proctype Protocol(port p1;...){
  /*    p1:     input, ...      */
  /* Memory declaration */
  chan m = [1] of {int}; ...
  /* Initial state */
  m!0; ...
  /* Local variables */
  int _m;  int _p1 ;  ...
  /* Guarded commands */
  do
  :: (guard_1) ->  command_1
  :: ...
  :: (guard_n) ->  command_n
  od }
```

Listing 3.5: a generic structure of a protocol

Note that the statements in the command are executed sequentially. We show in the next section that we can express some synchronous patterns as an LTL property on the state of the ports and memories.

LTL properties We give a translation to an LTL property on the Promela translation, such that the properties of synchrony are preserved. Therefore, we show that if the LTL property should hold on the Reo circuit, then the corresponding asynchronous LTL property should hold on the Promela program generated from the Reo circuit.

The property `p_silent` is defined as the negation of the firing property, and represents all the non firing states of the port p. The property `silent` denotes the conjunction of all silent properties for all ports. Note that internal memory updates are still allowed.

The two properties `p1_before_p2` and `p1_then_p2` express some asynchronous firing for ports p_1 and p_2. The latter property is stricter, since the `silent` property requires that between the firing of ports p_1 and p_2, no other ports fire: it is true that `p1_then_p2` implies `p1_before_p2`.

We define the synchronous property in Promela as a binary relation between two ports p_1 and p_2. We say that the two ports are synchronous if it is always the case that p_1 fires and then p_2 fires (or the opposite), and all steps in between the firings satisfy the `silent` property. Synchronous property is a stricter form of asynchronous property: it is true that `sync_p1_p2` implies `async_p1_p2`.

Properties	Temporal formulas
p_fires	len(p.data) != 0 && len(p.trig)!=0 && X(len(p.data)==0 || len(p.trig)==0)
p_silent	!(p_fires)
m_full	len(m.data)!=0
m_empty	len(m.data)==0
m_fires	m_full && X(m_empty)) || m_empty && X(m_full)
m_silent	!(m_fires)
p1_before_p2	(p1_fires -> <> (p2_fires))
p1_then_p2	(p1_fires -> X(silent U p2_fires))
sync_p1_p2	[] (p1_then_p2 \/ p2_then_p1)
async_p1_p2	[] (p1_before_p2 \/ p2_before_p1)

Arguments for correctness In this section, we show that the sequence of messages exchanged between the generated Promela processes can be related to the sequence of data exchanges at a port of a Reo circuit. We first establish, based on the structure of the generated code resembling the guarded command form of the connector, that every data exchanged between Promela processes can be related to an assignment that satisfies the connector. As a consequence, every sequences of data exchanges between Promela processes can be related to a sequence of assignments in the behavior of a connector. We then show that there exists, for any sequences in the behavior of a Reo connector in its ideal environment, a set of processes in Promela such that the message exchanges between Promela processes correspond to the sequence in the behavior of the Reo connector. For simplicity and clarity, we consider a binary data domain for ports. The arguments for a larger domain are similar.

We use the semantic of Promela defined in [80] to show the correctness of our translation. The operational semantics of a Promela program $\mathcal{P}$ composed of processes P_i is defined as a graph $T = (Q, \rightarrow, q_0)$ where Q is a set of states and $\rightarrow$ is a binary relation on states. A state $q \in Q$ is a tuple $q = (l_0, ..., l_m, lv1, ..., lvm, gv)$ where each l_i is a location in process P_i, lvi is the vector of local variable values in process P_i, and gv is the vector of global variables in P. The state $q_0 \in Q$ denotes the initial state. We write $l_i \xrightarrow{st} l_i'$ if and only if there is a statement st from l_i to l_i'. The variables after executing st are $st(lv)$ and $st(gv)$. In our case, st is an assignment, `skip`, or

conditional statement; or it is an asynchronous send (resp. receive) from a non-full (resp. non-empty) channel. The initial state of a Promela program $T = (Q, \rightarrow, q_0)$ is $q_0 = (l^0, lv^{init}, gv^{init})$ and a *path* of T is a sequence of state $q_0 q_1 ...$ such that $(q_i, q_{i+1}) \in \rightarrow$.

A Promela program produced by a Reo compiler consists of a set of N processes running concurrently, where $N-1$ processes are agents interacting through the protocol process. Agents share ports with the protocol, such that they can *put* values on the input port of the protocol, and *take* values from the output ports of the protocol.

By construction, the connector in Promela can only change its internal variables (memories) and the global vector of variables (boundary ports) if one of its guard is true and the command is performed. Guards are boolean statements checking whether the `trig` and `data` channels of a port are full (or empty), or if the memories channels are full (or empty). Given a connector with n ports at its interface, and k internal memories, there are 2^{2n+k} possible states in the graph of the Promela process.

We show soundness of our implementation with the following arguments:

1. In the initial state, the vector gv is set to its initial value, together with all memory channels.

2. Given a vector gv of global variable, and a vector lv of local variables, the process for the connector, if scheduled, evaluate its guards. From the set of guards satisfied, one is selected non-deterministically, and the corresponding command is performed. The guard ensure that the `take` and `put` statements in the command will not block. The value exchanged at the ports, the current and next values for the memory constitute an assignment that satisfies a unique guarded command in the deterministic formula of the connector (Proposition 3).

3. If no guards are satisfied, processes for agents are scheduled, modifying the global vector of variables gv.

All sequences of assignments allowed by the connector process are included in the sequences of assignments satisfying the formula of the connector.

Completeness can be derived by showing that every solution of the guarded command corresponds to an implementation where the set of agents simulate the environment of the solution. By taking an implementation that unifies the agents, and using the non-deterministic properties of Promela, we can simulate any solution of the formula as a statement and a gv vector in Promela. Elaboration of this part remains as future work.

3.3.2 Case Study

In this section, we present an application of our approach. We show the Promela specification of a *fifo* channel, and study its LTL properties in two contexts: ideal and constrained environment. A second example is available in [58], in which we analyse a composite connector representing the protocol involved in a railway system. We show the Promela program compiled from the corresponding Reo circuit, and verify some properties of interest using SPIN.

We refer an on-line repository [59] for reproduction of the results presented in this section. A binary for the compiler, together with some properties of interest discussed in the section are accessible.

Study on the *fifo* channel We consider the Reo specification of *fifo* described in Example 42, where ports a and b are renamed to ports $p1$ and $p2$. We connect the channel with two components: one that produces data at the input port $p1$ and one that consumes that at the output port $p2$.

We study the properties of a *fifo* channel in two different environments:

- The ideal environment: the two components connected by the *fifo* channel, producer and consumer, behave in the ideal way. The producer is willing to produce messages infinitely often, and the consumer is willing to consume messages infinitely often.
- The constrained environment: the producer is willing to produce messages infinitely often, but the consumer consumes a finite number of messages only. We show in this case that some properties of the *fifo* channel that were satisfied in the ideal environment may now be violated due to some non ideal behavior of boundary components.

Before detailing the verification of LTL properties in these two cases, we present the Promela implementation of *fifo* channel, described in Listing 3.6. The Reo protocol is implemented by the process `Protocol`, and the producer and consumer are implemented respectively by the processes `Prod` and `Cons`.

```
proctype Protocol(port p1;port p2 ){
bit _m = 0 ;
do
:: (empty(m) && full(p1.data)) ->  take(p1,_m); m!_m
:: (full(m)  && full(p2.trig)) ->  m?_m; put(p2,_m)
od
```

```
}

init{
run Prod(p1); run Cons(p2); run Protocol(p1,p2); }
```

Listing 3.6: Promela implementation of a *fifo* channel

The Promela implementations of the components producer and consumer are described in Listing 3.7.

```
proctype Prod(port a){
  do
  :: atomic{put(a,1)}
  od
}
proctype Cons(port a){
bit y;
  do
  :: atomic{take(a,y)}
  od
}
```

Listing 3.7: Promela implementation of a consumer and producer

Verification in the ideal environment. We present three properties to verify on *fifo1* connector:

- $prop_1 \equiv \Box(\mathbf{p1} \implies \Diamond(\mathbf{p2}))$, which states that always if the port $p1$ fires then eventually the port $p2$ fires. This property is verified. It is implemented in Promela as follows:

```
ltl prop1 {[] ( p1_fires -> <> p2_fires ) }
```

- $prop_2 \equiv \Box(m \neq * \implies \Diamond(\mathbf{p2}))$, which states that always if the buffer m is full then eventually the port $p2$ fires. This property is verified. It is implemented in Promela as follows:

```
ltl prop2 {[] ( m_full -> <> p2_fires ) }
```

- $prop_3 \equiv \Box(m \neq * \implies \mathsf{X}(\mathbf{p2}))$, which states that always if m is full then in the next state $p2$ fires. This property is not verified because the port $p2$ becomes silent in the next state. It is implemented in Promela as follows:

```
ltl prop3 {[] ( m_full -> X p2_fires ) }
```

- $prop_4 \equiv \Box(\mathbf{p1} \implies \mathsf{X}(\neg \mathbf{p1} \ \mathsf{U} \ \mathbf{p2}))$, which states that always if $p1$ fires then in the next state $p1$ becomes silent until $p2$ fires. This property is verified. It is implemented in Promela as follows:

```
ltl prop4 {[](p1_fires -> X( p1_silent U p2_fires))}
```

Remark 11. *The result of the property verification, described above, shows that the implementation of the fifo1 connector is correct. Indeed, based on the behavior of fifo1, this result is the one we expected.*

Verification in the constrained environment: In this case, the producer interacts with a finite consumer, so the number of messages that could be consumed by the is limited to 5. In Listing 3.8, is presented the Promela implementation of a finite consumer, specified by the process $consFinite()$.

We verified the properties described above, $prop1$, $prop2$, $prop3$, and $prop4$, and as expected, all these properties are not satisfied. Indeed, the properties $prop1$, $prop2$, and $prop4$ are not satisfied because of the behavior of the component consumer that prevents firing the port $p2$ after consuming the first 5 messages. Therefore their violation is due to the protocol environment. However the property $prop3$, remains not satisfied as in the case of the ideal environment.

```
proctype consFinite(port a){
bit y;
int i = 5;
do
:: i>0; atomic{take(a,y)}; i = i-1
:: break
od
}
```

Listing 3.8: a generic structure of a protocol

3.4 Related work and future work

Model checker Vereofy [16, 15, 14] is a model checking tool developed at the University of Dresden to analyze and verify Reo connectors. Vereofy has two input languages: the Reo Scripting Language (RSL), used to specify the coordination protocol, and a guarded command language called Constraint Automata Reactive Module Language (CARML), a textual version of constraint automata used to specify the behavior of

components. Vereofy allows the verification of temporal properties expressed in LTL and CTL-like logics.

Our work differs from Vereofy, since we use the Treo [31] (Textual Reo language) to describe both the protocol and the boundary components. In Treo, the description of the behavior of primitive channels and components is parametric: the user has to define a semantic domain, and the Reo composition operation in that semantics. We make use of the rule based semantics [32] for channels and give the description of boundary components directly as Promela processes. Our work extended the set of backend for the compiler so that the Textual Reo description (i.e., Treo input file) compiles to a Promela program that can be used by the Spin model checker.

Denotational semantics for Reo In [3], the authors give a co-inductive semantic model for Reo connectors, based on timed-data streams (TDS). We shall briefly highlight the main differences between such model and ours. First, the TES model for component behavior explicitly captures atomic set of events in an observation, while the TDS model implicitly represent atomicity as all firing of ports that occur at the same time. The time stamp of a TES can therefore be dropped while preserving the information of atomicity: this construction simplifies reasoning about atomicity.

Another difference between TES and TDS is that TESs are not restricted to communication ports, but can model arbitrary events. Moreover, the implicit semantics in TDS that a port fires infinitely often (which is used as an argument for fairness in [3]) is no longer assumed in the TES model, but can still be recovered if needed.

Operational semantics for Reo The constraint automata semantics for Reo was also considered in [17] for defining and verifying bisimulation and language equivalence between Reo connectors. In [6, 9], the authors considered time constraints, and proposed a timed version of constraint automaton to verify by model checking timed CTL properties. In [47, 46], the authors use timed constraint automata and present a SAT-based approach for bounded model checking of real-time component connectors. Another verification approach based on SAT solvers was proposed in [11], exploiting Alloy. This work allows analysis of Reo circuits and verification of their properties expressed as predicates in the lightweight modeling language of Alloy, which is based on first-order relational logic. In [50, 49], Kokash et al. proposed a framework for the verification of Reo circuits using the mCRL2 toolset (developed at the TU of Eindhoven). Their tool automatically generates mCRL2 specifications from Reo graphical models. The translation from Reo to mCRL2 uses the constraint automata semantics of Reo.

Chapter 4

Operational specifications of components

In Chapter 2, we presented a model of components that captures timed-event sequences (TESs) as instances of their behavior. An observation is a set of events with a unique time stamp. A component has an interface that defines which events are observable, and a behavior that denotes all possible sequences of its observations (i.e., a set of TESs). Our component model is equipped with a family of operators parametrized with an interaction signature. Thus, cyber-physical systems are defined modularly, where the product of two components models the interaction occurring between the two components. The strength, as well as practical limitation, of our semantic model is its abstraction: there is no fixed machine specification that generates the behavior of a component. We give, in this chapter, three operational descriptions of components each at different level of abstraction.

As a first operational specification, we present a state-based description of a component's behavior using labeled transition systems. A *TES transition system* has transitions labeled with observations, and enriches components with states. Different TES transition systems may denote the same component, as a component is oblivious to internal non-determinism of the machinery that manifests its behavior. As for components, we introduce a family of parametrized algebraic products on TES transition systems. The parameter here is a composability relation on observations, and each transition in the product is the result of the composition of a pair of transitions with composable labels. We show that the TES transition system component semantics is

compositional with respect to such products, i.e., the component resulting from the product of two TES transition systems is equal to the product of the components resulting from each TES transition system. On components, the composability relation is co-inductively lifted from observations to TESs, and the composition function is set union on observations and interleaving on streams.

Because the composability relation on observations is a step-wise operation, it may lead to deadlock states in the product TES transition system, i.e., states with no outgoing transitions. We call two TES transition systems *compatible* with respect to a composability relation on observations if, for every reachable pair of states, there is at least one pair of transitions whose pair of labels is composable. We give some sufficient conditions for TES transition systems to be compatible, and show that if two TES transition systems are compatible, then their product can be done lazily, i.e., step by step at runtime. Note that, however, a TES transition system may not be executable, e.g., have countably infinite states or transitions.

As a second operational specification, we introduce a finitely representable description of components in a rewriting logic specification. Rewriting logic is a powerful framework to model concurrent systems [64, 63]. Moreover, implementations, such as Maude [29], make system specifications both executable and analyzable. Rewriting logic is suitable for specifying cyber-physical systems, as its underlying equational theory can represent both discrete and continuous changes. We give an operational specification for components as rewriting systems, and show its compositionality under some assumptions.

Our rewriting specification has the following benefits. First, performing lazy composition keeps the representation of an interacting system small. Second, step-wise runtime composition renders our runtime framework modular, where run-time replacement of individual components becomes possible (as long as the update complies with some rules). Finally, the runtime framework more closely matches the architecture of a distributed framework, where entities are physically separated and no party may have access to the whole description of the entire system. A Maude implementation of our framework is provided in Chapter 5, with a series of detailed examples. The rewriting specification of components, however, does not offer mechanisms to resolve non-determinism at the component or the system levels: multiple transitions may be allowed, and one is selected non-deterministically.

As a third operational specification, we introduce an algebra of weighted automata whose transition values model an internal strategy for an agent. The preference structure is compositional, which allows for reasoning about local choices of an agent in

a state, or global choices of a system of agents given the product of their respective transition values.

As an example application, we use a subset of the Reo language as a domain specific language to graphically specify preference aware agents in interaction.

4.1 Components as transition systems

In Section 2.1, we give a declarative specification of components, and considers infinite behaviors only. We give, in Section 4.1.1, an operational specification of components using TES transition systems. We relate the parametrized product of TES transition systems with the parametrized product on their corresponding components, and show its correctness. The composition of two TES transition systems may lead to transitions that are not composable, and ultimately to a deadlock, i.e., a state with no outgoing transitions.

Notation Given $\sigma : \mathbb{N} \to \Sigma$, let $\sigma[n] \in \Sigma^n$ be the finite prefix of size n of σ and let $\sim n$ be an equivalence relation on $(N \to \Sigma) \times (N \to \Sigma)$ such that $\sigma \sim_n \tau$ if and only if $\sigma[n] = \tau[n]$. Let $FG(L)$ be the set of left factors of a set $L \subseteq \Sigma^\omega$, defined as $FG(L) = \{\sigma[n] \mid n \in \mathbb{N}, \sigma \in L\}$. We write $\sigma(n)$ for the n-th derivative of σ, i.e., the stream such that $\sigma(n)(i) = \sigma(n + i)$ for all $i \in \mathbb{N}$.

4.1.1 TES transition systems.

The behavior of a component as in Definition 1 is a set of TESs. We give an operational definition of such set using a labelled transition system.

Definition 30 (TES transition system)**.** *A TES transition system is a triple* $(Q, E, \to)$ *where* Q *is a set of states,* E *is a set of events, and* $\to \subseteq Q \times (\mathcal{P}(E) \times \mathbb{R}_+) \times Q$ *is a labeled transition relation, where labels are observations.*

We present two different ways to give a semantics to a TES transition system: inductive and co-inductive. Both definitions give the same behavior, as shown in Theorem 6, and we use interchangeably each definition to simplify the proofs of, e.g., Theorem 7.

Semantics 1 (runs). Let $T = (Q, E, \rightarrow)$ be a TES transition system. We write $q \xrightarrow{u} p$ for the sequence of transitions $q \xrightarrow{u(0)} q_1 \xrightarrow{u(1)} q_2 ... \xrightarrow{u(n-1)} p$, where $u = \langle u(0), ..., u(n-1)\rangle \in (\mathcal{P}(E) \times \mathbb{R}_+)^n$. We write $|u|$ for the size of the sequence u. We use $\mathcal{L}^{\text{fin}}(T, q)$ to denote the set of finite sequences of observables labeling a finite path in T starting from state q, such that

$$\mathcal{L}^{\text{fin}}(T, q) = \{u \mid \exists q'.q \xrightarrow{u} q', \forall i < |u| - 1.u(i) = (O_i, t_i) \wedge t_i < t_{i+1}\}$$

Additionally, the set $\mathcal{L}^{\text{fin}*}(T, q)$ is the set of sequences from $\mathcal{L}^{\text{fin}}(T, q)$ postfixed with empty observations, i.e., the set

$$\mathcal{L}^{\text{fin}*}(T, q) = \{u\tau \in TES(E) \mid u \in \mathcal{L}^{\text{fin}}(T, q) \text{ and } \tau \in TES(\emptyset)\}$$

We use $\mathcal{L}^{\text{inf}}(T, q)$ to denote the set of TESs labeling infinite paths in T starting from state q, such that

$$\mathcal{L}^{\text{inf}}(T, q) = \{\sigma \in TES(E) \mid \forall n.\sigma[n] \in \mathcal{L}^{\text{fin}}(T, q)\}$$

where $\sigma[n]$ is the prefix of size n of σ. The semantics of such a TES transition system $T = (Q, E, \rightarrow)$, starting in a state $q \in Q$, is the component $C_T(q) = (E, \mathcal{L}^{\text{inf}}(T, q))$.

Let $X \subseteq TES(E)$, we use $cl(X)$ to denote the set that contains the continuation with empty observations of any prefix of an element in X, i.e., $cl(X) = \{u\tau \in TES(E) \mid \tau \in TES(\emptyset) \text{ and } \exists\sigma.\exists i.\sigma \in X \wedge \sigma[i] = u\}$. Given a component $C = (E, L)$, we write $cl(C)$ for the new component $(E, cl(L))$.

Semantics 2 (greatest post fixed point) Alternatively, the semantics of a TES transition system is the greatest post fixed point of a function over sets of TESs paired with a state. For a TES transition system $T = (Q, E, \rightarrow)$, let $\mathcal{R} \subseteq TES(E) \times Q$. We introduce $\phi_T : \mathcal{P}(TES(E) \times Q) \rightarrow \mathcal{P}(TES(E) \times Q)$ as the function:

$$\phi_T(\mathcal{R}) = \{(\tau, q) \mid \quad \exists p \in Q,\ q \xrightarrow{\tau(0)} p \wedge (\tau', p) \in \mathcal{R}\}$$

We can show that ϕ_T is monotonous, and therefore ϕ_T has a greatest post fixed point $\Omega_T = \bigcup\{\mathcal{R} \mid \mathcal{R} \subseteq \phi_T(\mathcal{R})\}$. We write $\Omega_T(q) = \{\tau \mid (\tau, q) \in \Omega_T\}$ for any $q \in Q$. Note that the two semantics coincide.

Theorem 6 (Equivalence)**.** *For all $q \in Q$, $\mathcal{L}^{\text{inf}}(T, q) = \{\tau \mid (\tau, q) \in \Omega_T\}$.*

Proof. Let $T = (Q, E \rightarrow)$ and $\Omega_T(q) = \{\tau \mid (\tau, q) \in \Omega_T\}$.

$$\begin{aligned}
\tau \in \Omega_T(c_0) &\iff (\tau, c_0) \in \Omega_T \\
&\iff c_0 \xrightarrow{\tau(0)} c_1 \wedge (\tau', c_1) \in \Omega_T \\
&\iff \exists \chi \in \rightarrow^\omega \,.\, \chi(0) = c_0 \xrightarrow{\tau(0)} c_1 \wedge (\tau', c_1) \in \Omega_T \\
&\iff \exists \chi \in \rightarrow^\omega, c \in Q^\omega.\ c(0) \in Q_0 \wedge \forall n \in \mathbb{N}. \\
&\qquad \chi(n) = c(n) \xrightarrow{\tau(n)} c(n+1) \wedge (\tau^{(n+1)}, c(n+1)) \in \Omega_T \\
&\iff \exists \chi \in \rightarrow^\omega \,.\, \chi(0) = c_0 \xrightarrow{\tau(0)} c_1 \wedge \tau \in \mathcal{L}^{\inf}(T, q) \\
&\iff \tau \in \mathcal{L}^{\inf}(T, q)
\end{aligned}$$

In the fourth equivalence, we state that the infinite sequence $\chi \in \rightarrow^\omega$ has, as sequence of labels, the sequence of observations in τ. We prove the step by induction. Let $n \in \mathbb{N}$, and let $\chi \in \rightarrow^\omega$ be such that, for all $k \leq n$, $\chi(k) = c_k \xrightarrow{\tau(k)} c_{k+1}$ with $c_k \in Q$ and $(\tau^{(k)}, c_k) \in \Omega_T$. Then, given that $(\tau^{(n)}, c_n) \in \Omega_T$, there exists a transition $c_n \xrightarrow{\tau(n+1)} c_{n+1}$ and there exists $\rho \in \rightarrow^\omega$ such that, for all $k \leq n$, $\rho = \chi$ and $r(n+1) = c_n \xrightarrow{\tau(n+1)} c_{n+1}$, which proves the implication. The other direction of the equivalence is simpler. If there exists $\chi \in \rightarrow^\omega$ such that for all $n \in \mathbb{N}$, $\chi(n) = c_n \xrightarrow{\tau(n)} c_{n+1}$, then we have a witness, for every $n \in \mathbb{N}$, that $(\tau^{(n)}, c_n)$ is an element of Ω_T. □

Remark 12 (Deadlock)**.** *Observe that $FG(\mathcal{L}^{\inf}(T, q)) \subseteq \mathcal{L}^{\text{fin}}(T, q)$ which, in the case of strict inclusion, captures the fact that some states may have no outgoing transitions and therefore deadlock.*

Remark 13 (Abstraction)**.** *There may be two different TES transition systems T_1 and T_2 such that $\mathcal{L}^{\inf}(T_1) = \mathcal{L}^{\inf}(T_2)$, i.e., a set of TESs is not uniquely characterized by a TES transition system. In that sense, the TES representation of behaviors is more abstract than TES transition systems.*

Example 48. *The behavior of a robot introduced earlier is a TES transition system $T_R = (\mathbb{R}_+, E_R, \rightarrow)$ where $t \xrightarrow{(\{e\},t)} t + \delta$ for abitrary t and δ in $\mathbb{R}_+$ and $e \in E_R$.*

Similarly, the behavior of a grid is a TES transition system $T_G(I, n, m) = (Q_G, E_G(I, n, m), \rightarrow$) where:

- $Q_G \subseteq \mathbb{R}_+ \times (I \rightarrow ([0; n] \times [0; m]))$,
- $(t, f) \xrightarrow{(O,t)} (t + \delta, f')$ *for abitrary t and δ in $\mathbb{R}_+$, such that*
  - *$d_R \in O$ implies $f'(R)$ is updated according to the direction d if the resulting position is within the bounds of the grid;*

- $(x,y)_R \in O$ *implies* $f(R) = (x,y)_R$ *and* $f'(R) = f(R)$*;*
- $f'(R) = f(R)$*, otherwise.*

The behavior of a swap protocol $S(R_i,R_j)$ *with* $i < j$ *is a TES transition system* $T_S(R_1, R_2) = (Q, E, \rightarrow)$ *where, for* $t_1, t_2, t_3, t_4 \in \mathbb{R}_+$ *with* $t_1 < t_2 < t_3 < t_4$*:*

- $Q \subseteq \mathbb{R}_+ \times \{s_1, s_2, s_3, s_4, s_6\}$*;*
- $E = E_{R_i} \cup E_{R_j} \cup \{lock(R_i, R_j), unlock(R_i, R_j), start(R_i, R_j), end(R_i, R_j)\}$
- $(t_1, s_1) \xrightarrow{(\{lock(R_i,R_j)\},t_1)} (t_2, s_2)$*;*
- $(t_1, s_2) \xrightarrow{(\{unlock(R_i,R_j)\},t_1)} (t_2, s_1)$*;*
- $(t_1, s_1) \xrightarrow{(\{start(R_i,R_j),(x,y)_{R_i},(x+1,y)_{R_j}\},t_1)} (t_2, s_3)$*;*
- $(t_1, s_3) \xrightarrow{(\{N_{R_j}\},t_1)} (t_2, s_4) \xrightarrow{(\{\{W_{R_j},E_{R_i}\}\},t_2)} (t_3, s_5) \xrightarrow{(\{\{S_{R_j}\}\},t_3)} (t_4, s_6)$*;*
- $(t_1, s_6) \xrightarrow{(\{end(R_i,R_j)\},t_1)} (t_2, s_1)$*;*

■

The product of two components is parametrized by a composability relation κ on observations and syntactically constructs the product of two TES transition systems.

Definition 31 (Product)**.** *The product of two TES transition systems* $T_1 = (Q_1, E_1, \rightarrow_1)$ *and* $T_2 = (Q_2, E_2, \rightarrow_2)$ *under the constraint* κ *is the TES transition system* $T_1 \times_\kappa T_2 = (Q_1 \times Q_2, E_1 \cup E_2, \rightarrow)$ *such that:*

$$\frac{q_1 \xrightarrow{(O_1,t_1)}_1 q_1' \quad q_2 \xrightarrow{(O_2,t_2)}_2 q_2' \quad ((O_1,t_1),(O_2,t_2)) \in \kappa(E_1,E_2) \quad t_1 < t_2}{(q_1,q_2) \xrightarrow{(O_1,t_1)} (q_1',q_2)}$$

$$\frac{q_1 \xrightarrow{(O_1,t_1)}_1 q_1' \quad q_2 \xrightarrow{(O_2,t_2)}_2 q_2' \quad ((O_1,t_1),(O_2,t_2)) \in \kappa(E_1,E_2) \quad t_2 < t_1}{(q_1,q_2) \xrightarrow{(O_2,t_2)} (q_1,q_2')}$$

$$\frac{q_1 \xrightarrow{(O_1,t_1)}_1 q_1' \quad q_2 \xrightarrow{(O_2,t_2)}_2 q_2' \quad ((O_1,t_1),(O_2,t_2)) \in \kappa(E_1,E_2) \quad t_1 = t_2}{(q_1,q_2) \xrightarrow{(O_1 \cup O_2,t_1)} (q_1',q_2')}$$

Observe that the product is defined on pairs of transitions, which implies that if T_1 or T_2 has a state without outgoing transition, then the product has no outgoing transitions from that state. The reciprocal is, however, not true in general.

Theorem 7 states that the product of TES transition systems denotes (given a state) the set of TESs that corresponds to the product of the corresponding components (in their respective states). Then, the product that we define on TES transition systems does not add nor remove behaviors with respect to the product on their respective components.

Theorem 7 (Correctness)**.** *For all TES transition systems T_1 and T_2, and for all composability relation κ:*

$$C_{T_1 \times_\kappa T_2}(q_1, q_2) = C_{T_1}(q_1) \times_{([\kappa],[\cup])} C_{T_2}(q_2)$$

Proof. Let $T_1 = (Q_1, E_1, \rightarrow_1)$ and $T_2 = (Q_2, E_2, \rightarrow_2)$. The proof goes in two directions:

1. We first show that, for any τ in the behavior of $\mathcal{L}^{\text{inf}}(T_1 \times_\kappa T_2, (q_1, q_2))$, there exist $\tau_1 \in \mathcal{L}^{\text{inf}}(T_1, q_1)$ and $\tau_2 \in \mathcal{L}^{\text{inf}}(T_2, q_2)$ such that $\tau = \tau_1[\cup]\tau_2$ and $(\tau_1, \tau_2) \in [\kappa](E_1, E_2)$.

2. We then show that, for any $\tau_1 \in \mathcal{L}^{\text{inf}}(T_1, q_1)$ and $\tau_2 \in \mathcal{L}^{\text{inf}}(T_2, q_2)$ such that $(\tau_1, \tau_2) \in [\kappa](E_1, E_2)$, we have $(\tau_1[\cup]\tau_2) \in \mathcal{L}^{\text{inf}}(T_1 \times_\kappa T_2, (q_1, q_2))$.

We recall the definition of $\Phi_\kappa : \mathcal{P}(TES(E) \times TES(E)) \rightarrow \mathcal{P}(TES(E) \times TES(E))$, the function defining the lifting of κ from observations to TESs (by co-induction), to be such that, for all $\mathcal{R} \subseteq TES(E) \times TES(E)$:

$$\Phi_\kappa(\mathcal{R}) = \{(\tau_1, \tau_2) \mid \quad (\tau_1(0), \tau_2(0)) \in \kappa(E_1, E_2) \wedge (\tau_1, \tau_2)' \in \mathcal{R}\}$$

First, observe that, by construction, for all $\tau \in \Omega_{T_1 \times_\kappa T_2}(q_1, q_2)$, there exist $\tau_1 \in \Omega_{T_1}(q_1)$ and $\tau_2 \in \Omega_{T_2}(q_2)$ such that $\tau = \tau_1[\cup]\tau_2$. Indeed, each transition in $T_1 \times_\kappa T_2$ is constructed out of a composable pair of observations from a transition in T_1 and in T_2. The resulting label is identical to the label that $[\cup]$ defines co-inductively. Moreover, each element $\tau \in \Omega_{T_1 \times_\kappa T_2}(q_1, q_2)$ contains infinitely many labels from T_1 and from T_2, due to the assumption that a TES has an increasing an non-Zeno sequence of time stamps.

We therefore use $\widetilde{\Omega}_{T_1 \times_\kappa T_2}(q_1, q_2)$ to denote the set of such pairs. We then prove that, for all $(q_1, q_2) \in Q_1 \times Q_2$, $(\tau_1, \tau_2) \in \widetilde{\Omega}_{T_1 \times_\kappa T_2}(q_1, q_2)$ implies that $(\tau_1, \tau_2) \in [\kappa](E_1, E_2)$ and $(\tau_1, \tau_2) \in \Omega_{T_1}(q_1) \times \Omega_{T_2}(q_2)$.

We prove that $\widetilde{\Omega}_{T_1 \times_\kappa T_2}(q_1, q_2) = \{(\tau_1, \tau_2) \in [\kappa](E_1, E_2) \mid (\tau_1, \tau_2) \in \Omega_{T_1}(q_1) \times \Omega_{T_2}(q_2)\}$ by showing forward and backward inclusion, i.e., point 1 and point 2 respectively.

Forward inclusion. We know that since $(\tau_1, \tau_2) \in \widetilde{\Omega}_{T_1 \times_\kappa T_2}(q_1, q_2)$, then there exists a post fixed point R of $\Phi_{T_1 \times_\kappa T_2}$ such that $((\tau_1, \tau_2), (q_1, q_2)) \in R$. Let $\widetilde{R}$ be the set $\{(\tau_1, \tau_2) \mid \exists q.((\tau_1, \tau_2), q) \in R\}$. We show that $\widetilde{R}$ is a post fixed point of $[\kappa](E_1, E_2)$. By definition of the transition relation $\rightarrow$ of $T_1 \times_\kappa T_2$, we have that if $(\tau_1, \tau_2) \in \widetilde{R}$, then $(\tau_1(0), \tau_2(0)) \in \kappa(E_1, E_2)$ and, if $t_1 < t_2$ then $(\tau_1', \tau_2) \in \widetilde{R}$, if $t_2 < t_1$ then $(\tau_1, \tau_2') \in \widetilde{R}$, and if $t_1 = t_2$ then $(\tau_1', \tau_2') \in \widetilde{R}$. We thus conclude that $\widetilde{R} \subseteq \Phi_\kappa(\widetilde{R})$, and therefore $\widetilde{\Omega}_{T_1 \times_\kappa T_2}(q_1, q_2) \subseteq [\kappa](E_1, E_2)$.

We then showed that, for any τ in the behavior of $\mathcal{L}^{\inf}(T_1 \times_\kappa T_2, (q_1, q_2))$, there exist $\tau_1 \in \mathcal{L}^{\inf}(T_1, q_1)$ and $\tau_2 \in \mathcal{L}^{\inf}(T_2, q_2)$ such that $\tau = \tau_1[\cup]\tau_2$ and $(\tau_1, \tau_2) \in [\kappa](E_1, E_2)$.

Backward inclusion. We show that, for all $(q_1, q_2) \in Q_1 \times Q_2$, if $(\tau_1, \tau_2) \in [\kappa](E_1, E_2)$ and $(\tau_1, \tau_2) \in \Omega_{T_1}(q_1) \times \Omega_{T_2}(q_2)$ then $(\tau_1, \tau_2) \in \widetilde{\Omega}_{T_1 \times_\kappa T_2}(q_1, q_2)$.

To do so, we construct a post fixed points of Φ_κ, which we denote S, such that $((\tau_1, \tau_2), (q_1, q_2)) \in S$; and, for all $((\tau, \sigma), (q, s)) \in S$, we have $\tau(0) = (O_1, t_1)$ and $\sigma(0) = (O_2, t_2)$:

- if $t_1 < t_2$ and $(q_1, q_2) \xrightarrow{\tau(0)} (q_1', q_2)$ then $((\tau', \sigma), (q_1', q_2)) \in S$;
- if $t_2 < t_1$ and $(q_1, q_2) \xrightarrow{\sigma(0)} (q_1, q_2')$ then $((\tau, \sigma'), (q_1, q_2')) \in S$; and
- if $t_2 = t_1$ and $(q_1, q_2) \xrightarrow{\sigma(0) \cup \tau(0)} (q_1', q_2')$ then $((\tau', \sigma'), (q_1', q_2')) \in S$.

Since $(\tau_1, \tau_2) \in [\kappa](E_1, E_2)$, we know that $S \subseteq [\kappa](E_1, E_2)$. We now show that $S \subseteq \Phi_{T_1 \times_\kappa T_2}(S)$. The argument for S being a post fixed point of $\Phi_{T_1 \times_\kappa T_2}$ follows from the definition of S. □

We give in Example 49 the TES transition systems resulting from the product of the TES transition systems of two robots and a grid. Example 49 defines operationally the components in Section 2.2, i.e., their behavior is generated by a TES transition system.

Example 49. *Let T_{R_1}, T_{R_2} be two TES transition systems for robots R_1 and R_2, and let $T_G(\{1\}, n, m)$ be a grid with robot R_1 alone and $T_G(\{1, 2\}, n, m)$ be a grid with robots R_1 and R_2. We use κ^{sync} as defined in Example 11.*

The product of T_{R_1}, T_{R_2}, and $T_G(\{1, 2\}, n, m)$ under κ^{sync} is the TES transition system $T_{R_1} \times_{\kappa^{sync}} T_{R_2} \times_{\kappa^{sync}} T_G(\{1, 2\}, n, m)$ such that it synchronizes observations of the two robots with the grid, but does not synchronize events of the two robots directly, since the two set of events are disjoint. ■

As a consequence of Theorem 1, letting κ^{sync} be the composability relation used in the product $\bowtie$ and writing $T = T_{R_1} \times_{\kappa^{sync}} T_{R_2} \times_{\kappa^{sync}} T_G$, $C_T(q1, q2, q3)$ is equal to the component $C_{T_{R_1}}(q_1) \bowtie C_{T_{R_2}}(q_2) \bowtie C_{T_G}(q_3)$

Definition 32. *Let T be a TES transition system, and let $C_T(q) = (E, \mathcal{L}^{\inf}(T, q))$ be a component whose behavior is defined by T. Then, C is* deadlock free *if and only if $FG(\mathcal{L}^{\inf}(T, q)) = \mathcal{L}^{\text{fin}}(T, q) \neq \emptyset$. As a consequence, we also say that (T, q) is deadlock free when $C_T(q)$ is deadlock free.*

A class of deadlock free components is that of components that accept arbitrary insertion of $\emptyset$ observables in between two observations. We say that such component is *prefix-closed*, as every sequence of finite observations can be continued by an infinite sequence of empty observables, i.e., C is such that $C = C^*$ (as defined after Definition 8). We say that a TES transition system T is prefix-closed if and only if and only if $C_T(q) = C^*_T(q)$. For instance, if T is such that, for any state q and for any $t \in \mathbb{R}_+$ there is a transition $q \xrightarrow{(\emptyset,t)} q$, then T is prefix-closed.

Lemma 19. *If T_1 and T_2 are prefix-closed, then $T_1 \times_{\kappa^{sync}} T_2$ is prefix-closed.*

Proof. The proof follows from the fact that $\emptyset$ is independent with any non-empty observable $O \subseteq E_1 \cup E_2$. Then, any pair of silent observation is composable, and therefore the following TES transition system is prefix-closed. □

We search for the condition under which deadlock freedom is preserved under a product. Section 3.3 gives a condition for the product of two deadlock free components to be deadlock free.

4.1.2 Compatibility of components

Informally, the condition of κ-compatibility of two TES transition systems T_1 and T_2, respectively in initial state q_1 and q_2, translates the existence of a relation $\mathcal{R}$ on pairs of states of T_1 and T_2 such that $(q_1, q_2) \in \mathcal{R}$ and for every state $(q, s) \in \mathcal{R}$, there exists an outgoing transition from T_1 (reciprocally T_2) that composes under κ with an outgoing transition of T_1 (respectively T_1). The pair of outgoing states is in the relation $\mathcal{R}$.

Formally, a TES transition system $T_1 = (Q_1, E_1, \rightarrow_1)$ from state q_1 is κ-compatible with a TES transition system $T_2 = (Q_2, E_2, \rightarrow_2)$ from state q_2, and we say (T_1, q_1) is κ-compatible with (T_2, q_2) if there exists a relation $\mathcal{R} \subseteq Q_1 \times Q_2$ such that $(q_1, q_2) \in \mathcal{R}$ and for any $(p_1, p_2) \in \mathcal{R}$,

- there exist $p_1 \xrightarrow{(O_1,t_1)}_1 r_1$ and $p_2 \xrightarrow{(O_2,t_2)}_2 r_2$ such that $((O_1,t_1),(O_2,t_2)) \in \kappa(E_1,E_2)$; and
- for all $p_1 \xrightarrow{(O_1,t_1)}_1 r_1$ and $p_2 \xrightarrow{(O_2,t_2)}_2 r_2$ if $((O_1,t_1),(O_2,t_2)) \in \kappa(E_1,E_2)$ then $(u_1,u_2) \in \mathbb{R}$, where $u_i = r_i$ if $t_i = \min\{t_1,t_2\}$, and $u_i = p_i$ otherwise for $i \in \{1,2\}$.

In other words, if (T_1,q_1) is κ-compatible with (T_2,q_2), then there exists a composable pair of transitions in T_1 and T_2 from each pair of states in $\mathcal{R}$ (first item of the definition), and all pairs of transitions in T_1 composable with a transition in T_2 from a state in $\mathcal{R}$ end in a pair of states related by $\mathcal{R}$. If (T_2,q_2) is κ-compatible to (T_1,q_1) as well, then we say that (T_1,q_1) and (T_2,q_2) are κ-compatible.

Theorem 8 (Deadlock free)**.** *Let (T_1,q_1) and (T_2,q_2) be κ-compatible. Let $C_{T_1}(q_1)$ and $C_{T_2}(q_2)$ be deadlock free, as defined in Definition 32. Then, $C_{T_1}(q_1) \times_{([\kappa],[\cup])} C_{T_2}(q_2)$ is deadlock free.*

Proof. We reason by contradiction. If the product $C_{T_1}(q_1) \times_{([\kappa],[\cup])} C_{T_2}(q_2)$ is not deadlock free, then $\mathcal{L}^{\text{fin}}(T_1 \times_\kappa T_2,(q_1,q_2)) \neq FG(\mathcal{L}^{\text{inf}}(T_1 \times_\kappa T_2,(q_1,q_2)))$. Thus, there exists a state (s,q), reachable from (q_1,q_2), such that $\mathcal{L}^{\text{fin}}(T_1 \times_\kappa T_2,(s,q)) = \emptyset$, i.e., no pairs of compatible transitions from T_1 and T_2 in states s and q respectively. Given the fact that both TES transition systems are deadlock free, and given that s (respectively q) is reachable from q_1 (respectively q_2) for T_1 (respectively T_2), then $\mathcal{L}^{\text{fin}}(T_2,q_1) \neq \emptyset$ and $\mathcal{L}^{\text{fin}}(T_1,q_2) \neq \emptyset$.

Since (T_1,q_1) and (T_2,q_2) are κ-compatible, then there exists $\mathcal{R}$ such that for each pair $(s,q) \in \mathcal{R}$, there exists an outgoing transition in T_1 and T_2 from s and q respectively that is composable under κ. Such property would imply that there is a transition in $T_1 \times_\kappa T_2$ from state (s,q) and therefore $\mathcal{L}^{\text{fin}}(T_1 \times_\kappa T_2,(s,q)) \neq \emptyset$. In other words, the property of compatibility contradicts the presence of deadlock in the product $C_{T_1}(q_1) \times_{([\kappa],[\cup])} C_{T_2}(q_2)$. □

Lemma 20. *Let $T_1 = (Q_1,E_1,\rightarrow_1)$ and $T_2 = (Q_1,E_1,\rightarrow_1)$ be two TES transition systems and $(q_1,q_2) \in Q_1 \times Q_2$. If (T_1,q_1) and (T_2,q_2) are deadlock free and $E_1 \cap E_2 = \emptyset$, then (T_1,q_1) is κ^{sync}-compatible with (T_2,q_2).*

Proof. If $E_1 \cap E_2 = \emptyset$, then any pair of observations is composable under κ^{sync}. As a consequence, any sequence of transitions in T_1 and T_2 is allowed. □

The consequence of two TES transition systems T_1 and T_2 to be κ-compatible on (q_1,q_2) and deadlock free, is that they can be run *step-by-step* from (q_1,q_2) (i.e.,

the product can be done at runtime), and the resulting behavior is an element of $\mathcal{L}^{\inf}(T_1 \times_\kappa T_2, (q_1, q_2))$. In general however, κ-compatibility is not preserved over product, demonstrated by Example 50. For the case of coordinated cyber-physical systems, components are usually not prefix-closed as there might be some timing constraints or some mandatory actions to perform in a bounded time frame.

Example 50. *Suppose three TES transition systems $T_i = (\{q_i\}, \{a, b, c, d\}, \rightarrow_i)$, with $i \in \{1, 2, 3\}$, defined as follow for all $n \in \mathbb{N}$:*

- $q_1 \xrightarrow{(\{a,b\},n)}_1 q_1$ *and* $q_1 \xrightarrow{(\{a,c\},n)}_1 q_1$*;*
- $q_2 \xrightarrow{(\{a,c\},n)}_2 q_2$ *and* $q_2 \xrightarrow{(\{a,d\},n)}_2 q_2$*;*
- $q_3 \xrightarrow{(\{a,d\},n)}_3 q_3$ *and* $q_3 \xrightarrow{(\{a,b\},n)}_3 q_3$*.*

It is easy to show that $T_1(q_1)$, $T_2(q_2)$, and $T_3(q_3)$ are pairwise κ^{sync}-compatible. However, $T_1(q_1)$ is not κ^{sync}-compatible with $T_2(q_2) \times_{\kappa^{sync}} T_3(q_3)$. ■

4.2 Components as rewrite systems

We start by giving an illustration of our approach on an intuitive and simple cyber-physical system consisting of two robots roaming on a shared field. A robot exhibits some cyber aspects, as it takes discrete actions based on its readings. Every robot interacts, as well, with a shared physical resource as it moves around. The field models the continuous response of each action (e.g., read or move) performed by a robot. A question that will motivate the section is: given a strategy for both robots (i.e., sequence of moves based on their readings), will both robots, sharing the same physical resource, achieve their goals? If not, can the two robots, without changing their policy, be externally coordinated towards their goals?

In this section, we specify components in a rewriting framework in order to simulate and analyze their behavior. In this framework, an *agent*, e.g., a robot or a field, specifies a component as a rewriting theory. A *system* is a set of agents that run concurrently. The equational theory of an agent defines how the agent states are updated, and may exhibit both continuous and discrete transformations. The dynamics is captured by rewriting rules and an equational theory at the system level that describes how agents interact. In our example, for instance, each move of a robot is synchronous with an effect on the field. Each agent therefore specifies how the action

affects its state, and the system specifies which composite actions (i.e., set of simultaneous actions) may occur. We give hereafter an intuitive example that abstracts from the underlying algebra of each agent.

Agent A robot and a field are two examples of an agent that specifies a component as a rewriting theory. The dynamics of both agents is captured by a rewrite rule of the form:

$$(s, \emptyset) \Rightarrow (s', acts)$$

where s and s' are state terms, and $acts$ is a set of actions that the field or the robot proposes as alternatives. Given an action $a \in acts$ from the set of possibilities, a function ϕ updates the state s and returns a new state $\phi(s', a)$. The equational theory that specifies ϕ may capture both discrete and continuous changes. The robot and the field run concurrently in a system, where their actions may interact.

Example 51 (Battery)**.** *A battery is characterized by a set of internal physical laws that describe the evolution of its energy profile over time under external stimulations. We consider three external stimuli for the battery as three events: a charge, a discharge, and a read event. Each of those events may change the profile of the battery, and we assume that in between two events, the battery energy follows some fixed internal laws. Formally, we model the energy profile of a battery as a function* $f : \mathbb{R}_+ \to [0, 100\%]$ *where* $f(t) = 50\%$ *means that the charge of the battery at time* t *is of* 50%*. In general, the co-domain of* f *may be arbitrarily complex, and captures the response of event occurrences (e.g., charge, discharge, read) and passage of time coherently with the underlying laws (e.g., differential equation). For instance, a charge (or discharge) event at a time* t *coincides with a change of slope in the function* f *after time* t *and before the next event occurrence.*
For simplicity, we consider a battery for which f *is piecewise linear in between any two events. The slope changes according to some internal laws at points where the battery is used for charge or discharge.*
In our model, a battery interacts with its environment only at discrete time points. Therefore, we model the observables of a battery as a function $l : \mathbb{N} \to [0, 100\%]$ *that intuitively samples the state of the battery at some monotonically increasing and non-Zeno sequence of timestamp values. We capture, in Definition 1, the continuous profile of a battery as a component whose behavior contains all of such increasing and non-Zeno sampling sequences for all continuous functions* f*.*

Example 52 (Robot)**.** *A robot's state contains the previously read values of its sen-*

sors. Based on its state, a robot decides to move in some specific direction or read its sensors.
Similarly to the battery, we assume that a robot acts periodically at some discrete points in time, such as the sequence move(E) (i.e., moving East) at time 0, read($(x, y), l$) (i.e., reading the position (x, y) and the battery level l) at time T, move(W) (i.e., moving West) at time $3T$ while doing nothing at time $2T$, etc. The action may have as effect to change the robot's state: typically, the action read($(x, y), l$) updates the state of the robot with the coordinate (x, y) and the battery value l.

System A system is a set of agents together with a composability constraint κ that restricts their updates. For instance, take a system that consists of a robot id and a field F. The concurrent execution of the two agents is given by the following system rewrite rule:

$$\{(s_{\text{id}}, acts_{\text{id}}), (s_F, acts_F)\} \Rightarrow_S \{(\phi_{\text{id}}(s_{\text{id}}, a_{\text{id}}), \emptyset), (\phi_F(s_F, a_F), \emptyset)\}$$

where $a_{\text{id}} \in acts_{\text{id}}$ and $a_F \in acts_F$ are two actions related by κ.

Each agent is unaware of the other agent's decisions. The system rewrite $\Rightarrow_S$ filters actions that do not comply with the composability relation κ. As a result, each agent updates its state with the (possibly composite) action chosen at runtime, from the list of its submitted actions. The framework therefore clearly separates the place where agent's and system's choices are handled, which is a source of runtime analysis.

Already, at this stage, we can ask the following question on the system: will robot id eventually reach the location (x, y) on the field? Note that the agent alone cannot answer the query, as the answer depends on the characteristics of the field.

Example 53 (Battery-Robot)**.** *Typically, a move of the robot* synchronizes *with a change of state in the battery, and a read of the robot occurs at the same time as a sampling of the battery value.*
The system behavior therefore consists of sequences of simultaneous events occurring between the battery and the robot. By composition, the battery exposes the subset of its behavior that conforms to the specific frequency of read and move actions of the robot. The openness of the battery therefore is reflected by its capacity to adapt to any observation frequency.

Coordination Consider now a system with three agents: two robots and a field. Each robot has its own objective (i.e., location to reach) and strategy (i.e., sequence

of moves). Since both robots share the same physical field, some exclusion principals apply, e.g., no two robots can be at the same location on the field at the same time. It is therefore possible that the system deadlocks if no actions are composable, or livelocks if the robots enter an infinite sequence of repeated moves.

We add a protocol agent to the system, which imposes some coordination constraints on the actions performed by robots id_1 and id_2. Typically, a protocol coordinates robots by forcing them to do some specific actions. As a result, given a system configuration $\{(s_{\mathrm{id}_1}, acts_{\mathrm{id}_1}), (s_{\mathrm{id}_2}, acts_{\mathrm{id}_2}), (s_F, acts_F), (s_P, acts_P)\}$ the run of robots id_1 and id_2 has to agree with the observations of the protocol, and the sequence of actions for each robot will therefore be conform to a permissible sequence under the protocol.

In the case where the two robots enter a livelock and eventually run out of energy, we show in Section 5.4.1 the possibility of using a protocol to remove such behavior.

Example 54 (Safety property)**.** *A safety property is typically a set of traces for which nothing bad happens. In our framework, we consider only observable behaviors, and a safety property therefore declares that nothing bad* is observable*. However, it is not sufficient for a system to satisfy a safety property to conclude that it is safe: an observation that would make a sequence violate the safety property may be absent, not because it did not actually happen, but merely because the system missed to detect it. For example, consider a product of a battery component and a robot with a sampling period T, as introduced in Example 53. Consider the safety property:* the battery energy is between the energy thresholds e_1 and e_2*. The resulting system may exhibit observations with energy readings between the two thresholds only, and therefore satisfy the property. However, had the robot used a smaller sampling period $T' = T/2$, which adds a reading observation of its battery between every two observations, we may have been able to detect that the system is not safe because it produces sequences at this finer granularity sampling rate that violate the safety property. We show how to algebraically capture the safety of a system constituted of a battery-robot.*

4.2.1 System of agents and compositional semantics

Components in Chapter 2 are declarative. Their behavior consists of a set of TESs. The abstraction of internal states in components makes the specification of observables and their interaction easier. The downside of such declarative specification lies in the difficulty of generating an element from the behavior, and ultimately verifying properties on a product expression.

An operational specification of a component provides a mechanism to construct elements in its behavior. An *agent* is the operational specification that produces finite sequences of observations that, in the limit, determine the behavior of a component. An agent is stateful, and has transitions between states, each labeled by an observation, i.e., a set of events with a time-stamp. We consider a finite specification of an agent as a rewrite theory, where finite applications of the agent's rewrite rules generate a sequence of observables that form a prefix of some elements in the behavior of its corresponding component. We restrict the current work to integer time labeled observations. While in the cyber-physical world, time is a real quantity, we consider in our fragment a countable infinite domain for time, i.e., natural numbers. The time interval between two tics is therefore the same for all agents, and may be interpreted as, e.g., seconds, milliseconds, femtoseconds, etc. We show how an agent may synchronize with a local clock that forbids actions at some time values, thus modeling different execution speeds.

An operational specification of a composite component provides a mechanism to construct elements in the behavior of a product expression. The product on components is parametrized by an interaction signature that tells *which* TESs can compose, and *how* they compose to a new TES. We consider, in the operational fragment of this section, interaction signatures each of whose composability relation is co-inductively defined from a relation on observations κ. Intuitively, such restriction enables a step-by-step operation to check that the head of each sequence is valid, i.e., extends the sequence to be a prefix of some elements in the composite component. Moreover, we require κ to be such that the product on component $\times_{([\kappa],\cup)}$ is commutative and associative (see [57]). By *system* we mean a set of agents that compose under some interaction signature $\Sigma = ([\kappa], \cup)$. A system is stateful, where each state is formed from the states of its component agents, and has transitions between states, each labeled by an observation, formed from the component agent observations. We consider a finite specification of a system as the composition of a set of rewriting theories (one for each agent), and a system rewrite rule that produces a composite observation complying with the relation κ. We prove compositionality: the system component is equal to the product under the interaction signature $\Sigma = ([\kappa], \cup)$ of every one of its constituent agent components.

We give the operational counterparts of an observation, a component, and a product of components as, respectively, an action, an agent, and a system of agents.

Action Actions are terms of sort Action. An action has a name of sort AName and some parameters. We distinguish two typical actions, the idle action $\star$ and the ending action end. A term of sort Action corresponds to an observable, i.e., a set of events. The idle action $\star$ and the ending action end both map to the empty set of events. An example of an action is move(R1,d) or read(R1, position, l) that, respectively, moves agent R1 in direction d or reads the value l from the position sensor of R1. The semantics of action move(R1, d) consists of all singleton events of the form {move(R1, d)} with d a constant direction value. We use the associative, commutative, and idempotent operation $\cdot$: Action Action $\rightarrow$ Action to construct a composite action a1 $\cdot$ a2 out of two actions a1 and a2.

Agent An agent operationally specifies a component in rewriting logic. We give the specification of an agent as a rewrite theory, and provide the semantics of an agent as a component. An agent is a four tuple $(\Lambda, \Omega, \mathcal{E}, \Rightarrow)$, each of whose elements we introduce as follow.

The set of sorts Λ contains the State sort and the Action sort, respectively for state and action terms. A pair of a state and a set of actions is called a configuration. The set of function symbols Ω contains ϕ : State $\times$ Action $\rightarrow$ State, that takes a pair of a state and an action term to produce a new state. The (Λ, Ω)-equational theory $\mathcal{E}$ specifies the update function ϕ. The set of equations that specify the function ϕ can make ϕ either a continuous or discrete function.

The rule pattern in (4.1) updates a configuration with an empty set to a new configuration, i.e.,

$$(s, \emptyset) \Rightarrow (s', acts) \tag{4.1}$$

with *acts* a non-empty set of action terms, and s' a new state. We call an agent *productive* if, for any state s : State, there exists a state s' with $(s, \emptyset) \Rightarrow (s', acts)$ and *acts* non empty set. Such agent may eventually do the idling action $\star$.

We give a semantics of an agent as a component by considering the limit application of the agent rewrite rules. We construct a TES transition system $\mathcal{T}_{\mathcal{A}} = (Q, E, \rightarrow)$ as an intermediate representation for agent $\mathcal{A} = (\Lambda, \Omega, \mathcal{E}, \Rightarrow)$. The set of states Q = State$\times\mathbb{N}$ is the set of pairs of a state of $\mathcal{A}$ and a time-stamp natural number. We use the notation $[s, t]$ for states in Q where $t \in \mathbb{N}$. The set of events E is the union of all observables labeling the transition relation $\rightarrow \subseteq Q \times (\mathcal{P}(E) \times \mathbb{N}) \times Q$, defined as the smallest set

such that, for $t \in \mathbb{N}$:

$$\frac{(s, \emptyset) \Rightarrow (s', acts) \qquad a \in acts \qquad \phi(s', a) =_{\mathcal{E}} s'' \qquad d \in \mathbb{N}}{[s, t] \xrightarrow{(a, t+d)} [s'', t + d]} \tag{4.2}$$

An agent that performs a rewrite moves the global time from an arbitrary but finite amount of time units. Note that we can safely consider $d \neq 0$, as the case of two consecuitive observations with the same time stamp is ruled out in the TES behavior of an agent (see below). All agents share the same time semantically, and we show some mechanisms at the system level to artificially run some agents *faster* than others.

Let $\mathcal{A} = (\Lambda, \Omega, \mathcal{E}, \Rightarrow)$ be an agent initially in state $s_0 \in S$ at time $t_0 \in \mathbb{N}$. The finite, respectively infinite, component semantics of $\mathcal{A}$ is the component $[\![\mathcal{A}([s_0, t_0])]\!]^* = (E, \mathcal{L}^{\text{fin}*}(\mathcal{T}_{\mathcal{A}}, [s_0, t_0]))$, respectively the component $[\![\mathcal{A}([s_0, t_0])]\!] = (E, \mathcal{L}^{\text{inf}}(\mathcal{T}_{\mathcal{A}}, [s_0, t_0]))$, with $E = \bigcup_{a \in \mathsf{Action}} a$.

Lemma 21 (Closure)**.** *Let $\mathcal{A}$ be a productive agent initially in state $[s_0, t_0]$. Then $[\![A([s_0, t_0])]\!]^* = cl([\![A([s_0, t_0])]\!])$.*

Proof. Given that $[\![A([s_0, t_0])]\!]^* = (E, \mathcal{L}^{\text{fin}*}(\mathcal{T}_{\mathcal{A}}, [s_0, t_0]))$ and given $cl([\![A([s_0, t_0])]\!]) = (E, \mathcal{L}^{\text{inf}}(\mathcal{T}_{\mathcal{A}}, [s_0, t_0]))$, we have to show that $\mathcal{L}^{\text{fin}*}(\mathcal{T}_{\mathcal{A}}, [s_0, t_0]) = cl(\mathcal{L}^{\text{inf}}(\mathcal{T}_{\mathcal{A}}, [s_0, t_0]))$.

$$\begin{aligned} cl(\mathcal{L}^{\text{inf}}(\mathcal{T}_{\mathcal{A}}, [s_0, t_0])) &= \{s\tau \in TES(E) \mid \tau \in TES(\emptyset) \;\; and \\ &\qquad\qquad \exists \sigma. \exists i. \sigma \in \mathcal{L}^{\text{inf}}(\mathcal{T}_{\mathcal{A}}, [s_0, t_0]) \wedge \sigma[i] = s\} \\ &= \{s\tau \in TES(E) \mid \tau \in TES(\emptyset) \;\; and \\ &\qquad\qquad \exists \sigma. \exists i. \forall n. \sigma[n] \in \mathcal{L}^{\text{fin}}(\mathcal{T}_{\mathcal{A}}, [s_0, t_0]) \wedge \sigma[i] = s\} \\ &\subseteq \mathcal{L}^{\text{fin}*}(\mathcal{T}_{\mathcal{A}}, [s_0, t_0]) \end{aligned}$$

The other direction comes from the assumption that $\mathcal{A}$ is productive. Then, every reachable state in $\mathcal{T}_A$ has an outgoing transition and therefore every finite sequence of transition is a prefix of an infinite sequence. Thus, $\mathcal{L}^{\text{fin}*}(\mathcal{T}_{\mathcal{A}}, [s_0, t_0]) \subseteq cl(\mathcal{L}^{\text{inf}}(\mathcal{T}_{\mathcal{A}}, [s_0, t_0]))$. □

Lemma 21 gives a condition under which a step by step execution of the agent is sound with respect to generating prefixes of elements in the component semantics. More precisely, if an agent $\mathcal{A}$ is productive, Lemma 21 ensures that finite sequences of rewrite rule applications generate finite sequences of observations each of which is a prefix of an element in the behavior of the component corresponding to $\mathcal{A}$. Alternatively, if $\mathcal{A}$ is not productive, a finite sequence of rule application may lead to a state

for which no rule applies anymore. In such a case, there may not be any corresponding element in the agent component for which such finite sequence is a prefix.

Remark 14. *An agent $\mathcal{A}$ initially in state s_0 at t_0 denotes a component $[\![\mathcal{A}([s_0, t_0])]\!]$. Note that, a strategy for agent $\mathcal{A}$ would be any mechanism that, given a state for agent $\mathcal{A}$, filters a subset of possible actions. For instance, an agent may decide to discard actions that bring it further from its goal. We give in Section 4.3 a Domain Specific Language to describe agents equipped with a strategy.*

System A system gives an operational specification of a product of a set of components under $\Sigma = ([\kappa], \cup)$. The composability relation κ is fixed to be symmetric, so that the product $\times_\Sigma$ is commutative. We define $[\kappa]$ co-inductively, as in [57, 56]. Formally, a system consists of a set of agents with additional sorts, operations, and rewrite rules. A system is a tuple $(\mathcal{A}, \Lambda, \Omega, \mathcal{E}, \Rightarrow_S)$ where $\mathcal{A}$ is a set of agents. We use $(\Lambda_i, \Omega_i, \mathcal{E}_i, \Rightarrow_i)$ to refer to agent $\mathcal{A}_i \in \mathcal{A}$.

The set of sorts Λ contains a sort $\mathsf{Action} \in \Lambda$ which is a super sort of each sort Action_i for $\mathcal{A}_i \in \mathcal{A}$. The set Ω contains the function symbol $\mathsf{comp} : \mathsf{Action} \times \mathsf{Action} \to \mathsf{Bool}$, which relates pairs of action terms. Given two actions $\mathtt{a1}$,$\mathtt{a2}$:Action, $\mathsf{comp}(\mathtt{a1}, \mathtt{a2}) = \mathsf{True}$ when the two actions $\mathtt{a1}$ and $\mathtt{a2}$ are *composable*. The set of equations $\mathcal{E}$ specifies the composability relation comp. First, we impose comp to be symmetric, i.e., for all actions $\mathtt{a1}$,$\mathtt{a2}$:Action, $\mathsf{comp}(\mathtt{a1}, \mathtt{a2}) = \mathsf{comp}(\mathtt{a2}, \mathtt{a1})$. Second, we assume that $\mathsf{comp}(\mathtt{a1} \cdot \mathtt{a2},\ \mathtt{a3})$ and $\mathsf{comp}(\mathtt{a1},\ \mathtt{a2})$ hold if and only if $\mathsf{comp}(\mathtt{a2},\ \mathtt{a3})$ and $\mathsf{comp}(\mathtt{a1},\ \mathtt{a2} \cdot \mathtt{a3})$ hold, for any actions $\mathtt{a1}$, $\mathtt{a2}$, $\mathtt{a3}$ from disjoint agents. Given a set $\mathtt{actions}$ of actions, we use the notation $\mathsf{comp}(\mathtt{actions})$ for the predicate that is True if all pairs of actions in $\mathtt{actions}$ are composable, i.e., for all $\mathtt{a1}$, $\mathtt{a2}$ in $\mathtt{actions}$, $\mathsf{comp}(\mathtt{a1}, \mathtt{a2})$ is True and for all agent $\mathcal{A}_i$ such that there is no $\mathtt{a3} : \mathtt{Action_i} \in \mathtt{actions}$, then $\mathsf{comp}(\mathtt{a1}, \star_\mathtt{i})$ is True. We call a set $\mathtt{actions}$ of actions for which $\mathsf{comp}(\mathtt{actions})$ holds, a *clique*. The conditions for a set of actions to form a clique models the fact that each action in the clique is independent from agent $\mathcal{A}_i$ with no action in that clique (see Chapter 5 for an instance of comp), and therefore composable with the silent action $\star_i$.The relation comp can be graphically modelled as an undirected graph relating actions, where a clique is a connected component.

The rewrite rule pattern in (4.3) selects a set of actions, at most one from each agent, checks that the set of actions forms a clique with respect to comp, and applies the update accordingly. For $\{k_1, ..., k_j\} \subseteq \{1, ..., n\}$:

$$\{(s_{k_1}, acts_{k_1}), ..., (s_{k_j}, acts_{k_j})\} \Rightarrow_S \{(\phi_{k_1}(s_{k_1}, a_{k_1}), \emptyset), ..., (\phi_{k_j}(s_{k_j}, a_{k_j}), \emptyset)\} \qquad (4.3)$$

if $\mathsf{comp}(\bigcup_{i\in[1,j]}\{a_{k_i}\}))$. As we show later, a system does not necessarily update all agents in lock steps, and an agent not doing an action may stay in the configuration $(s,\emptyset)$. As multiple cliques may be possible, there is non-determinism at the system level. Different strategies may therefore choose different cliques as, for instance, taking the largest clique.

We define the transition system for $\mathcal{S} = (\mathcal{A}, \Lambda, \Omega, \mathcal{E}, \Rightarrow_S)$ as the TES transition system $\mathcal{T}_\mathcal{S} = (Q, E, \rightarrow)$ with $Q = \mathsf{StateSet} \times \mathbb{N}$ the set of states, E the union of all observables labeling the transition relation $\rightarrow \subseteq Q \times (\mathcal{P}(E) \times \mathbb{N}) \times Q$, which is the smallest transition relation such that, for $\{k_1, ..., k_j\} \subseteq \{1, ..., n\}$:

$$\frac{\{(s_{k_i}, acts_{k_i})\}_{i\in[1,j]} \Rightarrow_S \{(\phi_{k_i}(s_{k_i}, a_{k_i}), \emptyset)\}_{i\in[1,j]} \quad \bigwedge_{i\in[1,j]} \phi_{k_i}(s_{k_i}, a_{k_i}) =_{\mathcal{E}_i} s''_{k_i}}{[\{s_i\}_{i\in[1,n]}, t] \xrightarrow{(\bigcup_{i\in[1,j]} a_{k_i}, t+d)} [\{s_1, ..., s''_{k_1}, ..., s''_{k_j}, ..., s_n\}, t+d]} \quad (4.4)$$

for $t, d \in \mathbb{N}$ and where we use the notation $\{x_i\}_{i\in[1,n]}$ for the set $\{x_1, ..., x_n\}$.

Remark 15. *The top left part of the rule is a rewrite transition at the system level. As defined earlier, the condition for such rewrite to apply is the formation of a clique by all of the actions in the update. The states and labels of the TES transition system (bottom of the rule) are sets of states and sets of labels from the TES transition system of every agent in the system.*

Let $\mathcal{A} = \{\mathcal{A}_1, ..., \mathcal{A}_n\}$ be a set of agents, and let $\mathcal{S} = (\mathcal{A}, \Lambda, \Omega, \mathcal{E}, \Rightarrow_S)$ be a system initially in state $\{(s_{0i}, \emptyset)\}_{i\in[1,n]}$ at time t_0 such that, for all $i \in [1, n]$, $\mathcal{A}_i$ is initially in state s_{0i} at time t_0. The finite, respectively infinite, semantics of initialized system $\mathcal{S}([s_0, t_0])$, is the component $[\![\mathcal{S}([s_0, t_0])]\!]^* = (E, \mathcal{L}^{\text{fin}*}(\mathcal{T}_\mathcal{S}, [s_0, t_0]))$, respectively $[\![\mathcal{S}([s_0, t_0])]\!] = (E, \mathcal{L}^{\text{inf}}(\mathcal{T}_\mathcal{S}, [s_0, t_0]))$, where $E = \bigcup_{i\in[1,n]} E_i$ with E_i the set of events for the agent component $[\![\mathcal{A}([s_{0i}, t_0])]\!]$.

Given a composability relation comp, we define the interaction signature $\Sigma = ([\kappa_\mathsf{comp}], \cup)$, with $\kappa_\mathsf{comp}(E_1, E_2) \subseteq (\mathcal{P}(E_1) \times \mathbb{N}) \times (\mathcal{P}(E_2) \times \mathbb{N})$ to be such that, for $\mathtt{ai} : \mathtt{Action_i}$ and $\mathtt{aj} : \mathtt{Action_j}$:

- if $\mathsf{comp}(\mathtt{ai}, \mathtt{aj})$, then $((a_i, n), (a_j, n)) \in \kappa_\mathsf{comp}(E_i, E_j)$ for all $n \in \mathbb{N}$, i.e., two composable actions occur at the same time;
- $\mathsf{comp}(\mathtt{ai}, \star_\mathtt{j})$ if and only if $((a_i, n), (a, k)) \in \kappa_\mathsf{comp}(E_i, E_j)$ for all $(a, k) \in \mathcal{P}(E_j) \times \mathbb{N}$ with $k > n$, i.e., $\mathcal{A}_i$ can do the a_i action independently to the action of $\mathcal{A}_j$.

with E_i the set of events of agent $\mathcal{A}_i$.

Lemma 22 (Composability)**.** *If* $\texttt{Action}_\texttt{i} \cap \texttt{Action}_\texttt{j} = \emptyset$ *for all disjoint agents* i *and* j*, then the product* $\times_{([\kappa_{\mathsf{comp}}],\cup)}$ *is commutative and associative.*

Proof. We abbreviate κ_{comp} to κ, and use $\Sigma = ([\kappa], \cup)$. We know that:

1. for all actions $\mathtt{a1}$ and $\mathtt{a2}$, $\mathsf{comp}(\mathtt{a1},\mathtt{a2}) = \mathsf{comp}(\mathtt{a2},\mathtt{a1})$;
2. for all $\mathtt{a1} : \texttt{Action}_1$, $\mathtt{a2} : \texttt{Action}_2$, and $\mathtt{a3} : \texttt{Action}_3$, $\mathsf{comp}(\mathtt{a1}, \mathtt{a2})$ and $\mathsf{comp}(\mathtt{a1} \cdot \mathtt{a2},\ \mathtt{a3})$ if and only if $\mathsf{comp}(\mathtt{a2}, \mathtt{a3})$ and $\mathsf{comp}(\mathtt{a1},\ \mathtt{a2} \cdot \mathtt{a3})$.

Item 1 implies symmetry of κ and commutativity of $\times_\Sigma$.

We show that, for three observations (a_1, n), (a_2, k), and (a_3, l):

$$\begin{aligned}&((a_1,n),(a_2,k)) \in \kappa(E_1,E_2) \wedge ((a_1,n)+(a_2,k),(a_3,l)) \in \kappa(E_1 \cup E_2, E_3)\\ \iff &((a_2,k),(a_3,l)) \in \kappa(E_2,E_3) \wedge ((a_1,n),(a_2,k)+(a_3,l)) \in \kappa(E_1, E_2 \cup E_3)\end{aligned}$$

where $((a,u),(b,v)) = (a \cup b, u)$ if $u = v$, (a,u) if $u < v$, and (b,v) otherwise.

Suppose that $n = k = l$. Then,

$$\begin{aligned}&((a_1,n),(a_2,n)) \in \kappa(E_1,E_2) \wedge ((a_1 \cup a_2,n),(a_3,n)) \in \kappa(E_1 \cup E_2, E_3)\\ \iff &\ \mathsf{comp}(\mathtt{a1},\mathtt{a2}) \wedge \mathsf{comp}(\mathtt{a1} \cdot \mathtt{a2}, \mathtt{a3})\\ \iff &\ \mathsf{comp}(\mathtt{a2},\mathtt{a3}) \wedge \mathsf{comp}(\mathtt{a1}, \mathtt{a2} \cdot \mathtt{a3})\\ \iff &((a_2,n),(a_3,n)) \in \kappa(E_2,E_3) \wedge ((a_1,n),(a_2 \cup a_3,n)) \in \kappa(E_1, E_2 \cup E_3)\end{aligned}$$

The second equivalence follows from E_1 and E_2 being disjoint.

Suppose that $n < k$, then $((a_1,n),(a_2,k)) \in \kappa(E_1,E_2)$ if and only if $((a_1,n),(\emptyset,n)) \in \kappa(E_1,E_2)$, by definition of κ. Thus, for $n = l < k$, we have:

$$\begin{aligned}&((a_1,n),(a_2,k)) \in \kappa(E_1,E_2) \wedge ((a_1,n),(a_3,n)) \in \kappa(E_1 \cup E_2, E_3)\\ \iff &((a_1,n),(\emptyset,n)) \in \kappa(E_1,E_2) \wedge ((a_1,n),(a_3,n)) \in \kappa(E_1 \cup E_2, E_3)\\ \iff &\ \mathsf{comp}(\mathtt{a1},\star_2) \wedge \mathsf{comp}(\mathtt{a1} \cdot \star_2, \mathtt{a3})\\ \iff &\ \mathsf{comp}(\star_2,\mathtt{a3}) \wedge \mathsf{comp}(\mathtt{a1}, \star_2 \cdot \mathtt{a3})\\ \iff &((a_2,k),(a_3,n)) \in \kappa(E_2,E_3) \wedge ((a_1,n),(a_3,n)) \in \kappa(E_1, E_2 \cup E_3)\end{aligned}$$

Similar reasoning apply when $n \neq l$ or $l \neq k$.

We can conclude that κ_{comp} satisfies the condition of Lemma 10, and $\times_{([\kappa_{\mathsf{comp}}],\cup)}$ is commutative and associative. □

Theorem 9 (Compositional semantics)**.** *Let* $\mathcal{S} = (\mathcal{A}, \Lambda, \Omega, \mathcal{E}, \Rightarrow_S)$ *be a system of* n *agents with disjoint actions and* $[\{s_{01}, ..., s_{0n}\}, t_0]$ *as initial state. We fix* $\Sigma = ([\kappa_{\mathsf{comp}}], \cup)$*. Then,* $[\![\mathcal{S}([s_0, t_0])]\!] = \times_\Sigma \{[\![\mathcal{A}_i([s_{0i}, t_0])]\!]\}_{i\in[1,n]}$.

Proof. The proof uses the result of Lemma 22 that $\times_{([\kappa_{\mathsf{comp}}],\cup)}$ is associative and commutative. Then, we give an inductive proof that $[\![\mathcal{S}([s_0, t_0])]\!] = \times_\Sigma \{[\![\mathcal{A}_i([s_{0i}, t_0])]\!]\}_{i\in[1,n]}$. We fix $\mathcal{S} = (\{\mathcal{A}_1, ..., \mathcal{A}_n\}, \Lambda, \Omega, \mathcal{E}, \Rightarrow_S)$ and $\mathcal{A}_{n+1} = (\Lambda_{n+1}, \Omega_{n+1}, \mathcal{E}_{n+1}, \Rightarrow_{n+1})$, such that comp in Ω relates actions of agents in $\{\mathcal{A}_1, ..., \mathcal{A}_{n+1}\}$.
Let $\mathcal{S}' = (\{\mathcal{A}_1, ..., \mathcal{A}_n, \mathcal{A}_{n+1}\}, \Lambda, \Omega, \Rightarrow_S)$. We show that $\mathcal{T}_\mathcal{S} \times_\kappa \mathcal{T}_{\mathcal{A}_{n+1}} = (Q, E, \rightarrow)$ and $\mathcal{T}_{\mathcal{S}'} = (Q', E', \rightarrow')$ are bisimilar, which consists in the existence of a relation $\mathcal{R} \subseteq Q \times Q'$ such that, for all $(q, r) \in \mathcal{R}$:

1. $\forall q' \in Q$ with $q \xrightarrow{(O,t)} q'$, there exists $r' \in Q'$ with $r \xrightarrow{(O,t)}{}' r'$; and

2. $\forall r' \in Q'$ with $r \xrightarrow{(O,t)}{}' r'$, there exists $q' \in Q$ with $q \xrightarrow{(O,t)} q'$.

First, we define an equivalence relation $\sim$ on states in $Q \times Q'$ as $([s_\mathcal{S}, t], [s_\mathcal{A}, t']) \sim [(s_\mathcal{S}, s_\mathcal{A}), \max(t, t')]$. We show that the TES transition system $\mathcal{T}_\mathcal{S} \times_\kappa \mathcal{T}_{\mathcal{A}_{n+1}}$ and $\mathcal{T}_{\mathcal{S}'}$ are bisimilar, given the witness relation $\sim$.

Let $(s, s') \in\sim$ with $s = ([s_\mathcal{S}, t_1], [s_{\mathcal{A}_{n+1}}, t_2])$ where $s' = [(s_\mathcal{S}, s_{\mathcal{A}_{n+1}}), \max(t_1, t_2)]$. We study the case where $\mathcal{S}$ takes a step, while $\mathcal{A}_{n+1}$ stays idle (other cases are symmetric). Then, there is a transition $s \xrightarrow{(O,t)} q$ with $q = ([q_\mathcal{S}, t_1], [q_{\mathcal{A}_{n+1}}, t]) \in Q$ with $t > \max(t_1, t_2)$. As a result, there exists $q' = [(q_\mathcal{S}, q_{\mathcal{A}_{n+1}}), t] \in Q'$ with $s' \xrightarrow{(O,t)}{}' q'$ and $(q, q') \in\sim$. In the case where both $\mathcal{S}$ and $\mathcal{A}_{n+1}$ make a simultaneous move, then the transition is labeled by a time that is greater that both times, and the resulting state has updated the time from both the system and the agent.

Symmetrically, let $s' \xrightarrow{(O,t)} q'$ with $q' = [(q_\mathcal{S}, q_{\mathcal{A}_{n+1}}), t] \in Q$ with $t > \max(t_1, t_2)$. Then, if the observation comes from $\mathcal{S}$, there is a state $q = ([q_\mathcal{S}, t_1], [q_{\mathcal{A}_{n+1}}, t])$ such that $s \xrightarrow{(O,t)} q$ and $(q, q') \in\sim$. The argument is symmetric for the case of $\mathcal{A}$ being the only contributor to the observation. Lastly, if both $\mathcal{S}$ and $\mathcal{A}$ contribute to the observation, then the is a state $q = ([q_\mathcal{S}, t], [q_{\mathcal{A}_{n+1}}, t])$ such that $s \xrightarrow{(O,t)} q$ and $(q, q') \in\sim$.

As a result, we conclude that $\mathcal{T}_{\mathcal{S}'}$ and $\mathcal{T}_\mathcal{S} \times_\kappa \mathcal{T}_{\mathcal{A}_{n+1}}$ are bisimilar, and, by associativity and commutativity of $\times_\Sigma$, we conclude that $[\![\mathcal{S}([s_0, t_0])]\!] = \times_\Sigma \{[\![\mathcal{A}_i([s_{0i}, t_0])]\!]\}_{i\in[1,n]}$.

□

4.3 DSL for agents with preferences

In [45], we propose an automata-based paradigm based on soft constraint automata [8, 44], called soft component automata (SCAs). An SCA is a state-transition system where transitions are labeled with actions and preferences. Higher-preference transitions typically contribute more towards the goal of the component; if a component is in a state where it wants the system to move north, a transition with action north has a higher preference than a transition with action south. At run-time, preferences provide a natural fallback mechanism for an agent: in ideal circumstances, the agent would perform only actions with the highest preferences, but if the most-preferred actions fail, the agent may be permitted to choose a transition of lower preference. At design-time, preferences can be used to reason about the behavior of the SCA in suboptimal conditions, by allowing all actions whose preference is bounded from below by a threshold. In particular, this is useful if the designer wants to determine the circumstances where a property is no longer verified by the system.

The algebraic structure for preferences is called a *constraint semiring* and was proposed in [22]. A *c-semiring* is a tuple $(A, +, \times, 0, 1)$ such that

1. A is a carrier set that contains two element $0, 1 \in A$;
2. $+$ is a commutative associative idempotent binary operator, with unit element 0 and absorbing element 1;
3. $\times$ is a commutative associative binary operator that distributes over $+$, with unit element 1, and absorbing element 0.

Well-known instances of c-semirings are the

- *boolean* c-semiring $\mathbb{B} = (\{0, 1\}, \min, \max, 0, 1)$;
- *fuzzy* c-semiring $\mathbb{F} = ([0, 1], \min, \max, 0, 1)$;
- *bottleneck* c-semiring $\mathbb{K} = (\mathbb{R}_{\geq} \cup \{\infty\}, \max, \min, 0, \infty)$;
- *probabilistic* or *Viterbi* c-semiring $\mathbb{V} = ([0, 1], \max, \times, 0, 1)$;
- *weighted* c-semiring $\mathbb{W} = (\mathbb{R}_{\geq} \cup \{\infty\}, \min, +, \infty, 0)$.

Every c-semiring admits an order $\leq$ defined by $r \leq s$ iff $r + s = s$. It is shown in [22] that $\leq$ satisfies the following properties:

1. $\leq$ is a partial order, with minimum 0 and maximum 1;

2. $x + y$ is the least upper bound of x and y;
3. $x \times y$ is a lower bound of x and y;
4. $(S, \leq)$ is a complete lattice (i.e., the greatest lower bound exists);
5. $+$ and $\times$ are monotone on $\leq$.
6. if $\times$ is idempotent, then $+$ distributes over $\times$, $x \times y$ is the greatest lower bound of x and y, and $(S, \leq)$ is a distributive lattice.

The composability of actions and their resulting composition is defined in [45] with a Component Action System (CAS). A CAS can be lifted to an interaction signature on TESs by using, for instance, the synchronous composability relation defined in Chapter 2.

Definition 33 (Component action system)**.** *A* component action system (CAS) *is a tuple* $\langle \Sigma, \odot, \boxdot \rangle$*, such that* Σ *is a finite set of* actions*,* $\odot \subseteq \Sigma \times \Sigma$ *is a reflexive and symmetric relation and* $\boxdot : \odot \to \Sigma$ *is an idempotent, commutative and associative* $\odot$*-operator on* Σ*. We call* $\odot$ *the* composability relation*, and* $\boxdot$ *the* composition operator*.*

A Soft Component Automaton (SCA) is a finite characterization of an agent behavior, equipped with a strategy. The csemiring value labeling each transition induces a partial order, for each state, on the set of outgoing transitions. An agent may therefore filter its behavior by allowing only the k best actions from the partially ordered set of outgoing transitions.

Definition 34 (Soft component automaton)**.** *A* soft component automaton (SCA) *is a tuple* $\langle Q, \Sigma, \mathbb{E}, \to, q^0, t \rangle$ *where* Q *is a finite set of* states*, with* $q^0 \in Q$ *the* initial state*,* Σ *is a CAS, and* $\mathbb{E}$ *is a c-semiring, with* $t \in \mathbb{E}$ *the* threshold*. Lastly,* $\to \subseteq Q \times \Sigma \times \mathbb{E} \times Q$ *is a finite relation called the* transition relation*. We write* $q \xrightarrow{a,\, e} q'$ *when* $\langle q, a, e, q' \rangle \in \to$*.*

The threshold determines which actions have sufficient preference for inclusion in the behavior. Intuitively, the threshold is an indication of the amount of flexibility allowed. In the context of composition, lowering the threshold of a component is a form of compromise: the component potentially gains behavior available for composition. Setting a lower threshold makes a component more permissive, but may also make it harder (or impossible) to achieve its goal.

Definition 35 (Behavior of an SCA)**.** *Let* $A = \langle Q, \Sigma, \mathbb{E}, \to, q^0, t \rangle$ *be an SCA. We say that a stream* $\sigma \in \Sigma^\omega$ *is a* behavior *of* A *if there exist streams* $\mu \in Q^\omega$ *and* $\nu \in \mathbb{E}^\omega$

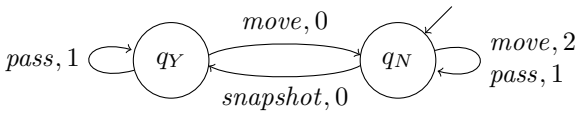


Figure 4.1: A component modeling the desire to take a snapshot at every location, A_s.

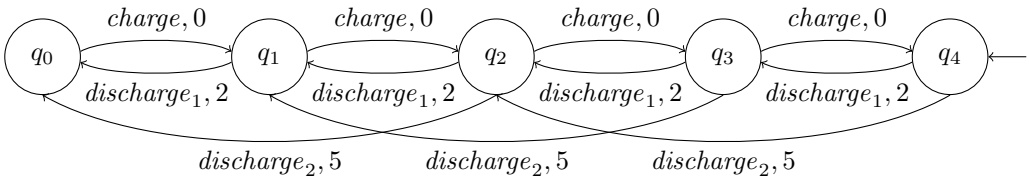


Figure 4.2: A component modeling energy management, A_e.

such that $\mu(0) = q^0$*, and for all* $n \in \mathbb{N}$*, we have* $t \leq \nu(n)$ *and* $\mu(n) \xrightarrow{\sigma(n),\ \nu(n)} \mu(n+1)$*. The set of behaviors of* A*, denoted by* $L(A)$*, is called the* language *of* A*.*

To discuss an example of SCA, we introduce the SCA A_s in Figure 4.1, which models the crop surveillance drone's objective to take a snapshot of every location before moving to the next. The CAS of A_s includes the pairwise incomposable actions *pass*, *move* and *snapshot*, and its c-semiring is the weighted c-semiring $\mathbb{W}$. We leave the threshold value t_s undefined for now. The purpose of A_s is reflected in its states: q_Y (resp. q_N) represents that a snapshot of the current location was (resp. was not) taken since moving there. If the drone moves to a new location, the component moves to q_N, while q_Y is reached by taking a snapshot. If the drone has not yet taken a snapshot, it prefers to do so over moving to the next spot (missing the opportunity).

Another example of an SCA is A_e, drawn in Figure 4.2; its CAS contains the incomposable actions *charge*, $discharge_1$ and $discharge_2$, and its c-semiring is the weighted c-semiring $\mathbb{W}$. This particular SCA can model the component of the crop surveillance drone responsible for keeping track of the amount of remaining energy in the system; in state q_n (for $n \in \{0, 1, \ldots, 4\}$), the drone has n units of energy left, meaning that in states q_1 to q_4, the component can spend one unit of energy through $discharge_1$, and in states q_2 to q_4, the drone can consume two units of energy through $discharge_2$. In states q_0 to q_3, the drone can try to recharge through *charge*. Recall that, in $\mathbb{W}$, higher values reflect a lower preference (a higher *weight* or *cost*); thus, *charge* is preferred over $discharge_1$.

Here, A_e is meant to describe the possible behavior of the energy management component only. Availability of the actions within the *total model* of the drone (i.e.,

the composition of all components) is subject to how actions compose with those of other components; for example, the availability of *charge* may depend on the state of the component modeling position. Similarly, preferences attached to actions concern energy management only. In states q_0 to q_3, the component prefers to top up its energy level through *charge*, but the preferences of this component under composition with some other component may cause the composed preferences of actions composed with *charge* to be different. For instance, the total model may prefer executing an action that captures $discharge_2$ over one that captures *charge* when the former entails movement and the latter does not, especially when survival necessitates movement.

Nevertheless, the preferences of A_e affect the total behavior. For instance, the weight of spending one unit of energy (through $discharge_1$) is lower than the weight of spending two units (through $discharge_2$). This means that the energy component prefers to spend a small amount of energy in a single step. This reflects a level of care: by preferring small steps, the component hopes to avoid situations where too little energy is left to avoid disaster.

Reo as a DSL We define a domain specification language for finite state preference aware agents as a subset of Reo. One reason for using Reo is its graphical syntax, which gives an intuitive encoding of soft component automata in terms of graphical connectors and interaction primitives. Moreover, Reo reflects the modular and compositional aspects that make SCAs suitable for specifying complex behaviors: connectors compose into more complex connectors, just like how SCAs compose into more complex SCAs. We take advantage of this feature and, after defining an encoding of SCAs into Reo connectors, we represent the composition of SCAs as the composition of their corresponding connectors. Another reason is the existence of a compilation chain that makes it possible to compile the same Reo model to an execution language (such as Java or C) or to a language that supports verification (such as the rewriting logic language Maude [13]). Effective optimizations implemented in the current Reo compiler help to keep the size of resulting composed models manageable, yielding similarly manageable models in Maude, Java, etc.

Some existing research has considered the question of synthesizing Reo circuits for constraint automata [13]. In our work, similar channels are used for encoding the structure of the automaton (syncfifo, xrouter, and merger), but a new channel, the bfilter, is introduced to encode the soft part of the action labeling transitions of SCAs. Moreover, we provide, along with the description, the representation of the Reo connector in a textual language, used as input for the compiler developed in [60].

We propose a general approach to represent SCAs and their composition as Reo circuits. Recall that, by Definition 34, an SCA is formally defined as a tuple $\langle Q, \Sigma, \mathbb{E}, \rightarrow, q^0, t\rangle$ where Q is the set of states, Σ a component action system, $\mathbb{E}$ a c-semiring, $\rightarrow$ a transition relation, $q^0 \in Q$ the initial state, and $t \in \mathbb{E}$ is the threshold value of the SCA. In the sequel, we give a procedure to write an SCA as a Reo circuit. The set of connectors defined hereafter constitutes a domain specific fragment of Reo for building SCA. We conclude this section with an example of composition of two SCAs obtained through composition of their respective Reo representations.

Actions and c-semirings Given Σ a CAS of an SCA, we map each action $a \in \Sigma$ into a Reo port with the same name. We consider the SCA "doing action a" equivalent to "firing of port a". Given the threshold $t \in \mathbb{E}$, we associate each c-semiring value $e \in \mathbb{E}$ with a predicate $P_t(e)$ whose semantics reflects the truth value of $t \leq e$ in the c-semiring. In order to mirror the semantics defined previously for composition of SCA, the c-semiring value and the threshold value of a predicate may change during composition. We consider the c-semiring to be fixed and shared by all SCAs.

States We define a state of an SCA as a Reo circuit, which we then graphically abbreviate as a user-defined node. Essentially, a state is mapped into a syncfifo channel, the empty/full status of whose buffer reflects whether or not the SCA is currently in that state. As depicted in the circuit below, we identify the source end of the syncfifo with the name of the state. Thus, to be in state q of the SCA corresponds to the syncfifo whose source end is q being full. The initial state q^0 starts with a full syncfifo buffer; the syncfifo buffers of all other states start empty. Intuitively, all incoming $(\mathsf{i}_1, \ldots, \mathsf{i}_n)$ transitions into a state q, merge at the source end of the syncfifo, and all outgoing transitions $(\mathsf{o}_1, \ldots, \mathsf{o}_m)$ out of q synchronize via mutual exclusion with one another on the sink end of the syncfifo. The reason for using the syncfifo instead of the standard fifo primitive is that an outgoing transition can also be an incoming transition into the same state, i.e., allow get and put operations on its ends to synchronously empty and fill its buffer. We use an n-ary *exclusive router* to express that only one outgoing transition is taken from a state with n outgoing transitions. The n-ary xrouter can be constructed out of a ternary xrouter.

We call our constructed circuit a *state*, and use ◎ to represent a state of an SCA as a graphical abbreviation and present it as a user-defined node in Reo with n inputs and m outputs. We use ⦿ as graphical abbreviation for the Reo circuit that corresponds to the initial (and current) state of an SCA.

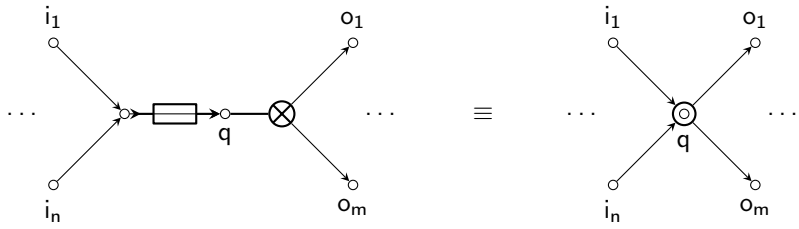


Figure 4.3: Graphical abbreviation for a state.

```
State(qi[1..n],qo[1..m]) {
  { sync(qi[k],x)  |  k:<1..n> }
  syncfifo1<"0">(x,y)
  xrouter(y,qo[1..m])
}
```

Listing 4.1: Component defining a state q in textual Reo.

Besides the graphical construct for a state, we introduce a State connector in the textual language of Reo shown in Listing 4.1. We adopt a convention, and prefix the input and output ports of a state with the name of the state. For instance, the component State(q0i[1..n], q0o[1..m]) represents the state q^0 with n incoming transitions and m outgoing transitions. We refer to the k-th incoming, resp. k-th outgoing, transition to state q0 with the port q0i[k], respectively q0o[k].

Listing 4.1 shows an example of a component defined using conditional set notation. The number of input ports in the interface of the State component influences how its body is instantiated. The variable k ranges over the list $[1, .., n]$, and thus creates a set of sync channels.

Transitions A transition in an SCA involves an action, a c-semiring value, a pre-state and a post-state. When the transition is enabled (i.e., its c-semiring value is above the threshold), the transition synchronously fires the action port, and moves the SCA from its pre-state to its post-state. We model this behavior in Reo as the circuit in Figure 4.4, which represents the conditional activation of a transition using a blocking-filter channel that compares the c-semiring value of the transition with the threshold of the SCA. Given a c-semiring value e, the predicate $P_t(e)$ of the blocking-filter channel is true if and only if the c-semiring value e is greater than or equal to the threshold value t.

The circuit in Figure 4.4 moves the token from node q_0 to node q_1 and fires port

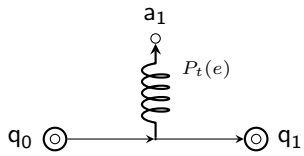


Figure 4.4: Reo circuit for a transition of a soft component automaton.

```
Transition<e,t>(q0,q1,a1) {
  sync(q0,x)
  bfilter<e,t>(x,a)
  sync(x,q1)
}
```

Listing 4.2: Component defining a transition in textual Reo.

a_1, only if $P_t(e)$ is true. If $P_t(e)$ is not satisfied, the circuit in Figure 4.4 blocks the transfer of the token from q_0 to q_1, mirroring the fact that its corresponding SCA transition cannot be taken.

The transition primitive in textual Reo is written in Listing 4.2. The transition component takes three ports in its interface, q_0 and q_1, being respectively the pre-state and post-state, and a_1 being the action. Two values are provided as parameters to a transition component: the c-semiring value e, and the threshold value t. Internally, the transition component connects the pre-state to the post-state through synchronization with the bfilter. The bfilter takes a c-semiring value of a given type as parameter, and performs internal comparison with the threshold value.

Soft component automata Given the constructs for states and transitions, we can build a Reo circuit for every SCA. For instance, the circuit for the automaton in Figure 4.1 is shown in Figure 4.5. The two states q_Y and q_N are represented as two state-nodes, with q_N initially full (designating it as the initial state).

To avoid visual clutter, we repeat the names of ports in the circuit (e.g., a_move appears twice in Figure 6), but all occurrences of the same port name correspond to a single, unique port. Each of the five transitions of A_s is an instance of the transition component in Reo. For example, the *move* transition from q_Y to q_N is represented by the transition connector with input from the state q_Y, output from the state q_N, blocking filter with predicate $P_t(e_1)$, and action port a_move. The corresponding component view of the automaton is represented by a box that abstracts away the details of its Reo circuit, exposing as its interface the boundary ports on which other components

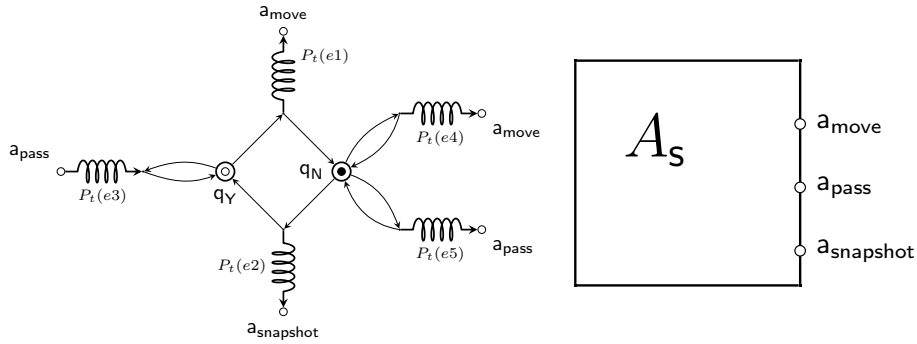


Figure 4.5: Reo circuit for the Snapshot SCA.

```
As<t>(move,pass,snap) {
  Transition<1,t>(qYo[1],qYi[1],pass)
  Transition<0,t>(qYo[2],qNi[1],move)
  Transition<0,t>(qNo[1],qYi[2],snap)
  Transition<2,t>(qNo[2],qNi[2],move)
  Transition<1,t>(qNo[3],qNi[3],pass)

  State(qYi[1..2],qYo[1..2])     //State qY
  State(qNi[1..3],qNo[1..3])       //State qN
}
```

Listing 4.3: Component defining the snapshot SCA in textual Reo.

can synchronize.

The snapshot SCA A_e is built out of the State and Transition connectors in Reo defined in Listings 4.1 and 4.2. We show the instance of the Snapshot SCA A_s in Listing 4.3, and adopt the convention defined previously to denote ports of incoming and outgoing transitions.

Component action system The composition of two SCAs can also be written as a Reo circuit, by encoding the composed SCA. However, such an approach uses the SCA composition and disregards the compositional nature of Reo. Instead, we propose to encode each individual SCA as a Reo circuit, and then compose those encodings on the level of Reo, to obtain a Reo circuit equivalent to their composed automaton. This approach allows for a transparent and incremental translation.

Since composition on the level of SCAs is mediated by their (common) CAS, composition at the level of Reo should also take the CAS into account. To do this, we

encode the CAS as a Reo circuit of its own; composition of two automata at the level of Reo is then given by the (Reo) composition of their individual encodings, together with the circuit obtained from their CAS. Furthermore, we hide all ports that are not output ports of the CAS after the composition, so that the only actions observable in the resulting Reo circuit are the actions that are brokered between the operand circuits by the CAS.

There are three "sides" (collections of ports) to a CAS component: one for each of the two operands in the composition, respectively called the *left* and the *right (operand) side*, and a *composite side* for the result of the composition. For each action α, we add three ports to the circuit, one in each side, labeled α_ℓ, α_r and α_c for the left, right and composite sides respectively. The ports on the operand sides are input ports, and the ports on the composite side are output ports.

The intention of the circuit structure is as follows. If the operand circuits are ready to perform actions α and β respectively, then ports α_ℓ and β_r will be enabled for writing. If $\alpha \odot \beta$, then the CAS circuit brokers their composition, by allowing α_ℓ and β_r to fire simultaneously, synchronously firing the port that represents their composition in the composite side, i.e., $(\alpha \boxdot \beta)_c$, as well. Moreover, the circuit ensures that firing two ports in the left and right sides (when permitted) gives rise to exactly one port firing in the composite side.

More formally, the circuit is built as follows. On the operand sides, each port α_o (where $o \in \{\ell, r\}$) is connected to an exclusive router labeled α_o^R. For each pair of actions in the left and right operand sides that are compatible, i.e., all $\alpha, \beta \in \Sigma$ such that $\alpha \odot \beta$, we draw a synchronous drain from α_ℓ^R and β_r^R to an internal node labeled $\alpha\beta$. Each of these nodes is then connected through a syncspout channel to the composite side node labeled $(\alpha \boxdot \beta)_c$.

The CAS defined for the SCAs A_e and A_s is depicted in Figure 4.6. In this example, the exclusive router has a single output, and is not strictly necessary. In general, the CAS could define multiple composite actions out of the same side action. For instance, suppose that the drone in our example is equipped with solar panels, and that the net result of charging using the solar panels while moving is that the energy level does not change. As a result, the energy component's action pass is compatible with the action move, and their composition is the action solar, which means "move with energy from the solar panels". Note how in this scenario, the firing of move_ℓ can occur only in conjunction with firing $\mathsf{discharge}_{2r}$ or pass_r, but not both; in the first case, the composite interface port move_{2c} fires, while in the second case the port solar_c fires.

We give in Listing 4.4 the corresponding Reo component for the CAS described in

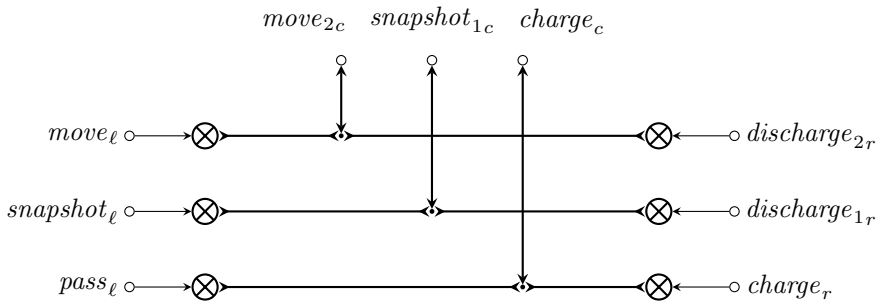


Figure 4.6: Partial encoding of a CAS.

```
cas(move,pass,snap,dchge1,dchge2,
    chge,move2,charge,snapshot1){
  syncdrain(move,x)  syncspout(x,move2)
  syncdrain(dchge2,x)
  syncdrain(pass,y) syncspout(y,charge)
  syncdrain(chge,y)
  syncdrain(snap,z) syncspout(z,snapshot1)
  syncdrain(dchge1,z)
}
```

Listing 4.4: Component defining the CAS for the composition of A_e and A_s in textual Reo

Figure 4.6 for the composition of the snapshot SCA and the energy SCA. We omitted the exclusive routers, since, in this case, they are not necessary.

Composition The Reo circuit corresponding to a composition of two soft component automata can now be defined as the composition of the Reo circuits for the individual soft component automata, together with the Reo circuit for the relevant component action system. Following the method above, we translate each of A_s and A_e, respectively representing the snapshot component and the energy management component, into its respective Reo connector.

Based on the steps described above, it is now possible to define a Reo circuit for both A_e and A_s, that we name respectively `Ae` and `As` in textual Reo. The resulting composition, shown in Listing 4.5, consists of a set containing the connector for each SCA together with the connector for the component action system. The two thresholds values are provided as parameter.

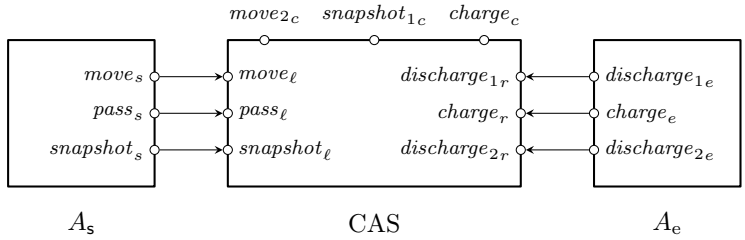


Figure 4.7: Composition of two component automata with their component action system.

```
composite(move2,charge,snapshot1) {
  Ae<t1>(move,pass,snap)
  cas(move,pass,snap,dchge1,dchge2,
      chge,move2,charge,snapshot1)
  As<t2>(dchge1,dchge2,chge)
|
  t1 = 5,
  t2 = 3
}
```

Listing 4.5: Component in textual Reo defining the composition between A_e and A_s.

4.4 Related work and future work

Real-time Maude Real-Time Maude is implemented in Maude as an extension of Full Maude [68], and is used in applications such as in [53]. There are two ways to interpret a real-time rewrite theory, called the pointwise semantics and the continuous semantics. Our approach to model time is similar to the pointwise semantics for real-time Maude, as we fix a global time stamp interval before execution. The addition of a composability relation, that may discard actions to occur within the same rewrite step, differs from the real-time Maude framework.

Models based on rewriting logic In [21], the modeling of cyber-physical systems from an actor perspective is discussed. The notion of event comes as a central concept to model interaction between agents. Softagents [78] is a framework for specifying and analyzing adaptive cyber-physical systems implemented in Maude. It has been used to analyze systems such as vehicle platooning [30] and drone surveillance [61]. In Softagents agents interact by sharing knowledge and resources implemented as part of the system timestep rule.

Softagents only considers compatibility in the sense of reachability of desired or undesired states. Our approach provides more structure enabling static analysis. Our framework allows, for instance, to consider compatibility of a robot with a battery (i.e., changing the battery specification without altering other agents in the system), and coordination of two robots with an exogenous protocol, itself specified as an agent.

Algebra, co-algebra The algebra of components described in this paper is an extension of [57]. Algebra of communicating processes [35] (ACP) achieves similar objectives as decoupling processes from their interaction. For instance, the encapsulation operator in process algebra is a unary operator that restricts which actions may occur, i.e., $\delta_H(t \parallel s)$ prevents t and s to perform actions in H. Moreover, composition of actions is expressed using communication functions, i.e., $\gamma(a, b) = c$ means that actions a and b, if performed together, form the new action c. Different types of coordination over communicating processes are studied in [20].

Discrete Event Systems Our work represents both cyber and physical aspects of systems in a unified model of discrete event systems [67, 5]. In [51], the author lists the current challenges in modelling cyber-physical systems in such a way. The author points to the problem of modular control, where even though two modules run without problems in isolation, the same two modules may block when they are used

in conjunction. In [74], the authors present procedures to synthesize supervisors that control a set of interacting processes and, in the case of failure, report a diagnosis. An application for large scale controller synthesis is given in [66]. Our framework allows for experiments on modular control, by adding an agent controller among the set of agents to be controlled. The implementation in Maude enables the search of, for instance, blocking configurations.

Timed Automaton Several operations on Timed Automata have been defined to model different aspects of concurrency. The UPPAAL modeling language allows such concurrent operations, and the UPPAAL tool computes the product automaton on the fly during verification.

It is shown that reachability is decidable, and it is proven that the infinite state-space of timed automata can be finitely partitioned into symbolic states using clock constraints known as zones.

UPPAAL is a tool in which network of timed automata are considered. Similarly to the case of a single timed automata, two types of transitions are considered: delay transitions, and action transitions. The difference is that action transitions decline into two kinds: single action transitions, and synchronous action transitions.

UPPAAL makes use of CTL formulas, that are dynamically verified on the tree unfolding of the transition system, making use of the zone optimization. UPPAAL is well-suited for timed automata but has some limitations in the support of hybrid automata, e.g. restricting their continuous parts to simple dynamics or applying the Euler integration method.

Chapter 5

Experimental framework

The component framework introduced in Chapter 2 and its operational fragments defined in Chapter 4 lay the foundation for the verification of properties of cyber-physical systems.

In this chapter, we detail and evaluate an implementation in Maude of the cyber-physical agent framework introduced in Section 4.2 of Chapter 4. This implementation extends the operational formal model with three additional features. First, an agent is equipped with an internal strategy, similar to the one introduced in Section 4.3. Thus, the Maude implementation enables two levels to make a preference aware system more specific: by selecting a subset of best actions either at the agent level, or at the system level. Second, an agent may perform a composite action *atomically*, i.e., a sequence of actions within the same clique. As a result, the value assigned to action parameters may depend on the effects of actions earlier in the sequence. The Maude implementation enables agents to reevaluate the parameters of their actions at runtime. Third and last, we give some constraints on how a set of atomic actions, called *macrostep*, is serialized into a sequence of *microsteps*, i.e., a sequence of agent actions. More precisely, we impose that such serialization is a function given as parameter for simulation or analysis. The runtime may still be non-deterministic, as several different cliques may be enabled at the same time. The above three features are detailed in Section 5.1.

We use our implementation to simulate and analyze a series of applications. More precisely, we use the Maude runtime to verify trace properties of concurrent systems. Recall that, as defined in Chapter 2, a trace property is a set of TESs, and a component C satisfies a property P, denotes as $C \models P$, if and only if the behavior of C is a subset

of the property P. In Section 2.2.2, we introduced the notion of conformance, which states that a component C is conformable to a component C' if there exists a non empty protocol P such that $C \times_\Sigma P \sqsubseteq C'$.

In the agent framework, a system is a set of interacting agents for which we gave in Section 4.2 a compositional semantics as components. Therefore, given a set of n agents $\mathcal{A}_1(s_{01}, t_0), ..., \mathcal{A}_n(s_{0n}, t_0)$ interacting under the interaction signature Σ, we say that the system $\mathcal{S} = \mathcal{A}_1(s_{01}, t_0) \times_\Sigma ... \times_\Sigma \mathcal{A}_n(s_{0n}, t_0)$ satisfies property P if the component semantics does so, i.e., if $[\![\mathcal{S}]\!] \models P$ (see Theorem 9 for soundness). In the case where $[\![S]\!] \not\models P$, we then identify two mechanisms to make $[\![S]\!]$ satisfy P.

The most obvious way is to substitute each of a subset of the agents $\mathcal{A}_i$ with a more specific agent $\mathcal{A}'_i$ such that the resulting system satisfies P. Such agent $\mathcal{A}'_i$ is more specific than agent $\mathcal{A}_i$ due to its smaller state space and behavior (where all TESs that violate property P have been removed). Finding the largest of such $\mathcal{A}'_i$ is therefore of primary importance to keep as much as possible the non-violating TESs from the behavior of agent $\mathcal{A}_i$.

An alternative way is to synthesize a coordinator agent D such that $[\![\mathcal{S} \times_\Sigma D]\!] \models P$. While similar to the first method, as it generates a system $\mathcal{C}' = \mathcal{S} \times_\Sigma D$, the coordinator D is a separate entity that can therefore be modified. In the case of a system $\mathcal{A}_1 \times_\Sigma \mathcal{A}_2$, composite of agents $\mathcal{A}_1$ and $\mathcal{A}_2$, that does not satisfy a property P, a coordinator may filter actions from $\mathcal{A}_1$ and $\mathcal{A}_2$ contextually, i.e., imposing a relation between actions occurring in $\mathcal{A}_1$ and $\mathcal{A}_2$.

We instantiate the framework to model diverse applications:

- cyber agents interacting via Reo coordination protocols. We give an implementation of a Reo nodes, primitive channels, and show how to compose protocols.

- N reservoirs, a valve, and a controller. We explore how the controller can control the valve to maintain a safety property, such as that the N reservoirs always have a water level within some thresholds.

- two robot agents, each interacting with a (shared) field and a (private) battery agent. We explore how safety, liveness, and coordination properties are enforced by a system of agents. For instance, such system is energy safe if the battery levels always stay above some thresholds. The system is alive if the agents keep patrolling between two locations on the field. Finally, coordination properties are such that agents eventually sort themselves on the grid by correctly exchanging their locations.

We run some analysis on the system using the Maude reachability search engine.

5.1 Maude framework for cyber-physical agents

The Maude framework is an instance of the general agent framework introduced in Chapter 4. We therefore introduce the Maude implementation for *actions*, *agents*, *system of agents*, and *composability relation*. The implementation is accessible at [54].

Actions An action is a pair that contains the name of the action, and the set of agent identifiers on which the action applies. An agent action is identified by the source agent identifier, and is a triple `(id, (a; ids))` where `id` is the agent doing the action with name `a` onto the set of agents `ids`, that we call resources of agent `id` for action named `a`.

```
fmod ACTION is
    inc STRING . inc BOOL . inc SET{Id} . ...
    sort AName Action AgentAction .
    op (_;_) : AName Set{Id} -> Action [ctor] .
    op (_,_) : Id Action -> AgentAction [ctor] .
    op mta : -> AgentAction .
endfm
```

Agent The `AGENT` module in Listing 5.1 defines the theories on which an agent relies, the `Agent` sort, and operations that an agent instance must implement. The module is parametrized with a `CSEMIRING` theory, that is used to rank actions of an agent. Additionally, the `AGENT` includes modules that define state and action terms. A term of sort `IdStates` is a pair of an identifier and a map of sort `MapKD`.

A term of sort `Agent` is a tuple `[id: C| state; ready?; softaction]`. The identifier `id` is unique for each agent of the same class `C`. The state `state` of an agent is a map from keys to values. For instance, the state of a robot has three keys, `position`, `energy`, and `lastAction`, with values in `Location`, `Status`, and `Bool`. The flag `ready?` is of sort `Bool` and is `True` when the agent has submitted a possibly empty list of actions, and `False` otherwise. The pending actions `softaction` is a set of actions valued in the parametrized `CSEMIRING`. The use of a constraint semiring as a structure for action valuations enables various kinds of reasoning about preferences at the agent and system levels. We use the two operations of the csemiring, sum $+$ and product $\times$, as respectively modeling the choice and the compromise of two alternatives. See [82, 78, 45] for more details.

As shown in Listing 5.1, an agent instance implements four operations: `computeActions`, `resolve`, `getOutput`, `getPostState`, and `internalUpdate`. Note that the four operations are an implementation of the abstract ϕ of Section 4.2.1. The operation `computeActions`, given a `state:MapKD` of agent `id` of class `C`, returns a set of valued actions in the parametrized `CSEMIRING`. The operation `internalUpdate`, given a `state:MapKD` of agent `id` of class `C`, returns a new state `state':MapKD`. For instance, an agent may record in its state, as an internal update, the outcome of `computeActions` that returns the set of possible actions for the agent. The `getOutput` operation, given an action name `a:Name` from agent identified by `id2` applied to an agent `id` of class `C` in a state `state`, returns a collection of outputs. The outputs generated by `getOutput` are of sort `MapKD` and therefore structured as a mapping from keys to values. For instance, the output of the action named `read` applied on a `FIELD` agent has a key `pos` that maps to the position value of the agent doing the `read` action. The operation `getPostState`, given an action name `a:AName` with inputs `input:IdStates` from agent identified by `id2` applied on an agent `id1` of class `C` in a state `state`, returns a new state. The input `input:IdStates` is a collection of key to value mappings that results from collecting the outputs, i.e., with `getOutput`, of an action `(id, an, ids)` on all its resources in `ids`. The new state returned by `getPostState` describes how the agent reacts to the input action, which could also capture, with an error state, that the action is not allowed by the agent. The operation `resolve`, given an action name `a:Name` performed by an agent with identifier `id` with class `C` in state `state`, returns a new action name `a':Name`. The `resolve` operation is called before the input action name is performed, and may instantiate some parameters of the action given the state of the agent.

Listing 5.1: Extract from the `AGENT` Maude module.

```
fmod AGENT{X :: CSEMIRING} is
 inc IDSTATE .  inc ACTION .
 sort Agent .
 op [_:_|_;_;_] : Id Class MapKD Bool X$Elt -> Agent [ctor].
 op computeActions : Id Class MapKD -> X$Elt .
 op internalUpdate : Id Class MapKD -> MapKD .
 op getPostState : Id Class Id AName IdStates MapKD -> MapKD
 op getOutput : Id Class Id AName MapKD -> MapKD .
 op resolve : AName Identifier Class MapKD -> AName .
endfm
```

The agent's dynamics are given by the rewrite rule in Listing 5.2, that updates the pending action to select one atomic action from the set of valued actions:

Listing 5.2: Conditional rewrite rule applying on agent terms.

```
crl[agent] : [sys [id : ac | state  ; false ; null]] =>
      [sys [id : ac | state' ; true  ; softaction]]
    if softaction + sactions := computeActions(id, ac, state)
       /\ state' := internalUpdate(id, ac, state) .
```

The rewrite rule in Listing 5.2 implements the abstract rule of Equation 4.2. After application of the rewrite rule, the `ready?` flag of the agent is set to `True`. The agent may, as well, perform an internal update independent of the success of the selected action.

Moreover, an agent module comes with the definition of an interface. The interface for an agent contains the constructors for action names, i.e., `move: direction -> Action` and `read: sensorName -> Action` for a TROLL robot agent. An agent that interacts with another agent must therefore include the interface module of that agent, i.e., the set of actions that the agent performs.

System The `SYSTEM` module in Listing 5.3 defines the sorts and operations that apply on a set of agents. The sort `Sys` contains set of `Agent` terms, and the term `Global` designates top level terms on which the system rewrite rule applies (as shown in Listing 5.4). The `SYSTEM` module includes the `Agent` theory parametrized with a fixed semiring `ASemiring`. The theory `ASemiring` defines valued actions as pairs of an action and a semiring value. While we assume that all agents share the same valuation structure, we can also define systems in which such a preference structure differs for each agent. The `SYSTEM` module defines four operations: `linearization`, `outputFromAction`, `updateSystemFromAction`, and `updateSystem`. The operation `linearization` returns a list of `AgentAction` given a set of `AgentAction`. As there are multiple ways to generate sequences of actions from a set of actions, we assume a total order among actions and a sorted output sequence. As several total orders may exist, we leave the equational specification of the linearization operation in each scenario. The operation `outputFromAction` returns, given an agent action `(id, (an, ids))` applied on a system `sys`, a collection of identified outputs given by the union of the results of `getOutput` produced by all agents in `ids`. The operation `updatedSystemFromAction` returns, given an agent action `(id, (an, ids))` applied on a system `sys`, an updated system `sys'`. The updated system may raise an error if the action is not allowed by some of the resource agents in `ids` (see the battery-field-robot example in 5.4). The updated system, otherwise, updates *synchronously* all agents with identifiers in `ids` by using the `getPostState` operation. The operation

`updateSystem` returns, given a list of agent actions `agentActions` and a system term `sys`, a new system `updateSystem(sys, agentActions)` that performs a sequential update of `sys` with every action in `agentActions` using `updatedSystemFromAction`. The list `agentActions` ends with a delimiter action `end` performed on every agent, which may trigger an error if some expected action does not occur (see `PROTOCOL` in Section 5.4).

Listing 5.3: Extract from the `SYSTEM` Maude module.

```
fmod SYS is
  inc AGENT{ASemiring} . sort Sys  Global .
  subsort Agent < Sys . op [_] : Sys -> Global [ctor] .
  op __ :  Sys Sys -> Sys [ctor assoc comm id: mt] .   ...
  op linearization : Set{AgentAction} -> List{AgentAction} .
  op outputFromAction : AgentAction Sys -> IdStates .
  op updatedSystemFromAction : AgentAction Sys -> Sys .
  op updateSystem : Sys List{AgentAction} -> Sys .
endfm
```

The rewrite rule in Listing 5.4 applies on terms of sort `Global` and updates each agent of the system synchronously, given that their actions are composable. The rewrite rule in Listing 5.4 implements the abstract rule of Equation 4.4. The rewrite rule is conditional on essentially two predicates: `agentsReady?` and `kbestActions`. The predicate `agentsReady?` is `True` if every agent has its `ready?` flag set to `True`, i.e., the agent rewrite rule has already been applied. The operation `kbestActions` returns a ranked set of cliques (i.e., composable lists of actions), each paired with the updated system. The element of the ranked set are lists of actions containing at most one action for each agent, and paired with the system resulting from the application of `updateSystem`. If the updated system has reached a `notAllowed` state, then the list of actions is not composable and is discarded. The operations `getSysSoftActions` and `buildComposite` form the set of lists of composite actions, from the agent's set of ranked actions, by composing actions and joining their preferences.

Listing 5.4: Conditional rewrite rule applying on system terms.

```
crl[transition] : [sys]  => [sys']
 if agentsReady?(sys)  /\ saAtom := getSysSoftActions(sys) /\
  saComp := buildComposite(saAtom , sizeOfSum(saAtom)) /\
  p(actseq, sys') ; actseqs := kbestActions(saComp, k, sys) .
```

Composability relation The term `saComp` defines a set of valued lists of actions. Each element of `saComp` possibly defines a clique. The operation `kbestActions` spec-

ifies which, from the set `saComp`, are cliques. We describe below the implementation of `kbestActions`, given the structure of action terms.
An action is a triple `(id, (an, ids))`, where `id` is the identifier of the agent performing the action `an` on resource agents `ids`. Each resource agent in `ids` reacts to the action `(id, (an, ids))` by producing an output `(id', an, O)` (i.e., the result of `getOutput`). Therefore, $\texttt{comp}((\texttt{id}, (\texttt{an}, \texttt{ids})), \texttt{a}_\texttt{i})$ holds, with $\texttt{a}_\texttt{i} : \texttt{Action}_\texttt{i}$ and $i \in \texttt{ids}$, only if $\texttt{a}_\texttt{i}$ is a list that contains an output `(i, an, O)`, i.e., an output to the action. If one of the resources outputs the value `(i,notAllowed(an))`, the set is discarded as the actions are not pairwise composable. Conceptually, there are as many action names `an` as possible outputs from the resources, and the system rule (4.2) selects the clique for which the action name and the outputs have the same value. In practice, the list of outputs from the resources get passed to the agent performing the action.

5.2 Concurrent Reo

In Chapter 3, we use our component algebra as a semantic model for Reo. More particularly, we introduce nodes as primitive components, and channels or connectors as the composition of ports under some fixed composition operators. In this section, we use our Maude implementation to provide a concurrent implementation of Reo connectors, as a set of interacting agents.

5.2.1 Reo primitives as agents

We refer to Section 3.1 for an introduction to Reo. The available implementation [60] of Reo focuses on compilation of input circuits to an executable. A remaining challenge was to construct a compositional runtime where each part of a Reo circuit can be compiled independently and run concurrently. As a consequence of such framework, one can keep the structure of a Reo circuit at runtime, mix different semantics for Reo channels (e.g., guarded commands, constraint automata, ...), while allowing for simulation and verification through reachability queries.

Our framework for concurrent Reo consists therefore of few primitive agents: `PORT`, `CHANNEL`, `CONNECTOR`.

Port as resource A port is a point of synchronization in Reo, and its typical behavior is to forward atomically data from its input connector to its output connector. We fix a port identifier to be of the form `P(i)` where `i:Float` is a rational number.

We later use the identifier of a port to define the operation of `linearization` and order actions.

A port contains a structure with two buffers that implement the atomic passing of data from the input to the output connector. One buffer collects the data that the input connector may *put*, and the other contains the request from the output connector to *take* some data. Only when the two buffers are full, as shown with the error raised by the `end` action, should the port allow both `put(d)` and `take` actions.

```
fmod PORT is
  inc AGENT{ASemiring} .
  inc PROTOCOL-INTERFACE .
  inc PORT-INTERFACE .
    ...
  eq computeActions(id, Port, M ) = null .

  ceq getPostState(r, Port, id, take, mtOutput, M) = M'
    if M[k("data")] =/= nodata /\
       M' := insert(k("sync"), bd(true), M ) .

  ceq getPostState(r, Port, id, put(data), mtOutput, M) = M'
    if M[k("data")] == nodata /\
       M' := insert(k("data"), data, M ) .

  ceq getPostState(id, Port, id, end, inputs, M) = M'
    if M[k("data")] =/= nodata /\ M[k("sync")] == bd(true) /\
       M' := insert(k("data"), nodata,
                 insert(k("sync"), bd(false), M)) .
  ceq getPostState(id, Port, id, end, inputs, M) =
        notAllowed(end)
    if M[k("data")] =/= nodata or M[k("sync")] == bd(true) .

  eq getPostState(r, Port, id, a, inputs, M) = M [owise] .

  ceq getOutput(r, Port, id, take, M) =
    k("data") |-> M[k("data")] if M[k("data")] =/= nodata .

  eq getOutput(r, Port, id', a, M) = empty [owise] .

  eq internalUpdate(id, Port, M) = M [owise] .

endfm
```

Connectors as agents We give three instances of primitive Reo connectors: a `SYNC`, a `FIFO`, and a `MERGER`. Other primitive connectors, such as a replicator, syncdrain, etc, could be defined similarly. While we do not expand on this point here, some parametric connectors such as alternator(n) could be defined recursively as well, making use of the port naming structure.

A `SYNC` agent has two ports on which it acts synchronously. The action of a `SYNC` agent consists of an atomic sequence of two actions: a `take` on the input port, and a `put` on the output port. The composite action succeeds if and only if the two parts of the action succeeds, and the value *put* on the output port corresponds to the value *taken* on the input port. Note that the value of the datum that a sync puts on its output port is initially unknown. We use the symbol `?` for such unknown value, and use the operation `resolve` to instantiate the value at runtime.

```
fmod SYNC is
  inc AGENT{ASemiring} .
  inc PORT-INTERFACE .
  inc CHANNEL-INTERFACE .
  ...
  eq getOutput(Sync(p1, p2), Channel, id', a, M) = empty .

  eq getPostState(Sync(p1, p2), Channel, Sync(p1, p2), take, (id, k("data
      ") |-> d), M) = insert(k("data"), d, M) .

  eq getPostState(Sync(p1, p2), Channel, Sync(p1, p2), take, inputs, M) =
       notAllowed(take) [owise] .

  ceq getPostState(Sync(p1, p2), Channel, Sync(p1, p2), put(d), outputs,
      M) = insert(k("data"), nodata, M)
      if M =/= notAllowed(take) .

  eq getPostState(Sync(p1, p2), Channel, id, a, outputs, M) = M [owise] .

  eq computeActions(Sync(p1, p2) , Channel, M ) = (((Sync(p1, p2) , (take
       ; p1)), (Sync(p1, p2) , (put(?) ;  p2))), 1) .

  eq internalUpdate(Sync(p1, p2), Channel, M) = M .

  ceq resolve(put(?), Sync(p1, p2), Channel, M) = put(d)
      if d := M[k("data")] .
endfm
```

A `FIFO` agent has two ports on which it acts in sequence. A `FIFO` agent has two actions, a `take` action that stores the data from its input port to a memory, or a `put`

action that outputs the data on the output port. The `take` action succeeds only if the current memory cell is empty, and the `put` action succeeds only if the current memory cell is full. As a result, the `FIFO` agent alternates between taking a value from its input port, and putting that value to its output port.

```
fmod FIFO is
  inc AGENT{ASemiring} .
  inc PORT-INTERFACE .
  inc CHANNEL-INTERFACE .
  ...
  eq getOutput(id, Channel, id', a, M) = empty .

  eq getPostState(Fifo(p1, p2), Channel, Fifo(p1, p2), take, (id, k("data
      ") |-> d), M) = insert(k("data"), d, insert(k("state"), nd(1), M))
      .
  eq getPostState(Fifo(p1, p2), Channel, Fifo(p1, p2), put(d), outputs, M
      ) = insert(k("data"), nodata, insert(k("state"), nd(0), M)) .
  eq getPostState(Fifo(p1, p2), Channel, id, a, outputs, M) = M [owise] .
  ceq computeActions(Fifo(p1, p2), Channel, M) = (Fifo(p1, p2) , (take ;
      p1), 1) if M[k("state")] == nd(0) .
  ceq computeActions(Fifo(p, p'), Channel, M) = (Fifo(p, p'), (put(M[k("
      data")]); p'), 1) if M[k("state")] == nd(1) .

  eq internalUpdate(Fifo(p1, p2), Channel, M) = M .
endfm
```

A `MERGER` is a ternary connector, that acts, for each of its port input, as a synchronous channel with its output port. Moreover, the `MERGER` relates its two input ports with a relation of exclusion, i.e., the two input ports cannot fire at the same time. A `MERGER` with the list of ports `P(1), P(2), P(3)` has two actions: a forwarding action from port `P(1)` to port `P(3)`, or a forwarding action from port `P(2)` to port `P(3)`. The two actions are exclusive, as they cannot occur at the same time. Moreover, the merger always enables both actions, which raises some non-determinism at the system level (i.e., to chose which of the two actions is selected). Similarly to the `SYNC` channel, a `MERGER` agent instantiates the value for the `put` action at runtime, once the result of the `take` action is known. We use the `?` symbol to denote a symbolic value.

```
fmod MERGER is
  inc AGENT{ASemiring} .
  inc PORT-INTERFACE .
  inc CHANNEL-INTERFACE .
```

```
  eq getOutput(Merger(p1, p2, p3), Channel, id', a, M) = empty .

  ceq getPostState(Merger(p1, p2, p3), Channel, Merger(p1, p2, p3), take,
       (id, k("data") |-> d), M) =
    insert(k("data"), d, M) if M[k("data")] == nodata .
  eq getPostState(Merger(p1, p2, p3), Channel, Merger(p1, p2, p3), take,
      inputs, M) = notAllowed(take) [owise] .
  ceq getPostState(Merger(p1, p2, p3), Channel, Merger(p1, p2, p3), put(d
      ), outputs, M) =
    insert(k("data"), nodata, M) if M =/= notAllowed(take) .
  eq getPostState(Merger(p1, p2, p3), Channel, id, a, outputs, M) = M [
      owise] .

  eq computeActions(Merger(p1, p2, p3) , Channel, M ) =
    (((Merger(p1, p2, p3), (take; p2)), (Merger(p1, p2, p3), (put(?);  p3
        ))), 1) + (((Merger(p1, p2, p3), (take; p1)), (Merger(p1, p2, p3)
        , (put(?);  p3))), 1) .

  eq internalUpdate(Merger(p1, p2, p3), Channel, M) = M .

 ceq resolve(put(?), Merger(p1, p2, p3), Channel, M) = put(d)
   if d := M[k("data")] .
endfm
```

A `PROD` and `CONS` agent respectively implement a producer and consumer. Both agents have a single port, on which they always perform, respectively, a `put` and a `take` action. The `PROD` agent puts natural numbers as values on its port, and increments the value if the `put` action succeeds. We use such canonical sequences of increasing natural numbers to verify some firing properties of a Reo circuit. When a `put` action succeeds for a `PROD` agent, its state is updated to contain the message that has been sent in the `k("sent")` field. Similarly, when a `take` action succeeds for a `CONS` agent, its state is updated to contain the message that has been received in the `k("recv")` field.

```
fmod PROD is
  inc AGENT{ASemiring} .
  inc PROD-INTERFACE .
  inc PORT-INTERFACE .

  ceq getPostState(id , Producer , id', put(d) , actionoutput, M) =
      insert(k("data"), nd(s(i)), insert(k("sent"), d, M)) if nd(i) := M[
      k("data")] .
  eq getPostState(id , Producer , id', a , actionoutput, M) = M [owise] .
```

```
  eq getOutput(id , Producer , id', a, M) = empty .

  eq computeActions(Prod(id) , Producer, M ) =
    (Prod(id), (put(M[k("data")]); id), 1) .

  eq internalUpdate(id, Producer, M) = insert(k("sent"), nodata, M) .
endfm

fmod CONS is
  inc AGENT{ASemiring} .
  inc CONS-INTERFACE .
  inc PORT-INTERFACE .

  eq getPostState(Cons(id) , Consumer , Cons(id), take, (id, k("data")
      |-> d), M) = insert(k("recv"), d, M) .
  eq getPostState(id , Consumer , id', a , actionoutput, M) = M [owise] .

  eq getOutput(id , Consumer , id', a, M) = empty .

  eq computeActions(Cons(id) , Consumer, M ) =
    (Cons(id), (take ; id), 1) .

  eq internalUpdate(id, Consumer, M) = M .

endfm
```

5.2.2 Execution and analysis

We present in `SCENARIO` two system terms for which we run some analysis. For the first scenario, the `PROD` and `CONS` are communicating through a sync channel. Therefore, the only possible composite action is a *clique* in which the `PROD` agents puts a value on port `P(1.0)`, that is forwarded to port `P(2.0)` by the `SYNC` agent, and then consumed by the `CONS` agent. The second scenario is similar, and models the `PROD` and `CONS` agents communicating through a fifo channel.

Note that the `SCENARIO` module instantiates the `linearization` operation as follows: a `take` action on a port `P(i)` happens after a `put` action a port `P(j)` if $i \geq j$; and a `take` action on a port `P(i)` happens after a `take` action on a port `P(j)` if $i > j$.

Listing 5.5: Scenarios for Producer/Consumer protocols.

```
mod SCENARIO is
  inc PORT .
  inc PROD . inc CONS .
```

```
  inc SYNC . inc FIFO . inc MERGER .
  inc RUN . inc CONVERSION .

  op init : Nat -> Global .

  eq init(1) = [
    [P(1.0) : Port | k("data")|-> nodata,
        k("sync") |-> bd(false) ; false ; null]
    [P(2.0) : Port | k("data")|-> nodata,
        k("sync") |-> bd(false) ; false ; null]
    [Prod(P(1.0)) : Producer |
        k("data") |-> nd(1) ; false ; null]
    [Cons(P(2.0)) : Consumer |
        k("data") |-> nodata ; false ; null]
    [Sync(P(1.0), P(2.0)) : Channel |
        k("data") |-> nodata ; false ; null]] .

  eq init(2) = [
    [P(1.0) : Port | k("data")|-> nodata,
        k("sync") |-> bd(false) ; false ; null]
    [P(2.0) : Port | k("data")|-> nodata,
        k("sync") |-> bd(false) ; false ; null]
    [Prod(P(1.0)) : Producer |
        k("data") |-> nd(1) ; false ; null]
    [Cons(P(2.0)) : Consumer |
        k("recv") |-> nodata ; false ; null]
    [Fifo(P(1.0), P(2.0)) : Channel | k("state") |-> nd(0),
        k("data") |-> nodata ; false ; null]] .

 ceq linearization(aSet) = a linearization(aSet')
   if a , aSet' := aSet .
 ceq (id, (take ; P(i))) (id', (put(d) ; P(j)))  aSeq'  =
   (id', (put(d); P(j))) (id, (take; P(i))) aSeq' if i >= j .
 ceq (id1, (take ; P(i))) (id2, (take ; P(j)))  aSeq' =
   (id2, (take ; P(j))) (id1, (take ; P(i))) aSeq' if i > j .
endm
```

We run the two following queries on the two initial terms of `SCENARIO`. The first query returns the first solution for which the same datum has passed from the producer to the consumer. The solution shows the synchronicity of the `SYNC` channel.

The second query returns the first solution for which the data received by the consumer has an offset of 1 with the data sent by the producer. The solution shows the asynchronicity of the `FIFO` channel.

```
search [1] init(1) =>* [sys::Sys
```

```
 [Prod(P(1.0)) : Producer | M1::MapKD,
     k("sent") |-> nd(i::Nat); false; null]
 [Cons(P(2.0)) : Consumer | M2::MapKD,
     k("recv") |-> nd(j::Nat); false; null]
 ] such that j::Nat == i::Nat .

Solution 1 (state 8) states: 9  rewrites: 1472 in 3ms cpu (0ms real)
    (443774 rewrites/second)
...
i::Nat --> 1
j::Nat --> 1
========================================
search [1] init(2) =>* [sys::Sys
[Prod(P(1.0)) : Producer | M1::MapKD,
    k("sent") |-> nd(i::Nat); false; null]
[Cons(P(2.0)) : Consumer | M2::MapKD,
    k("recv") |-> nd(j::Nat); false; null]
] such that j::Nat =/= i::Nat .

Solution 1 (state 24) states: 25  rewrites: 6390 in 0ms cpu (2ms real) (~
     rewrites/second)
...
i::Nat --> 2
j::Nat --> 1
```

5.3 Valve-controller

5.3.1 N-reservoir problem

Consider n reservoirs, filled with water. Each reservoir connects at the top to a valve that, if switched on, inputs some water. Each reservoir has a hole at the bottom from where water goes out. Each reservoir has a sensor measuring its water level. The measures are accessible by a digital controller. A valve is placed at the top of the n reservoirs, and fills, continuously, one of the reservoir at a time. The valve moves from one reservoir to another after reception of a switch command from the controller.

The problem is to design a controller such that none of the reservoirs is observed with its water level outside of given bounds, noted l_1^k for the lower and l_2^k for the higher bounds, for the k-th reservoir. We first introduce some intuitive physical explanations about the dynamics of the reservoirs, then specify the digital controller that interfaces the physical components, and finally analyse one instance of the composite cyber-physical system.

Physical part In the n-reservoirs problem, the physical part is described by the collection of n reservoirs, and a valve. The water level of a reservoir follows a continuous evolution: at each time $t \in \mathbb{R}_+$, there exists a value $x(t)$ that corresponds to the water level of the reservoir at t.

Along this part, we use $t \in \mathbb{R}_+$ for continuous time, $n \in \mathbb{N}$ for the number of reservoirs, and $k \in \{1, ..., n\}$ to denote the k-th reservoir.

Reservoir The reservoir of radius r captures, as a component, the evolution of the water level over time. The height of the water at time t in the k-th reservoir, written $x_k(t)$, follows the law $x_k(t) = x_k^{t_0} + (\int_{t_0}^{t} i_k(u) - o_k(u)\, du)/(\pi r^2)$, where $x_k^{t_0} \in \mathbb{R}$ is the initial level of water at $t_0 \in \mathbb{R}_+$, $i_k(t)$ is the rate of incoming water and $o_k(t)$ is the rate of outgoing water for $t \geq t_0$. Note that we use the relation $V = \pi r^2 L$ where V is the volume of a cylindrical reservoir of radius r and height L. The functions x_k, o_k and i_k are functions from $\mathbb{R}_+$ to $\mathbb{R}_+$.

The definition of $x_k(t)$ involves elements internal to the physical description of the reservoir (e.g. $o_k(t)$, the rate of water going out), and external elements (e.g. $i_k(t)$, the rate of water coming in). In particular, the function $i_k(t)$ depends on the valve component's specification.

The component for the reservoir of radius r is the pair $Res(r, i) = (E, L)$ where

$$E = \{read(l), inFlow(k) \mid l, k \in [0, 20]\}$$

and $L \subseteq TES(E)$ with $\sigma \in L$ implies there exists a function $f : \mathbb{R}_+ \to \mathbb{R}_+$ with

- $f(0) = 15$ and $f' = -outFlow$, where $outFlow$ is a constant that models how fast water goes out of the reservoir;
- $\sigma(t_i) = \{read(l)\}$ implies $l = \max(f(t_i), 0)$;
- $\sigma(t_i) = \{inFlow(k)\}$ implies $f(t) = f(t_i) + (t - t_i) \times (k - outFlow)/(r^2\pi)$ for all $t \in]t_i, t_{i+1}]$, where k is the rate of water going in the reservoir in the interval $]t_i, t_{i+1}]$.

Valve The valve fixes the rate at which the water flows into the reservoir, and moves its arm from one reservoir to another. The state of the valve is described by a value $s \in \{0, ..., n\}$ where $s = i$ encodes the fact that the valve is placed above reservoir i only. Under an input signal *switch(k)*, the valve changes state from $s = i$ to $s = k$.

The component for the valve is the pair $Valve(n) = (E(n), L)$ where

$$E(n) = \{switch(k), outFlow(k, j) \mid k \in \{1, ..., n\}, j \in [0, 20]\}$$

and $L \subseteq TES(E(n))$ with $\sigma \in L$ such that every switch event is simultaneous with 0 outFlow in the reservoir that are not recipient of the valve, i.e., for all t, if $switch(k) \in \sigma(t)$ then $outFlow(i, 0) \in \sigma(t)$ for $i \in \{1, ..., n\} \setminus \{k\}$ and $outFlow(i, j) \in \sigma(t)$ with $j \neq 0$.

Continuous formulation of the problem The problem can be formulated in the continuous setting as finding a sequence of states for the valve (or timed input signals $\delta(t)$, equivalently) such that

$$\forall k \in \{1, .., n\}.\ l_1^k \leq x_k(t) \leq l_2^k$$

where l_1^k and l_2^k are respectively lower and upper bounds for the water level x_k of reservoir k.

The function δ is not built as a physical component, but results from the measures and the actions of a digital component on the physical system. We show in the next subsection an instance of a controller, and later provide an instance of the physical components (reservoir and valve) in order to analyse the resulting interaction with a controller.

Cyber part The cyber component consists of a controller, reading from the reservoirs' sensors, and acting on the valve. In sequence, this subsection details the measurement device that interface the reservoir and the controller, the action that the controller performs on the valve, and the controller's state transition system.

The problem of decision inherent in the n-reservoirs system is dependent on the water level measured by the controller. If the agent can read precisely the level of water in each of the reservoir, at a high frequency, the reactivity of the system will be higher than the case where the agent measures the reservoir's state at low frequency, with less accuracy. From a formal point of view, increasing the sensor precision also increases the possible states in which the system can be observed, and therefore the complexity of the representation. The trade off is to find a reading frequency that discriminates the minimal amount of states, while ensuring the agent to be reactive to achieve its goal.

As a result of an action, the dynamic of the physical system may change, and lead to different sequences of measures from the controller.

Controller The controller has two types of events: *read* and *switch*. The *read(i,l)* event is parametrized by the reservoir i that the controller reads, and displays the value l that is read at the sensor. The *switch(i)* event synchronizes with the same event of the valve and takes as parameter the value of the reservoir to which the valve should switch.

A controller is a component defined as the pair $C(T) = (E, L(T))$ with

$$E = \{read(i, l), switch(i) \mid i \in \{1, ..., n\}, l \in [0, 20]\}$$

and $L(T) \subseteq TES(E)$ is such that $\sigma \in L(T)$ implies there exists a function $f : \mathbb{R}_+ \to \{1, ..., n\} \to [0, 20]$ with

- observations occur at multiple of T, i.e, $\sigma(t) \neq \emptyset$ implies $t = 0 \mod T$;
- $read(i, l) \in \sigma(t)$ implies $f(t)(i) = l$

The controller may change state after performing some measurements or moving the valve, such as recording the sensor values or the reservoir to which the valve has switched. In the next subsection, we give an executable specification for the controller, the valve, and the reservoir. We analyse the strategy of the controller to keep the water level of the reservoirs within a margin.

5.3.2 Execution and analysis

We present the three main Maude modules, namely the `CONTROLLER`, the `VALVE`, and the `RESERVOIR`, to model and analyse properties of the N-reservoirs problem.

Controller The `CONTROLLER` agent has two main actions: `read` and `switch`. The `read` action returns the value of the water level in all reservoirs. The `CONTROLLER` agent stores the value in its state, and implements the function `lowestLevel` to return the reservoir with the lowest water level (ordered with the reservoir identifier, in case of two reservoirs having the smallest water level). The `switch` action takes a reservoir's identifier as argument and acts on the `VALVE` agent by switching the valve to that reservoir.

Listing 5.6: A module for the `CONTROLLER` agent.

```
fmod CONTROLLER is
    inc SET{Nat} .
    inc TRACE-INTERFACE .
    inc RESERVOIR-INTERFACE .
    inc CONTROLLER-INTERFACE .
    inc AGENT{ASemiring} .
    ...
    eq getPostState(id, Controller , id, read, output, M) =
        getSensorValues(getResources(id, read), output, M) .
    eq getPostState(id, Controller, id', switch(r), actionoutput, M) =
        insert(k("state"), d(r), M) .
    eq getPostState(id, Controller, id', a, actionoutput, M) = M [owise]
        .

    eq computeActions(id , Controller , M) =  null [owise] .

    *** Upper and lower bound for the reservoir level
    ceq computeActions(id, Controller, M) = (id, (switch(r)  ;
        getResources(id , switch(r)) ) , 1) if
         M =/= empty /\ r := lowestLevel(M) /\
         d(r) =/= M[k("state")] /\ fd(j) := M[k("lev", r)] /\
         j <= lowbound(r) .
    eq computeActions(id, Controller, M) = (id, (read ; getResources(id,
        read)),  1) [owise] .

    eq internalUpdate(id, Controller, M) = M .
endfm
```

Valve The `VALVE` agent has a `fill` action for each reservoir. The `fill` action takes a flow value as argument, and changes the inflow rate in the `RESERVOIR` agent. As a consequence, the state of the `RESERVOIR` will change its dynamic and the `read` action from the `CONTROLLER` on the `RESERVOIR` will get updates accordingly. The `fill` action of the `VALVE` is composite of two other actions that turn `off` the inflow of the other `RESERVOIR` agents. In Listing 5.7, we show one of such composite action that fills reservoir 1 and turns off the two other reservoirs.

Listing 5.7: A module for the `VALVE` agent.

```
fmod VALVE is
  inc AGENT{ASemiring} .
  inc VALVE-INTERFACE .
  inc RESERVOIR-INTERFACE .
  inc TIME-INTERFACE .
  inc CONTROLLER-INTERFACE .
```

```
  ...
  *** Passive agent:
  ceq computeActions(id, Valve, M ) = (((valve, (fill(k) ; res(1))), (
      valve, (off ; res(2))), (valve, (off ; res(3)))), 1) if nd(1) := M[
      k("state")] /\ fd(k) := M[k("inFlow")] .
  ...
  ceq getPostState(r, Valve, id, switch(res(i)), output, M) =  M' if M'
      :=  insert( k("state") , nd(i) , M) .
  eq getPostState(r, Valve, id, switch(r), output, M) = notAllowed(switch
      (r)) [owise] .
  eq getPostState(r, Valve, id, an, output, M) =  M [owise] .

  eq internalUpdate(id, Valve, M) = M .
endfm
```

Reservoir For simplicity, we set the radius of the `RESERVOIR` to be equal to $\sqrt{\pi}$, and the inflow and outflow corresponds to the change of level in the reservoir. The `RESERVOIR` agent has no actions, but reacts to the `fill` and `off` actions of the `VALVE` agent, and the `read` action of the `CONTROLLER` agent. The `read` action generates an output that contains the current level of the reservoir. The `fill` action changes the `vin` state variable to take the value given by the `VALVE` agent. The `off` action sets the `vin` state variable to 0.0. A `RESERVOIR` agent defines a function `f` that computes the water level given the current level, the inflow and outflow models by `vin` and `vout` respectively. At the end of every round, the `RESERVOIR` updates its state with the value computed by `f`.

Listing 5.8: A module for the `RESERVOIR` agent.

```
fmod RESERVOIR is
  inc AGENT{ASemiring} .
  inc RESERVOIR-INTERFACE .
  inc TIME-INTERFACE .
  inc CONTROLLER-INTERFACE .
  inc VALVE-INTERFACE .
  ...
  *** Passive agent:
  eq computeActions(id , Reservoir, M ) = null .
  *** Compute the level of the reservoir
  op f : Float Float Float -> Float .
  eq f(cl, vin, vout) = max(min(cl + (vin - vout) , 20.0), 0.0) .

  eq getOutput(id, Reservoir, id', read, M) = k("lev") |-> M[k("lev")] .
```

```
  eq getPostState(id, Reservoir, id', read, output, M) = M .
  ceq getPostState(id, Reservoir, id', end, output, M) = insert(k("lev"),
       fd(f(cl, vin, vout)), M)
      if  k("inFlow") |-> fd(vin) , k("outFlow") |-> fd(vout), k("lev")
          |-> fd(cl), M' := M .
  eq getPostState(res(j), Reservoir, id', fill(vin), output, M) = insert(
      k("inFlow"), fd(vin), M)  .
  eq getPostState(id, Reservoir, id', off, output, M) =  insert(k("inFlow
      "), fd(0.0), M)  .
  eq getPostState( r, Reservoir, id , an , output, M) = M [owise] .

  eq internalUpdate(id, Reservoir, M) = M .
endfm
```

Scenarios We present in Listing 5.9 a scenario for which a controller periodically reads the value of the water level in the reservoirs, and switches the valve accordingly. We set the maximum capacity for each reservoir to be 20.0, the outflow rate to be 2.0 and the inflow rate from the valve to be 5.0. Initially, the valve is above `res(2)`.

We also define the `linearization` operation to order actions as follows. The `read` action occurs first in the sequence, followed by the `switch` action, and the other actions occur freely. The value taken by `read` action therefore takes the value of the water level at the end of the previous round, i.e., after update of each `RESERVOIR` agent.

Listing 5.9: A module for the `SCENARIO` agent.

```
mod SCENARIO is
  inc RESERVOIR . inc VALVE .
  inc CONTROLLER . inc RUN .
  ...
  *** Resources for controller agent.
  eq getResources(id , read) = res(1), res(2), res(3) .
  eq getResources(id , switch(id')) = valve .
  eq getResources(id , off) = res(1), res(2), res(3) .

  eq lowbound(res(i)) = 15.0 .  eq upbound(res(i)) = 18.0 .

  eq init = [[res(1) : Reservoir | k("lev") |-> fd(20.0),  k("state") |->
       nd(0), k("inFlow") |-> fd(0.0) , k("outFlow") |-> fd(2.0) ; false
      ; null ]
   [res(2) : Reservoir | k("lev") |-> fd(20.0),  k("state") |-> nd(1), k(
       "inFlow") |-> fd(5.0) , k("outFlow") |-> fd(2.0) ; false ; null ]
   [res(3) : Reservoir | k("lev") |-> fd(20.0),  k("state") |-> nd(0), k(
       "inFlow") |-> fd(0.0) , k("outFlow") |-> fd(2.0) ; false ; null ]
```

```
   [valve : Valve | k("state") |-> nd(2), k("inFlow") |-> fd(5.0)  ;
       false ; null ]
   [controller : Controller | k("state") |-> d(nil) ; false ; null ]] .
  ...
  ceq linearization(aSet) = a linearization(aSet')
   if a , aSet' := aSet .
  eq a (id, (read ; res )) = (id, (read ; res)) a  .
  ceq a (id, (switch(id'); res)) = (id, (switch(id'); res)) a
   if (id', (an ; res' )) := a  /\ an =/= read .
endm
```

Search queries We run two queries on the scenario of Listing 5.9. The first query searches for a state for which all the controller has read a value for the water level which is below 8.0. We can see that one of such solution exists as the result of the query.

The second query searches for a state for which the controller read one of the reservoir's water level to be 0.0. As shown in the output, a solution exists, and `res(2)` may reach a water level of 0.0.

Listing 5.10: Two search queries for some safety properties.

```
search [1] in SCENARIO : init =>* [sys:Sys
[controller : Controller | M::MapKD, k("lev",res(1)) |-> fd(j::Float), k(
    "lev",res(2)) |-> fd(i::Float), k("lev",res(3)) |-> fd(k::Float) ;
    false ; null]
] such that (j::Float < 8.0 and k::Float < 8.0) and i::Float < 8.0 = true
     .

Solution 1 (state 180)
states: 181  rewrites: 27201 in 13ms cpu (12ms real) (2041963 rewrites/
    second)
...
j::Float --> 3.0 i::Float --> 7.0 k::Float --> 7.0


search [1] init =>*
[ sys:Sys
    [controller : Controller |  k("lev", res(1)) |-> fd(j::Float), k("lev
        ", res(2)) |-> fd(i::Float), k("lev", res(3)) |-> fd(k::Float), M
        ::MapKD ; false ; null]
]  such that i::Float == 0.0 or j::Float == 0.0 or k::Float == 0.0  .

Solution 1 (state 96)
```

```
states: 97  rewrites: 14927 in 6ms cpu (6ms real) (2240618 rewrites/
    second)
...
j::Float --> 1.4e+1 i::Float --> 0.0 k::Float --> 9.0
```

However, after changing the rate of the valve from 5.0 to 7.0, the same queries as in Listing 5.10 return no solutions. As a consequence, the strategy implemented in the `CONTROLLER` agent therefore successfully keeps the water level above 8.0 for all reservoirs, and prevent any reservoir to reach 0.0.

5.4 Robot-Battery-Field system

We propose to study three properties:

1. Safety property: In the first scenario, we model two `TROLL` agents, moving on a shared `FIELD`, with private `BATTERY` agents; we study the cases for which the two robots can exchange their position without running out of energy.

2. Liveness property: In the second scenario, the field is equipped with a station, where the `TROLL` agent can recharge its battery. We want to prevent the agent from running out of energy while oscillating between the two locations, i.e., if the station can always supply energy, we want a sequence of actions such that the agent never runs out of energy.

3. Self-sorting property: In the third scenario, we place three `TROLL` agents on a grid, each with a unique natural number identifier. We study some self-sorting property of the system by global coordination (e.g., use of an external protocol) or local strategies (e.g., ranking of each agent's actions).

We consider the battery, robot, and grid components introduced in Section 1.3 and formalized in Section 2.1.6 and Section 2.2.3 as the expression

$$Sys(n, T_1, ..., T_n) = \bowtie_{i\in\{1,...,n\}} (R(i, T_i) \times_{\Sigma_{R_i B_i}} B_i) \times_{\Sigma_{RF}} G_\mu(\{1, ..., n\}, n, 2)$$

made of n robots $R(i, T_i)$, each interacting with a private battery B_i under the interaction signatures $\Sigma_{R_i B_i}$, and in product with a grid G under the interaction signature Σ_{RG}. We use $\bowtie$ for the product with the free interaction signature (i.e., every pair of TESs is composable), and the notation $\bowtie_{i\in\{1,...,n\}} \{C_i\}$ for $C_1 \bowtie ... \bowtie C_n$ as $\bowtie$ is commutative and associative.

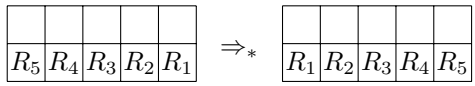


Figure 5.1: Initial state of the unsorted robots (left), and final state of the sorted robots (right).

	R_1	R_2	R_3	R_4	R_5
t_1	N	-	-	-	-
t_2	W	-	-	-	-
t_3	$(3;1)$	-	-	-	-
...	...	...	...	...	...

	R_1	R_2	R_3	R_4	R_5
t_1	N	-	-	-	-
t_2	W	E	N	-	-
t_3	S	$(4;0)$	E	-	-
...	...	...	...	...	...

	R_1	R_2	R_3	R_4	R_5
t_1	N	-	-	-	N
t_2	W	E	-	W	E
t_3	S	-	-	-	S
...	...	...	...	...	...

Table 5.1: Each table displays the three first observables at times t_1, t_2, and t_3 for three TESs in the behavior of the product of components R_1, R_2, R_3, R_4, and R_5 on the grid of Figure 5.1. We omit the subscript and use the column to identify the events. The symbol - represents the absence of observation in the TES.

We fix $n = 5$ and the same period T for each robots. We write E for the set of events of the composite system $Sys(2, T)$. We also reuse the grid component introduced in Section 2.2.3 where μ is the initial position of the robots on the grid, and the parameters n and 2 a refers to the x and y length of the grid. For simplicity, use R_i to denote the composite component $(R(i, T) \times_{\Sigma_{R_i B_i}} B_i)$ with fixed period T.

Self-sorting robots Figure 5.1 shows five robot instances, each of which has a unique and distinct natural number assigned, positioned at an initial location on a grid. The goal of the robots in this example is to move around on the grid such that they end up in a final state where they line-up in the sorted order according to their assigned numbers.

Three first observations for three behaviors are displayed in Table 5.1. Each behavior exposes different degrees of concurrency, where in the left behavior, only robot R_1 moves, while in the middle behavior, robots R_1 and R_2 swap their positions, and in the right behavior both R_1 and R_4 swap their positions with R_2 and R_5, respectively.

We consider the following property: *eventually, the position of each robot R_i is* $(i, 0)_{R_i}$, i.e., every robot successfully reaches its place. This property is a trace property, which we call P_{sorted} and consists of every behavior $\sigma \in TES(E)$ such that there

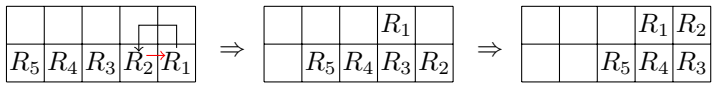


Figure 5.2: Initial state of the unsorted robot (left) leading to a possible deadlock (right) if each robot follows its strategy.

exists an $n \in \mathbb{N}$ with $\sigma(n) = (O_n, t_n)$ and $(i, 0)_{R_i} \in O_n$ for all robots R_i. As shown in Table 5.1, the set of behaviors for the product of robots is large, and the property P_{sorted} does not (necessarily) hold *a priori*: there exists a composite behavior τ for the component $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 \bowtie F(\{1, 2, 3, 4, 5\})$ such that $\tau \notin P_{sorted}$.

Robots may beforehand decide on some strategies to swap and move on the grid such that their composition satisfies the property P_{sorted}. For instance, consider the following strategy for each robot R_n:

- *swapping*: if the last read (x, y) of its location is such that $x < n$, then move North, then West, then South.
- *pursuing*: otherwise, move East.

Remember that the grid prevents two robots from moving to the same cell, which is therefore removed from the observable behavior. We emphasize that some sequences of moves for each robot may deadlock, and therefore are not part of the component behavior of the system of robots, but may occur operationally by taking a composable action step by step (see Section 4.1.2). Consider Figure 5.2, for which each robot follows its internal strategy. Because of non-determinism introduced by the timing of each observations, one may consider the following sequence of observations: first, R_1 move North, then West; in the meantime, R_2 moves West, followed by R_3, R_4, and R_5. By a similar sequence of moves, the set of robots ends in the configuration on the right of Figure 5.2. In this position and for each robot, the next move dictated by its internal strategy is disallowed, which corresponds to a *deadlock*. While behaviors do not contain finite sequences of observations, which makes the scenario of Figure 5.2 not expressible as a TES, such scenario may occur in practice. We give in next Section some analysis to prevent such behavior to happen.

Alternatively, the collection of robots may be coordinated by an external protocol that guides their moves. Besides considering the robot and the grid components, we add a third kind of component that acts as a coordinator. In other words, we make the protocol used by robots to interact explicit and external to them and the grid; i.e., we assume exogenous coordination. Exogenous coordination allows robots

to decide a priori on some strategies to swap and move on the grid, in which case their external coordinator component merely unconditionally facilitates their interactions. Alternatively, the external coordinator component may implement a protocol that guides the moves of a set of clueless robots into their destined final locations. The most intuitive of such coordinator is the property itself as a component. Indeed, let $C_{sorted} = (E, L)$ be such that $E = \bigcup_{i \in I} E_{R_i}$ with $I = \{1, 2, 3, 4, 5\}$ and $L = P_{sorted}$. Then, and as shown in [57], the coordinated component $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 \bowtie G(\{1, 2, 3, 4, 5\}, 5, 2) \bowtie C_{sorted}$ trivially satisfies the property P_{sorted}. While easily specified, such coordination component is non-deterministic and not easily implementable. We provide an example of a deterministic coordinators.

As discussed, we want to implement the property P_{sorted} as a collection of small coordinators that swap the position of unsorted robots. Intuitively, this protocol mimics the behavior of bubble sort, but for physical devices. Given two robot identifiers R_1 and R_2, we introduce the swap component $S(R_1, R_2)$ that coordinates the two robots R_1 and R_2 to swap their positions. Its interface $E_S(R_1, R_2)$ contains the following events:

- start(S(R_1,R_2)) and end(S(R_1, R_2)) that respectively notify the beginning and the end of an interaction with R_1 and R_2. Those events are observed when the swap protocol is starting or ending an interaction with either R_1 or with R_2.
- $(x, y)_{R_1}$ and $(x, y)_{R_2}$ that occur when the protocol reads, respectively, the position of robot R_1 and robot R_2,
- d_{R_1} and d_{R_2} for all $d \in \{N, W, E, S\}$ that occur when the robots R_1 and R_2 move;
- locked(S(R_1,R_2)) and unlocked(S(R_1,R_2)) that occur, respectively, when another protocol begin and end an interaction with either R_1 and R_2.

The behavior of a swapping protocol $S(R_1, R_2)$ is such that, it starts its protocol sequence by an observable start(S(R_1, R_2)), then it moves R_1 North, then R_2 East, then R_1 West and South. The protocol starts the sequence only if it reads a position for R_1 and R_2 such that R_1 is on the cell next to R_2 on the x-axis. Once the sequence of moves is complete, the protocol outputs the observable end(S(R_1, R_2)). If the protocol is not swapping two robots, or is not locked, then robots can freely read their positions.

Swapping protocols interact with each others by locking other protocols that share the same robot identifiers. Therefore, if S(R_1,R_2) starts its protocol sequence, then

S(R_2, R_i) synchronizes with a locked event locked(S(R_2,R_i)), for $2 < i$. Then, R_2 cannot swap with other robots unless S(R_1,R_2) completes its sequence, in which case end(S(R_1, R_2)) synchronizes with unlocked(S(R_2,R_i)) for $2 < i$. We extend the underlying composability relation κ on observations such that, for $i < j$, simultaneous observations (O_1, t) and (O_2, t) are composable, i.e., $((O_1, t), (O_2, t)) \in \kappa$, if:

$$\begin{aligned} \text{start}(\text{S}(R_i,R_j)) \in O_1 \implies & \exists k.k < i.\text{locked}(\text{S}(R_k, R_i)) \in O_2 \vee \\ & \exists k.j < k.\text{locked}(\text{S}(R_j, R_k)) \in O_2 \end{aligned}$$

and

$$\begin{aligned} \text{end}(\text{S}(R_i,R_j)) \in O_1 \implies & \exists k < i.\text{unlocked}(\text{S}(R_k, R_i)) \in O_2 \vee \\ & \exists j < k.\text{unlocked}(\text{S}(R_j, R_k)) \in O_2 \end{aligned}$$

For each pair of robots R_i, R_j such that $i < j$, we introduce a swapping protocol S(R_i, R_j). As a result, the coordinated system is given by the following composition:

$$R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 \bowtie G(\{1,2,3,4,5\},5,2) \bowtie_{i<j} S(R_i, R_j)$$

Note that the definition of $\bowtie$ imposes that, if one protocol starts its sequence, then all protocols that share some robot identifiers synchronize with a lock event. Similar behavior occurs at the end of the sequence.

5.4.1 Execution and analysis

Main agents To illustrate the use of our framework to simulate and verify cyber-physical systems, we present an agent specification for three components: a `FIELD`, a `TROLL`, and a `BATTERY`. A `FIELD` component interacts with the `TROLL` component by reacting to its move action, and its sensor reading. As shown in Listing 5.11 the `FIELD` agent has no actions, but reacts to the move action of the `TROLL` agent by updating its state and changing the agent's location. Currently, the update is discrete, but more sophisticated updates can be defined (e.g., changing the mode of a function recording the trajectory of the `TROLL` agent). In the case where the state of the `FIELD` agent forbids the `TROLL` agent's move, the `FIELD` agent enters in a disallowed state marked as `notAllowed(an)`, with `an` as the action name. The `FIELD` responds to the read sensor action by returning the current location of the `TROLL` agent as an output.

Listing 5.11: Extract from the `FIELD` Maude module.

```
fmod FIELD is
  inc TROLL-INTERFACE .
  inc FIELD-INTERFACE .
  inc PROTOCOL-INTERFACE .
  inc AGENT{ASemiring} .
  ...
  *** Passive agent:
  eq computeActions(id , Field, M) = null .

  eq internalUpdate(id , Field, M) = M .

  ceq getPostState(r, Field, id, a, mtOutput, M) = M'
      if isMove?(a) /\
         k(loc) |-> d(id) , M1' := M /\
         loc' := next(loc, a) /\
         loc' =/= loc /\
         M[k(loc')] == undefined /\
         M' := k(loc') |-> d(id), M1' .

  ceq getPostState(r, Field, id, a, mtOutput, M) = notAllowed(a)
      if isMove?(a) /\ k(loc) |-> d(id) , M1' := M /\
         loc' := next( loc , a ) /\ ((loc' =/= loc and  M[k(loc')] =/=
             undefined) or loc' == loc) .

  ceq getOutput(r, Field, id, readSensors(position sn), M)
                =  ( k("pos") |-> loc , M'  )
      if  k(loc) |-> d(id) , M1' := M /\
          M' := ( k("obstacles") |-> obstacle( 1, id, loc, M))   .
endfm
```

A `TROLL` agent reacts to no other agent actions, and therefore does not include any agent interface. A `TROLL` agent uses the function `getSoftActions` to return a ranked set of actions given its state, and implements the `computeActions` operation by returning the ranked set of actions. The operation `getSoftAction` implements a strategy for the agent which, for instance, ranks higher the move action that moves the robot closer to its target location. The expression may contain more than one action, with different weights. The weights of the action may depend on the internal goal that the agent set to itself, as for instance reaching a location on the field. The `TROLL` agent specifies how it reacts to, e.g., the sensor value input from the field, by updating the corresponding key in its state with `getSensorValues`.

Listing 5.12: Extract from the `TROLL` Maude module.

```
fmod TROLL is
```

```
  inc AGENT{ASemiring} .
  inc LOCATION .
  inc TROLL-INTERFACE .

  eq computeActions(id , Troll , M ) = getSoftActions(id, M ,
      trollActions(id, M)) .

  ceq internalUpdate(id, Troll, M) = insert(k("read"), nd(1), M) if M[k("
      read")] == nd(0) .
  ceq internalUpdate(id, Troll, M) = insert(k("read"), nd(0), M) if M[k("
      read")] == nd(1) .

  ceq getPostState(id, Troll, id, readSensors(sn), sensorvalues, M) = M'
      if M' := getSensorValues(getResources(id, readSensors(sn)) ,
          sensorvalues), k("goal") |-> M[k("goal")], k("read") |-> nd(1)
           .
endfm
```

A `BATTERY` agent does not act on any other agent, as the `FIELD`, but reacts to the `TROLL` agent actions. Each `move` action triggers in the `BATTERY` agent a change of state that decreases its energy level. As well, each `charge` action changes the `BATTERY` agent state to increase its energy level. Similarly to the field, in the case where the state of the battery agent has 0 energy, the battery enters a disallowed state marked as `notAllowed(an)`, with `an` as the action name. A sensor reading by the `TROLL` agent triggers an output from the `BATTERY` agent with the current energy level.

Listing 5.13: Extract from the `BATTERY` Maude module.

```
fmod BATTERY is
 inc AGENT{ASemiring} .
 inc BATTERY-INTERFACE .
 inc TROLL-INTERFACE .


 *** Passive agent:
 eq computeActions(id, Battery, M ) = null .
 eq internalUpdate(id, Battery, M ) = M .


 ceq getOutput(r, Battery, id, readSensors(energy sn), M)
           =  k("bat") |-> M[k("bat")]
     if r := getBattery(id) .

 *** Next state.
 ceq getPostState(r, Battery, id, an, mtOutput, M) =  M'
```

```
    if isMove?(an) /\
       k("bat") |-> nd(s i) , M1' := M /\
       M' := insert( k("bat") , nd(i) , M) .

 ceq getPostState(r, Battery, id, charge(j), mtOutput, M)  = M1
    if nd(i) := M[k("bat")] /\
        i  < capacity  /\
        M1 := insert( k("bat") , nd(min ( i + j, capacity)) , M ) .

 ceq getPostState(r, Battery, id, an, mtOutput, M) = notAllowed(an)
     if isMove?(an) /\ M[k("bat")] == nd(0) .

endfm
```

A PROTOCOL agent `swap(id1,id2)` acts on the TROLL agents `id1` and `id2`, and is used as a resource by the two TROLL agent move actions. A PROTOCOL internally has a finite state machine `T(id):Fsa` that accepts or rejects a sequence of actions. Each `move` action of a TROLL is accepted only if there is a transition in the PROTOCOL agent state transition system. A PROTOCOL agent `swap(id1, id2)` always tries to swap agents with ids `id1` and `id2`. Thus, if `id2` is on the direct East position of `id1` on the field, then action `start` succeeds, and the protocol enters in the sequence `move(N)` for `id2`, `move(W)` for `id2`, `move(E)` for `id1`, and then `move(S)` for `id2`. Eventually the sequence ends with `finish` action. The PROTOCOL agent may also have some transitions labeled with a set of actions, one for each of the agent `id1` and `id2`. In which case, the transition succeeds if the clique contains, for each agent involved in the protocol, an action that is composable with the action labeling the protocol transition. We use the `end` action to mark the end of the sequence of actions forming a clique. The PROTOCOL may reject such `end` action if the clique does not cover the set of actions labeling the transition, which therefore discard the set of actions as not composable.

Listing 5.14: Extract from the SWAP protocol Maude module.

```
fmod SWAP is
    inc AGENT{ASemiring} .
    inc TROLL-INTERFACE .
    inc PROCESS-INTERFACE .
    inc FIELD-INTERFACE .
    inc PROTOCOL-INTERFACE .

    op T : Identifier -> Fsa .
    *** Update of state from external move or its own swapping actions
    ceq getPostState(id, Protocol, id', move(d), sysState, M ) = M'
        if  {q(i)} := getState(M) /\
```

```
            M' :=  insert( k("recv") , recv(union(getLabel(M), {l(id',
                move(d))})) , M) .

    *** Ending transition correctly
    ceq getPostState(id , Protocol , id, end , sysState, M) = M'
        if  state := getState(M) /\
            label := getLabel(M) /\
            tr := getTransitions(T(id)) /\
            (state, label, state'), tr' := tr /\
            M' :=  insert( k("recv") , recv({}), insert( k("state") , ds(
                state') , M)) .

    *** Not allowed states
    eq getPostState(id, Protocol, id', end, sysState, M ) = notAllowed(
        end) [owise] .
    eq getPostState(id, Protocol, id', a, sysState, M) = M [owise] .

    eq getOutput(id, Protocol, id', a, M) = empty .
    ceq computeActions(swap(id, id') , Protocol, M ) = ((swap(id, id') ,
        ( start ; getResources(swap(id, id'), start))), 5)
        if {q(0)} := getState(M) .
    eq computeActions(swap(id, id'), Protocol, M) = null [owise] .
    eq internalUpdate(swap(id, id'), Protocol, M) = M .

endfm
```

Composability relation The `TROLL`, `FIELD`, and `BATTERY` modules specify the state space and transition functions for, respectively, a `TROLL`, `FIELD`, and `BATTERY` agent. A system consisting of a set of instances of such agents would need a composability relation to relate actions from each agent.

More precisely, we give some possible *cliques* of a system consisting of two `TROLL` agents with identifiers `id(0), id(1):TROLL`, one `field:FIELD` agent, and two `BATTERY` agents `bat(0), bat(1):BATTERY`.

The actions of agent `id(0)` compose with outputs of its corresponding battery `bat(0)` and of the shared `field` agent.

For instance, a move action of the `id(0)` agent is of the form `(id(0), (move(d), {bat(0), field}))`, where `d` is a direction for the move, and composes with outputs of the battery and field, both notifying that the move is possible.

Alternatively, a read action of the `id(0)` agent is of the form `(id(0), (read, {bat(0), field}))` and composes with outputs of the battery and field, each giving the battery level and the location of agent `id(0)`.

Safety property: not running of energy We consider a system containing two `TROLL` agents, with identifiers `id(0)` and `id(1)`, paired with two `BATTERY` agents with identifier `bat(0)` and `bat(1)`, and sharing the same `FIELD` resource. The goal for each agent is to reach the initial location of the other agent. If both agents follow the shortest path to their goal location, there is an instant for which the two agents need to swap their positions. The crossing can lead to a livelock, where agents move symmetrically until the energy of the batteries runs out.

The initial system term, without the protocol, is given by:

```
eq init = [[id(0): Troll | k("goal") |-> (5 ; 5) ; false ; null]
[bat(0) : Battery | k("bat") |-> nd(capacity) ; false ; null ]
[id(1): Troll | k("goal") |-> (0 ; 5) ; false ; null]
[bat(1) : Battery | k("bat") |-> nd(capacity) ; false ; null ]
[field : Field | (k(( 0 ; 5 )) |-> d(id(0)) , k(( 5 ; 5 )) |-> d(id(1)))
    ; false ; null]] .
```

The initial system term with the protocol is given by:

```
eq init = [[id(0): Troll | k("goal") |-> (5 ; 5) ; false ; null]
[bat(0) : Battery | k("bat") |-> nd(capacity) ; false ; null ]
[id(1): Troll | k("goal") |-> (0 ; 5) ; false ; null]
[bat(1) : Battery | k("bat") |-> nd(capacity) ; false ; null ]
[swap(id(0),id(1)) : Protocol | k("state") |-> ds({q(0)}), k("recv") |->
    recv({}) ; false ; null]
[field : Field | (k(( 0 ; 5 )) |-> d(id(0)) , k(( 5 ; 5 )) |-> d(id(1)))
    ; false ; null]] .
```

We analyze in Maude two scenarios. In one, each robot has as strategy to take the shortest path to reach its goal. As a consequence, a robot reads its position, computes the shortest path, and submits a set of optimal actions. A robot can sense an obstacle on its direct next location, which then allows for sub-optimal lateral moves (e.g., if the obstacle is in the direct next position in the West direction, the robot may go either North or South). In the other scenario, we add a protocol that swaps the two robots if robot `id(0)` is on the direct next location on the west of robot `id(1)`. The swapping is a sequence of moves that ends in an exchange of positions of the two robots.

In the two scenarios, we analyze the behavior of the resulting system with two queries. The first query asks if the system can reach a state in which the energy level of the two batteries is 0, which means that its robot can no longer move:

```
search [1] init =>* [sys::Sys
       [ bat(1) : Battery | k(level) |-> 0 ; true ; null],
       [ bat(2) : Battery | k(level) |-> 0 ; true ; null]] .
```

The second query asks if the system can reach a state in which the two robots successfully reached their goals, and end in the expected locations:

```
search [1] init =>* [sys::Sys  [ field : Field | k(( 5 ; 5 ))
  |-> d(id(0)), k(( 0 ; 5 )) |-> d(id(1)) ; true ; null]] .
```

As a result, when the protocol is absent, the two robots can enter in a livelock behavior and eventually fail with an empty battery:

```
Solution 1 (state 80)
states: 81  rw: 223566 in 73ms cpu (74ms real) (3053554 rw/s)
```

Alternatively, when the protocol is used, the livelock is removed using exogenous coordination. The two robots therefore successfully reach their end locations, and stop before running out of battery:

```
No solution. states: 102
rewrites: 720235 in 146ms cpu (145ms real) (4920041 rw/s)
```

In both cases, the second query succeeds, as there exists a path for both scenarios where the two robots reach their end goal locations. The results can be reproduced by downloading the archive at [1].

Liveness property: patrolling trolls A strategy ranks the action of an agent with respect to some internal measure. For instance, a `TROLL` agent prefers an action that moves it closer to its goal.

```
fmod STRATEGY is
  inc TROLL-INTERFACE .
  inc AGENT{ASemiring} .
  ...
  *** Valuation of an action based on the state of the agent M, and some
      additional measures (current goal,
  ***  distance to its goal, obstacles on the next cells).
  ceq getValue(id, M, a) = (id, (a; getResources(id, a)), 2)
      if isMove?(a)  /\
         M[k("read")] == nd(1) /\
         M[k("pos")] =/= undefined /\
         closest(a, M) /\
         enabled?(a , M) /\
         M[k("pos")] =/= M[k("goal")] /\
         M[k("charging")] =/= nd(1) .
  ceq getValue(id, M, a) = (id, (a; getResources(id, a)), 1)
      if isMove?(a) /\
         M[k("read")] == nd(1) /\
         M[k("pos")] =/= undefined /\
```

```
        not closest(a, M) /\
        a a' := neighbors(a, M) /\
        enabled?(a, M) /\
        M[k("pos")] =/= M[k("goal")] /\
        M[k("charging")] =/= nd(1) .
  ...
endfm
```

We verify the property of liveness for the system, i.e., that the two robots can always eventually swap their positions. Using the Maude search engine, we look for a state for which the two robots have an empty battery level. We find at least one solution for such state:

```
search [1] in SCENARIO : init =>* [sys::Sys
[bat(0) : Battery | k("bat") |-> nd(0) ; true ; null]
[bat(1) : Battery | k("bat") |-> nd(0) ; true ; null]] .

Solution 1 (state 389)
states: 390  rewrites: 1739739 in 299ms cpu (301ms real) (5803637
    rewrites/second)
sys::Sys -->
[field : Field | k(2 ; 3) |-> d(id(0)), k(3 ; 3) |-> d(id(1)) ; true ;
    null]
[id(0) : Troll | k("bat") |-> nd(1), k("goal") |-> 5 ; 5, k("next") |-> 0
     ; 5, k("pos") |-> 2 ; 2, k("read") |-> nd(0)
    ; false ; null]
[id(1) : Troll | k("bat") |-> nd(1), k("goal") |-> 0 ; 5, k("next") |-> 5
     ; 5, k("pos") |-> 3 ; 2, k("read") |-> nd(0)
    ; false ; null]
```

For the liveness property, the `TROLL` changes its goal while it reaches its first objective. Additionally, the `TROLL` agent has an additional `charge` action that, when located on the station, charges its corresponding battery.

```
fmod TROLL-ALT is
  inc TROLL .
  inc STRATEGY .

  var id : Identifier .
  var M M' upds : MapKD .
  var a : AName .
  var names : ANames .
  var actionoutput : IdStates .
  var sn : SensorNames .
  var sensorvalues : IdStates .
  vars d1 d2 : Data .
```

```
  op swapGoal? : MapKD -> MapKD .
  ceq swapGoal?(M) = insert( k("next"), d1, insert(k("goal"), d2, M))
      if d1 := M[k("goal")] /\
         d2 := M[k("next")] /\
         M[k("pos")] == M[k("goal")] .
  eq swapGoal?(M) = M [owise] .

  *** Alternates between read an write actions
  ceq getPostState(id, Troll, id, a, actionoutput, M) = insert(k("read"),
       nd(0), M) if isMove?(a) .

  ceq getPostState(id, Troll, id, readSensors(sn), sensorvalues, M) = M'
      if upds := getSensorValues(getResources(id, readSensors(sn)) ,
          sensorvalues) /\
         M' := insert(k("read"),  nd(1),
               insert(k("bat"), upds[k("bat")],
               insert(k("station"), upds[k("station")],
               insert(k("pos"), upds[k("pos")], M))))  .

  ceq getPostState(id, Troll, id, lock, actionoutput, M) =  M'
      if M' := insert(k("charging"), nd(1), M) .
  ceq getPostState(id, Troll, id, unlock, actionoutput, M) = M'
      if M' := insert(k("charging"), nd(0), M) .

  eq getPostState(id, Troll, id, end, actionoutput, M) = swapGoal?(M) .

  ceq internalUpdate(id, Troll, M) = insert(k("read"), nd(1), M) if M[k("
      read")] == nd(0) .
  ceq internalUpdate(id, Troll, M) = insert(k("read"), nd(0), M) if M[k("
      read")] == nd(1) .

  eq computeActions(id , Troll , M ) = getSoftActions(id, M ,
      trollActions(id, M)) .
  ceq getSoftActions( id , M , a names ) = getValue(id, M, a) +
      getSoftActions( id , M ,names ) if a =/= void .
  eq getSoftActions( id , M , void) = null .


endfm
```

We change `SCENARIO` to now use the new `TROLL-ALT` module. We search if both trolls can run out of energy:

```
search [1] in SCENARIO-STATION : init =>* [sys::Sys
[bat(0) : Battery | k("bat") |-> nd(0) ; true ; null]
```

```
[bat(1) : Battery | k("bat") |-> nd(0) ; true ; null]] .

No solution.
states: 280  rewrites: 1155509 in 189ms cpu (189ms real) (6091179
    rewrites/second)
```

The new strategy for the robot changes the overall behavior of the composition, and makes the liveness property true.

Self-sorting property The property P_{sorted} is a reachability property on the state of the grid, that states that *eventually, all robots are in the sorted position.* In Maude, given a system of 3 robots, we express such reachability property with the following search command:

```
search [1] init =>*
   [sys::Sys  [ field : Field |
                k((0;0)) |-> d(id(0)),
                k((1;0)) |-> d(id(1)),
                k((2;0)) |-> d(id(2)) ; true ; null]] .
```

The initial configuration of the grid is such that robot 0 is on location $(2;0)$, robot 1 on $(1;0)$, and robot 2 on $(0;0)$. Since the grid is of size 3 by 2, robots need to coordinate to reach the desired sorted configuration.

Table 5.2 features three variations on the sorting problem. The first system is composed of robots whose move are free on the grid. The second adds one battery for each component, whose energy level decreases for each robot move. The third system adds a swap protocol for every pair of two robots. The last system adds protocol and batteries to compose with the robots.

We record, for each of those systems, whether the sorted configuration is reachable (P_{sorted}), and if all three robots can run out of energy (P_{bat}). Observe that the reachability query returns a solution for both system: the one with and without protocols. However, the time to reach the first solution increases as the number of states and transition increases (adding the protocol components). We leave as future work some optimizations to improve on our results.

Table 5.2: Evaluation of different systems for the P_{sorted} and P_{bat} behavioral properties, where *st.* stands for states, *rw* for rewrites. Note that the P_{bat} property is not evaluated when the system does not contain battery components.

System	P_{sorted}	P_{bat}
$\bowtie_{0\leq i\leq 2} R_i \bowtie G$	12.10^3 st., 25s, 31.10^6 rw	
$\bowtie_{0\leq i\leq 2} (R_i \bowtie B_i) \bowtie G$	12.10^3 st., 25s, 31.10^6 rw	true
$\bowtie_{0\leq i\leq 2} R_i \bowtie G \bowtie_{0\leq i<j\leq 2} S(R_i, R_j)$	8250 st., 44s, 80.10^6 rw	
$\bowtie_{0\leq i\leq 2} (R_i \bowtie B_i) \bowtie G \bowtie_{0\leq i<j\leq 2} S(R_i, R_j)$	8250 st., 71s, 83.10^6 rw	false

Chapter 6

Conclusion

Modeling and analysis of cyber-physical systems are still challenging [52]. One reason is that cyber-physical systems involve many different parts (cyber or physical), of different nature (discrete or continuous), and in constant interaction via sensing and actuating. This thesis proposes a semantic framework in which both the behavior of cyber-physical systems and their interaction can be expressed. Below, we give a short summary of the contribution for each chapter, and provide a list of points for future work.

The component based model introduced in Chapter 2 proposes to explicitly model interactions in cyber-physical systems with parametrized algebraic operators, particularly for composition and decomposition. In this model, components are terms and interactions are captured by user defined operations on components. Properties of operators, such as associativity, commutativity, and idempotency, are expressed in terms of the properties of their interaction signatures. For instance, an interaction constraint that is symmetric leads to a commutative product on component. To ease the specification of interaction constraints, a co-inductive construction of composability relations is proposed to specify constraints on Timed-Event Streams given a constraint on observations. Similar algebraic properties for a co-inductively defined operator on components are deduced given properties of the underlying constraints on observations. The model not only provides ways to compose components, but also allows, under certain conditions, for decomposition of components. An operator of division is introduced that selects, out of a set of candidates, a component that in product with the divisor give back the dividend. Several such division operators are possible as each involves a way of choosing a component within a set of candidates.

In the case where a metric is used to order components, the division operator can be defined as taking the best (defined by the metric) of the candidate components. The cost of coordination is discussed as such a possible metric.

We instantiate our component model in Chapter 3 on a set of cyber components whose behaviors are independent of time. We give an algebraic semantics of the Reo coordination language, where ports are components and circuits are product of ports under some interaction signatures. The result adds some strength to existing work on Reo, by providing a suitable algebraic framework to show equivalence of connector behaviors in Reo. Within the class of order sensitive components, we consider two meaningful classes: transactional components that have observations with more than one event, and linear components that have observations with at most one event. The former class of components is adequate to represent behavior at a design/specification level where events within the same set are declared to be atomic; the latter class of components represents sequential machines that can generate only one event at a time. We study translation of (product of) transactional components into a (product of) linear components, which leads to the definition of correctness criteria for implementation of high-level transactional specifications of concurrent systems into their behaviorally equivalent implementations on (sets of interacting) sequential machines. Two instances of correct and compositional translations are defined.

Finally, we study temporal properties of some order sensitive components. Particularly, we consider temporal properties of Reo connectors, specifically to verify data flow properties of composite connectors. For that purpose, we give a translation to generate an executable in Promela from a logical specification of Reo connectors, using which the model checker Spin verifies properties written in Linear Temporal Logic (LTL). To ease the specification of LTL properties, we introduce a domain specific language for data flow properties, e.g., the temporal property that specifies the simultaneous or exclusive firing of a port.

In order to make our component model executable, in Chapter 4 we present several steps that lead to an operational yet compositional specification of components. First, we operationally define the specification of a component behavior, i.e., a set of Timed-Event Stream, as the semantics of a labeled transition system called a TES transition system. We introduce several operators on TES transition systems, and show them to semantically correspond to their component products: the behavior of a syntactic product of TES transition systems is the behavior of the semantic product of their corresponding two components.

While a TES transition system is not necessarily finitely representable (its sets

of states and transitions may be infinite), it serves as a first step towards a finite executable model, and gives some results as to when the product of two TES transition systems can be done lazily at runtime without deadlock.

In order to have a finite executable specification of components, we introduce agents as rewriting theories for component behaviors. An agent specifies finitely, with a set of equations and a rewrite rule, a TES transition system, which ultimately denotes a component. A system of agents runs each agent concurrently, while ensuring that each agent performs composable actions. As a result, the behavior of a system of agents is shown to coincide with the product (under the composability relation imposed by the system) of the behaviors of each agent.

Finally, to reduce the large state space involved in cyber-physical systems, we introduce a framework to extend agents with preferences. The use of preferences allows for ordering actions that an agent can perform. Thus, actions that are possible but less preferred can be discarded and therefore reduce the state space. Notions of compromise emerge out of the composition of agents with different preferences for their actions.

To demonstrate the utility of our component model, in Chapter 5 we give an implementation of the rewriting framework in Maude. We define agent as a module that implements a set of operations for interacting with other agents. A system module runs all agents concurrently, and ensures composability of their actions. To demonstrate the usefulness of this framework, we present three main applications. First, we specify ports and connectors in Reo as agents that run concurrently. As a result, a Reo circuit can be composed at runtime, and several search queries can be performed to extract properties of protocols. Next, we consider a system consisting of a controller, a valve, and N water reservoirs. The controller needs to move the valve in order to keep the water level in the reservoirs within some thresholds. We contrast the safety property of the controller not seeing any invalid states with the possibility for the controller to miss some states. In the latter case, the safety property may still be valid from the controller's perspective, while it effectively violates the property as the water level of the reservoirs may unobservably go out of bound. Finally, we analyse a system consisting of a set of robots, each equipped with a battery, running on a shared field. More specifically, we consider liveness and safety properties, such as the possibility for a robot to patrol through a set of points without running out of battery, or to coordinate with another robot to swap position and get in a sorted position.

A set of challenges emerge from the work presented in the chapters of this thesis that we consider as relevant future work. Hereafter is a list of few points, organized

per chapters, that are considered relevant for future investigation.

1.1 Metrics for division.

Our algebra of components introduces a family of operators to compose components into larger systems. Dually, the operation of decomposition is equally important to extract some structure from a composite component, and eventually update some of its parts. For that matter, the operation of division is defined to return one of the possible quotients. As the division operator assumes a choice function over a set of components, several metrics can be used to partially order components, and therefore act as the choice function for the operator of division. One of such metric, discussed in Section 2.2, is the *cost of coordination* that orders components with respect to the amount of interaction required during composition. As future work, other measures may be considered, as well as their corresponding implication on system's decomposition.

1.2 Laws of distribution.

The algebraic properties of operators in the algebra of components in Section 2.1 are mostly studied independently (i.e., commutativity, associativity, idempotency). However, the law of distribution of one product over another product would allow for additional reasoning on component expression by showing equivalence between different types of interaction. More precisely, being able to factorize or distribute an operation may practically reflect the distinction between the cases where a group of components needs global as opposed to local interaction. Considering for instance the operation of division, the distribution of division over multiplication may, in some cases, allow for modular decomposition. Moreover, on the level of components, distribution of a component over a product could capture the fact that such component is independent of the interaction constraints imposed by that product. As a result, such distributive law can reveal mechanisms to split a protocol component into subparts, that get distributed onto the smallest set of components that needs their coordination.

2.1 Extension of Reo with cyber-physical connectors.

In Reo, the main primitive of interaction is a port, which serves as the junction through which data flow between a component and its environment. By definition, a port in Reo observes the transport of a finite amount of data within a time interval, which precludes the possibility of continuous physical data transfer. Instead, a physical phenomenon must be encapsulated into a component that samples the physics at a fix rate. Only then can a port transport the value, at such rate, to other Reo components. Note that the choice of the sampling rate for the component encapsulating a physical

process will influence the overall protocol, as other components may not be able to see some relevant data, or take action accordingly. Alternatively, introducing a cyber-physical variant of a port, with sampling as a parameter, leads to the possibility of having cyber and physical components in Reo. Instead of the standard semantics where the firing of a port occurs if and only if a data of a port is present, a cyber-physical port is equipped with a sampler that can only observe a datum within a time period. Similarly, the composition operator can be extended such that ports tune their sampling rates for transmission. As a result, such cyber-physical extension of Reo may enable compositional construction to find the highest frequency (i.e., smallest amount of observations) for samplers while not missing any important event.

2.2 Temporal properties as Reo connectors.

The definition of components allows both executable specifications and properties to be denoted as such. As a result, the structure of our algebra can be exploited to specify temporal properties compositionally. More precisely, as already hinted in Section 3.3, the use of graphical and compositional primitives of Reo can serve as a starting point for languages to construct temporal properties. As a result of such modular specification, ways of distributing the property over components, or decomposing the property into a set of simpler properties may facilitate modular verification.

3.1 Real time extension of agents.

The current specification of agents uses rewriting logic without real time. Agents run in steps, and may interact with other agents within each step. The time that lapses between two steps is a constant that is fixed at the beginning of the execution. Alternatively, our framework can be extended with real time variables, that account for the variable time that lapses between two steps. As well, making the time of a step explicit while having a modular specification of each agent may facilitate certain meta level reasoning, such as finding the largest sampling rate such that resulting system satisfies, for instance, a safety property.

3.2 Fusion of agents.

Each composition operator of our algebra captures a constraint of interaction between two components. Practically, such an interaction constraint may represent actual physical constraints (e.g., spatial, time, hardware) that enforce the practical observations to be related. For instance, two components tied with each other by a synchronous product may be realised by two processes running on the same machine. Following the result of Section 4.2, agents denote components, and the product of agents denotes the product of their corresponding components. In some cases, it can

be useful to define a new agent out of the product of two agents, such that the newly formed agent denotes the same behavior as the product component. For instance, forming such new agent may reduce the search space, or improve some runtime measures, while preserving the system behavior. Practically, such transformation acts on agent theories, and needs to generate a new agent specification out of two agent specifications.

3.3 Symbolic execution under a class of environments.

The rewriting framework developed in Section 4.2 currently assumes that the collection of agents forming a system is *closed*. A closed system is such that every agent's action is matched with that of its corresponding recipient agent in the set. While this condition holds for many practical examples, allowing for *open* system analysis is still desirable. In an open system, one can analyse the behavior of an agent under a class of environments, and conclude which of a set of potential interacting agents complies or conflicts with some system properties.

3.4 Implementability of division.

Our algebra of components introduces several operations on behaviors that are shown to be useful for design and modular analysis. One such operation is the operation of division, which enables reasoning about alternative compositions that preserve the same behavior. The operation of division does not yet enjoy the same operational specification as the operation of composition. Studying how to implement the operation of division within our agent framework may enable various important reasoning schemes for updates and fault diagnosis. Practically, the operator of division may reveal some independences within an agent, and factor such agents into two independent parts. The two specifications may be run in parallel, speeding up the overall execution time.

Bibliography

[1] https://scm.cwi.nl/FM/cp-agent.

[2] Bowen Alpern and Fred B. Schneider. "Defining liveness". In: *Information Processing Letters* 21.4 (1985), pp. 181–185. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(85)90056-0.

[3] F. Arbab and J. J. M. M. Rutten. "A Coinductive Calculus of Component Connectors". In: *Recent Trends in Algebraic Development Techniques*. Ed. by Martin Wirsing, Dirk Pattinson, and Rolf Hennicker. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 34–55. ISBN: 978-3-540-40020-2.

[4] Farhad Arbab. "Proper Protocol". In: *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*. Ed. by Erika Ábrahám, Marcello M. Bonsangue, and Einar Broch Johnsen. Vol. 9660. Lecture Notes in Computer Science. Springer, 2016, pp. 65–87. DOI: 10.1007/978-3-319-30734-3_7.

[5] Farhad Arbab. "Puff, The Magic Protocol". In: *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*. Ed. by Gul Agha, Olivier Danvy, and José Meseguer. Vol. 7000. Lecture Notes in Computer Science. Springer, 2011, pp. 169–206. DOI: 10.1007/978-3-642-24933-4_9.

[6] Farhad Arbab. "Reo: A Channel-based Coordination Model for Component Composition". In: *Mathematical. Structures in Comp. Sci.* 14.3 (June 2004), pp. 329–366. ISSN: 0960-1295. DOI: 10.1017/S0960129504004153.

[7] Farhad Arbab and Jan J. M. M. Rutten. "A Coinductive Calculus of Component Connectors". In: *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, September 24-27, 2002, Revised Selected Papers*. Ed. by Martin Wirsing, Dirk Pattinson,

and Rolf Hennicker. Vol. 2755. Lecture Notes in Computer Science. Springer, 2002, pp. 34–55. DOI: 10.1007/978-3-540-40020-2_2.

[8] Farhad Arbab and Francesco Santini. "Preference and Similarity-Based Behavioral Discovery of Services". In: *Web Services and Formal Methods - 9th International Workshop, WS-FM 2012, Tallinn, Estonia, September 6-7, 2012, Revised Selected Papers*. Ed. by Maurice H. ter Beek and Niels Lohmann. Vol. 7843. Lecture Notes in Computer Science. Springer, 2012, pp. 118–133. DOI: 10.1007/978-3-642-38230-7_8.

[9] Farhad Arbab et al. "Models and temporal logical specifications for timed component connectors". In: *Software & Systems Modeling* 6.1 (Mar. 2007), pp. 59–82. ISSN: 1619-1374. DOI: 10.1007/s10270-006-0009-9.

[10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The Internet of Things: A survey". In: *Comput. Networks* 54.15 (2010), pp. 2787–2805. DOI: 10.1016/j.comnet.2010.05.010.

[11] J.C.M. Baeten, T. Basten, and M.A. Reniers. *Process algebra : equational theories of communicating processes*. English. Cambridge tracts in theoretical computer science. United Kingdom: Cambridge University Press, 2010. ISBN: 978-0-521-82049-3. DOI: 10.1017/CBO9781139195003.

[12] Jos C. M. Baeten and Cornelis A. Middelburg. "Real time process algebra with time-dependent conditions". In: *J. Log. Algebraic Methods Program.* 48.1-2 (2001), pp. 1–38. DOI: 10.1016/S1567-8326(01)00004-2.

[13] Christel Baier, Joachim Klein, and Sascha Klüppelholz. "Synthesis of Reo Connectors for Strategies and Controllers". In: *Fundamenta Informaticae* 130 (Jan. 2014), pp. 1–20. DOI: 10.3233/FI-2014-980.

[14] Christel Baier et al. "A Uniform Framework for Modeling and Verifying Components and Connectors". In: *Coordination Models and Languages*. Ed. by John Field and Vasco T. Vasconcelos. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 247–267. ISBN: 978-3-642-02053-7.

[15] Christel Baier et al. "Design and Verification of Systems with Exogenous Coordination Using Vereofy". In: *Leveraging Applications of Formal Methods, Verification, and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 97–111. ISBN: 978-3-642-16561-0.

[16] Christel Baier et al. "Formal Verification for Components and Connectors". In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelaine. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 82–101. ISBN: 978-3-642-04167-9.

[17] Christel Baier et al. "Modeling component connectors in Reo by constraint automata". In: *Science of Computer Programming* 61.2 (2006). Second International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'03), pp. 75–113. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2005.10.008.

[18] L. S. Barbosa. "Components as coalgebras". PhD thesis. University of Minho, 2001.

[19] Barbara Rita Barricelli, Elena Casiraghi, and Daniela Fogli. "A Survey on Digital Twin: Definitions, Characteristics, Applications, and Design Implications". In: *IEEE Access* 7 (2019), pp. 167653–167671. DOI: 10.1109/ACCESS.2019.2953499.

[20] J.A. Bergstra and J.W. Klop. "Process algebra for synchronous communication". In: *Information and Control* 60.1 (1984), pp. 109–137. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(84)80025-X.

[21] Gérard Berry. "The foundations of Esterel". In: *Proof, Language, and Interaction, Essays in Honour of Robin Milner*. Ed. by Gordon D. Plotkin, Colin Stirling, and Mads Tofte. The MIT Press, 2000, pp. 425–454. ISBN: 978-0-262-16188-6.

[22] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. "Semiring-based constraint satisfaction and optimization". In: *J. ACM* 44.2 (1997), pp. 201–236. DOI: 10.1145/256303.256306.

[23] Stephen D. Brookes, C. A. R. Hoare, and A. W. Roscoe. "A Theory of Communicating Sequential Processes". In: *J. ACM* 31.3 (1984), pp. 560–599. DOI: 10.1145/828.833.

[24] Paul Caspi et al. "Lustre: A Declarative Language for Programming Synchronous Systems". In: *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 1987, pp. 178–188. DOI: 10.1145/41625.41641.

[25] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems, Second Edition*. Springer, 2008. ISBN: 978-0-387-33332-8. DOI: 10.1007/978-0-387-68612-7.

[26] D. M. Chapiro. "Globally-asynchronous locally-synchronous systems". PhD thesis. Stanford Univ., CA., Oct. 1984.

[27] Taolue Chen et al. "A Compositional Specification Theory for Component Behaviours". In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Ed. by Helmut Seidl. Vol. 7211. Lecture Notes in Computer Science. Springer, 2012, pp. 148–168. DOI: 10.1007/978-3-642-28869-2_8.

[28] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties". In: *J. Comput. Secur.* 18.6 (Sept. 2010), pp. 1157–1210. ISSN: 0926-227X.

[29] Manuel Clavel et al., eds. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Vol. 4350. Lecture Notes in Computer Science. Springer, 2007. ISBN: 978-3-540-71940-3. DOI: 10.1007/978-3-540-71999-1.

[30] Yuri Gil Dantas, Vivek Nigam, and Carolyn L. Talcott. "A Formal Security Assessment Framework for Cooperative Adaptive Cruise Control". In: *IEEE Vehicular Networking Conference*. IEEE, 2020, pp. 1–8. DOI: 10.1109/VNC51378.2020.9318334.

[31] K. Dokter and F. Arbab. "Treo: Textual Syntax for Reo Connectors". In: *ArXiv e-prints* (June 2018).

[32] Kasper Dokter and Farhad Arbab. "Rule-Based Form for Stream Constraints". In: *Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018. Proceedings*. Ed. by Giovanna Di Marzo Serugendo and Michele Loreti. Vol. 10852. Lecture Notes in Computer Science. Springer, 2018, pp. 142–161. DOI: 10.1007/978-3-319-92408-3_6.

[33] José Fiadeiro et al. "Dynamic networks of heterogeneous timed machines". In: *Mathematical Structures in Computer Science* 28.6 (2018), pp. 800–855. DOI: 10.1017/S0960129517000135.

[34] Marie-Paule Flé. "Serialization of Concurrent Programs". In: *J. Comput. Syst. Sci.* 38.3 (1989), pp. 474–493. DOI: 10.1016/0022-0000(89)90012-3.

[35] Wan J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2000. ISBN: 978-3-540-66579-3. DOI: 10.1007/978-3-662-04293-9.

[36] Paul Gastin. "Infinite Traces". In: *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science, La Roche Posay, France, April 23-27, 1990, Proceedings*. Ed. by Irène Guessarian. Vol. 469. Lecture Notes in Computer Science. Springer, 1990, pp. 277–308. ISBN: 3-540-53479-2. DOI: 10.1007/3-540-53479-2_12.

[37] Heiko Hamann. *Swarm Robotics - A Formal Approach*. Springer, 2018. ISBN: 978-3-319-74526-8. DOI: 10.1007/978-3-319-74528-2.

[38] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. First. Addison-Wesley Professional, 2003. ISBN: 0-321-22862-6.

[39] Ryszard Janicki et al. "Step traces". In: *Acta Informatica* 53.1 (2016), pp. 35–65. DOI: 10.1007/s00236-015-0244-z.

[40] Theo M. V. Janssen and Thomas Ede Zimmermann. "Montague Semantics". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2021. Metaphysics Research Lab, Stanford University, 2021.

[41] S.-S. T. Q. Jongmans. "Automata-theoretic protocol programming". PhD thesis. Leiden University, 2016.

[42] Sung-Shik T. Q. Jongmans and Farhad Arbab. "Overview of Thirty Semantic Formalisms for Reo". In: *Sci. Ann. Comput. Sci.* 22.1 (2012), pp. 201–251. DOI: 10.7561/SACS.2012.1.201.

[43] Sung-Shik T. Q. Jongmans and Farhad Arbab. "Overview of Thirty Semantic Formalisms for Reo". In: *Sci. Ann. Comp. Sci.* 22 (2012), pp. 201–251.

[44] Tobias Kappé, Farhad Arbab, and Carolyn L. Talcott. "A Compositional Framework for Preference-Aware Agents". In: *Proceedings of the The First Workshop on Verification and Validation of Cyber-Physical Systems, V2CPS@IFM 2016, Reykjavik, Iceland, June 4-5, 2016*. Ed. by Mehdi Kargahi and Ashutosh Trivedi. Vol. 232. EPTCS. 2016, pp. 21–35. DOI: 10.4204/EPTCS.232.6.

[45] Tobias Kappé et al. "Soft component automata: Composition, compilation, logic, and verification". In: *Sci. Comput. Program.* 183 (2019). DOI: 10.1016/j.scico.2019.08.001.

[46] S. Kemper. "SAT-based verification for timed component connectors". In: *Science of Computer Programming* 77.7 (2012). (1) FOCLASA'09 (2) FSEN'09, pp. 779–798. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2011.02.003.

[47] Stephanie Kemper. "Compositional Construction of Real-Time Dataflow Networks". In: *Coordination Models and Languages*. Ed. by Dave Clarke and Gul Agha. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 92–106. ISBN: 978-3-642-13414-2.

[48] S. C. Kleene. "Representation of Events in Nerve Nets and Finite Automata". In: *Automata Studies. (AM-34), Volume 34*. Ed. by C. E. Shannon and J. McCarthy. Princeton: Princeton University Press, 2016, pp. 3–42. DOI: doi:10.1515/9781400882618-002.

[49] Natallia Kokash, Christian Krause, and Erik de Vink. "Reo + mCRL2 : A framework for model-checking dataflow in service compositions". In: *Formal Aspects of Computing* 24.2 (Mar. 2012), pp. 187–216. ISSN: 1433-299X. DOI: 10.1007/s00165-011-0191-6.

[50] Natallia Kokash, Christian Krause, and Erik P. de Vink. "Verification of Context-Dependent Channel-Based Service Models". In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 21–40. ISBN: 978-3-642-17071-3.

[51] Stéphane Lafortune. "Discrete Event Systems: Modeling, Observation, and Control". In: *Annual Review of Control, Robotics, and Autonomous Systems* 2.1 (2019), pp. 141–159. DOI: 10.1146/annurev-control-053018-023659.

[52] Edward A. Lee. "Cyber Physical Systems: Design Challenges". In: *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008), 5-7 May 2008, Orlando, Florida, USA*. IEEE Computer Society, 2008, pp. 363–369. DOI: 10.1109/ISORC.2008.25.

[53] Jaehun Lee et al. "HybridSynchAADL: Modeling and Formal Analysis of Virtually Synchronous CPSs in AADL". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 491–504. DOI: 10.1007/978-3-030-81685-8_23.

[54] Benjamin Lion. *Cyber-physical agent framework in Maude*. May 2022. DOI: 10.5281/zenodo.6592275.

[55] Benjamin Lion. *Treo to Promela*. Dec. 2022. DOI: 10.5281/zenodo.7393621.

[56] Benjamin Lion, Farhad Arbab, and Carolyn Talcott. *Runtime Composition Of Systems of Interacting Cyber-Physical Components*. 2022. DOI: 10.48550/ARXIV.2205.13008.

[57] Benjamin Lion, Farhad Arbab, and Carolyn L. Talcott. "A Semantic Model for Interacting Cyber-Physical Systems". In: *Proceedings 14th Interaction and Concurrency Experience, ICE 2021, Online, 18th June 2021*. Ed. by Julien Lange et al. Vol. 347. EPTCS. 2021, pp. 77–95. DOI: 10.4204/EPTCS.347.5.

[58] Benjamin Lion, Samir Chouali, and Farhad Arbab. "Compiling Protocols to Promela and Verifying their LTL Properties". In: *Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018*. 2018, pp. 31–39.

[59] Benjamin Lion, Samir Chouali, and Farhad Arbab. *Environment for verification of Reo protocols with SPIN model checker*. July 2019. DOI: 10.5281/zenodo.3265435.

[60] Benjamin Lion and Kasper Dokter. *The Reo Coordination Language*. https://github.com/ReoLanguage/Reo. 2019.

[61] Ian A. Mason et al. "A Framework for Analyzing Adaptive Autonomous Aerial Vehicles". In: *Software Engineering and Formal Methods Collocated Workshops: DataMod, FAACS, MSE, CoSim-CPS, and FOCLASA, Revised Selected Papers*. Ed. by Antonio Cerone and Marco Roveri. Vol. 10729. Lecture Notes in Computer Science. Springer, 2017, pp. 406–422. DOI: 10.1007/978-3-319-74781-1_28.

[62] Antoni W. Mazurkiewicz. "Introduction to Trace Theory". In: *The Book of Traces*. 1995.

[63] José Meseguer. "Conditioned Rewriting Logic as a United Model of Concurrency". In: *Theor. Comput. Sci.* 96.1 (1992), pp. 73–155. DOI: 10.1016/0304-3975(92)90182-F.

[64] José Meseguer. "Twenty years of rewriting logic". In: *J. Log. Algebraic Methods Program.* 81.7-8 (2012), pp. 721–781. DOI: 10.1016/j.jlap.2012.06.003.

[65] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989. ISBN: 978-0-13-115007-2.

[66] L. Moormann et al. "Supervisory control synthesis for large-scale systems with isomorphisms". In: *Control Engineering Practice* 115 (2021), p. 104902. ISSN: 0967-0661. DOI: https://doi.org/10.1016/j.conengprac.2021.104902.

[67] Maurice Nivat. "Behaviors of Processes and Synchronized Systems of Processes". In: *Theoretical Foundations of Programming Methodology: Lecture Notes of an International Summer School, directed by F. L. Bauer, E. W. Dijkstra and C. A. R. Hoare*. Ed. by Manfred Broy and Gunther Schmidt. Dordrecht: Springer Netherlands, 1982, pp. 473–551. ISBN: 978-94-009-7893-5. DOI: 10.1007/978-94-009-7893-5_14.

[68] Peter Csaba Ölveczky. "Real-Time Maude and Its Applications". In: *Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers*. Ed. by Santiago Escobar. Vol. 8663. Lecture Notes in Computer Science. Springer, 2014, pp. 42–79. DOI: 10.1007/978-3-319-12904-4_3.

[69] Bahman Pourvatan et al. "Decomposition of Constraint Automata". In: *Formal Aspects of Component Software -7th International Workshop, FACS 2010, Guimarães, Portugal, October 14-16, 2010, Revised Selected Papers*. Ed. by Luís Soares Barbosa and Markus Lumpe. Vol. 6921. Lecture Notes in Computer Science. Springer, 2010, pp. 237–258. DOI: 10.1007/978-3-642-27269-1_14.

[70] José Proença. "Synchronous coordination of distributed components". PhD thesis. Leiden University, 2011.

[71] José Proença et al. "Dreams: a framework for distributed synchronous coordination". In: *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*. Ed. by Sascha Ossowski and Paola Lecca. ACM, 2012, pp. 1510–1515. ISBN: 978-1-4503-0857-1. DOI: 10.1145/2245276.2232017.

[72] Jean-François Raskin. "An Introduction to Hybrid Automata". In: *Handbook of Networked and Embedded Control Systems*. Ed. by Dimitrios Hristu-Varsakelis and William S. Levine. Birkhäuser, 2005, pp. 491–518.

[73] A. W. Roscoe. *The Theory and Practice of Concurrency*. USA: Prentice Hall PTR, 1997. ISBN: 0136744095.

[74] Meera Sampath, Stéphane Lafortune, and Demosthenis Teneketzis. "Active diagnosis of discrete-event systems". In: *IEEE Trans. Autom. Control.* 43.7 (1998), pp. 908–929. DOI: 10.1109/9.701089.

[75] Harald Schöning. "Industry 4.0". In: *it Inf. Technol.* 60.3 (2018), pp. 121–123. DOI: 10.1515/itit-2018-0015.

[76] *Stanford Encyclopedia of Philosophy: The Axiom of Choice*. https://plato.stanford.edu/entries choice/. Accessed: 2022-06-08.

[77] Zoltán Gendler Szabó. "Compositionality". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Fall 2022. Metaphysics Research Lab, Stanford University, 2022.

[78] Carolyn Talcott, Farhad Arbab, and Maneesh Yadav. "Soft Agents: Exploring Soft Constraints To Model Robust Adaptive Distributed Cyber-Physical Agent Systems". In: *Software, Services, and Systems - Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering*. Vol. 8950. LNCS. Springer, 2015.

[79] S. Tripakis et al. "A modular formal semantics for Ptolemy†". In: *Mathematical Structures in Computer Science* 23 (2013), pp. 834–881.

[80] Stavros Tripakis and Costas Courcoubetis. "Extending promela and spin for real time". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tiziana Margaria and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 329–348. ISBN: 978-3-540-49874-2.

[81] Norbert Wiener. *Cybernetics: or Control and Communication in the Animal and the Machine*. 2nd ed. Cambridge, MA: MIT Press, 1948.

[82] Martin Wirsing et al. "A Rewriting Logic Framework for Soft Constraints". In: *Electr. Notes Theor. Comput. Sci.* 176.4 (2007), pp. 181–197. DOI: 10.1016/j.entcs.2007.06.015.

[83] Xiong Xu et al. "Semantics Foundation for Cyber-Physical Systems Using Higher-Order UTP". In: *ACM Trans. Softw. Eng. Methodol.* (Feb. 2022). Just Accepted. ISSN: 1049-331X. DOI: 10.1145/3517192.