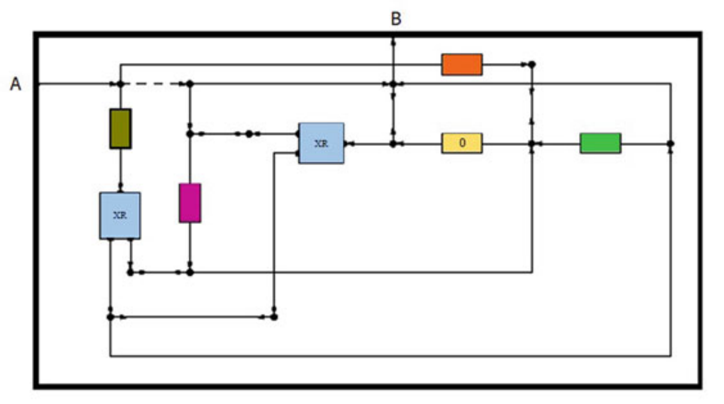
Festschrift

LNCS 10865

Frank de Boer
Marcello Bonsangue
Jan Rutten (Eds.)

It's All About Coordination

Essays to Celebrate the Lifelong Scientific Achievements of Farhad Arbab



Springer

Lecture Notes in Computer Science 10865

More information about this series at http://www.springer.com/series/7408

Frank de Boer · Marcello Bonsangue
Jan Rutten (Eds.)

It's All About Coordination

Essays to Celebrate the Lifelong Scientific Achievements of Farhad Arbab

Springer

Editors

Frank de Boer
Centre for Mathematics
and Computer Science
Amsterdam
The Netherlands

Jan Rutten
Centre for Mathematics
and Computer Science
Amsterdam
The Netherlands

Marcello Bonsangue
Leiden University
Leiden
The Netherlands

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-90088-9 ISBN 978-3-319-90089-6 (eBook)
https://doi.org/10.1007/978-3-319-90089-6

Library of Congress Control Number: 2018939624

LNCS Sublibrary: SL2 – Programming and Software Engineering

Cover illustration: The cover illustration was created by the editors. Used with permission.

Printed on acid-free paper

This Springer imprint is published by the registered company Springer International Publishing AG
part of Springer Nature
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Farhad Arbab

Preface

Having Iranian, American, and Dutch nationality, Farhad Arbab is a citizen of the world, a fact that is reflected not only by his personality and views of life, but also by his scientific career.

Farhad was born in Iran and got his bachelor's degree in chemical engineering at Sharif University in Tehran, in July 1976. One year later, he obtained his master's degree in computer science, again at Sharif, in August 1977. Before and during his university studies, Farhad found the time for various other activities. He worked as a computer systems analyst and as a system engineer at various companies in Tehran, including IBM, and married Hamideh Afsarmanesh, in March 1977.

Newly wed and freshly graduated, Farhad moved to the United States, in the Fall of the same year, to start his PhD studies at the University of California, Los Angeles (UCLA). He obtained his PhD degree in 1982, by defending a thesis with the somewhat mysterious (at least, to some of us) title "Requirements and Architecture of a CAM-Oriented CAD System for Design and Manufacture of Mechanical Parts." In the period 1980–1989, i.e., both during and after his PhD studies, Farhad held various positions at UCLA, and at the University of Southern California (USC). And again, life was more than only science: Hamideh and Farhad became the proud parents of two beautiful daughters, Taraneh in 1985 and Mandana in 1987.

On his way to a third nationality, then, Farhad moved to Amsterdam, in January 1990, to start the Dutch branch of his career at CWI, a position he combined later, in 2014, with an appointment as professor at the University of Leiden.

All in all, Farhad worked in different countries, in various fields, on many diverse subjects, a border-crossing experience which led to scientific explorations in various directions.

In the United States, Farhad worked in the fields of computer graphics, solid modeling, and computer-aided design and manufacturing of mechanical parts. At CWI, in the 1990s, Farhad worked on the design, implementation, and applications of Manifold: a coordination language for managing the interactions among cooperating autonomous concurrent processes in heterogeneous distributed computing environments. Over the years, the scope of Farhad's research at CWI became wider still, including software composition, service-oriented computing, component-based software, concurrency theory, coordination models and languages, parallel and distributed computing, visual programming environments, constraints, logic and object-oriented programming.

Here we would like to highlight the foundational contributions of Farhad to the field of coordination models and languages. His insight that it is all about exogenous coordination gave rise to the IWIM (idealized workers and idealized managers) model for coordination of concurrent activities, which emphasized the basic separation of concerns between computation and coordination. This line of research culminated in the striking elegance and beauty of Reo: an exogenous coordination model whose formal semantics is based on a calculus of channel composition. This powerful declarative model of coordination offers a visual glue-code language for the construction of coordinating connectors in distributed, mobile, and dynamically reconfigurable component-based systems. Reo has been extremely successful and is having a great impact in many of the areas mentioned here, as is illustrated by the high number of citations of Farhad's scientific publications.

We believe that it took someone as talented as Farhad, capable of crossing boundaries, to be able to observe and distill the essence of interaction, communication, and coordination, and to come up with the definition of Reo.

On the occasion of Farhad's retirement, the present volume collects a number of papers by several of Farhad's close collaborators over the years. On behalf of all your friends in science, we wish you all the best, Farhad!

March 2018

Frank de Boer
Marcello Bonsangue
Jan Rutten

Organization

This festschrift was organized by Farhad's colleagues and friends Frank de Boer, Marcello Bonsangue, and Jan Rutten. We are grateful to Susanne van Dam for her help. We acknowledge the Centrum Wiskunde en Informatica (CWI) for the support in the organization of the associated workshop.

Reviewers

Wil van der Aalst
Krzysztof Apt
Christel Baier
Luís S. Barbosa
Kees Bloom
Kasper Dokter
Fabio Gaducci
Jean-Marie Jacquet
Ali Jafari
Einar Broch Johnsen
Sung-Shik Jongmans
Ehsan Khamespanah
Tobias Kappé
Matteo Sammartino
Francesco Santini
Alexandra Silva
Marjan Sirjani
Meng Sun
Carolyn Talcott
Leendert van der Torre
Erik de Vink

Contents

Discovering the "Glue" Connecting Activities

Exploiting Monotonicity to Learn Places Faster

Wil M. P. van der Aalst(✉)

Process and Data Science (PADS), RWTH Aachen University, Aachen, Germany
wvdaalst@pads.rwth-aachen.de

Abstract. Process discovery, one of the key areas within process mining, aims to derive behavioral models from event data. Since event logs are inherently incomplete (containing merely example behaviors) and unbalanced, this is often challenging. Different target languages can be used to capture sequential, conditional, concurrent, and iterative behaviors. In this paper, we assume that a process model is merely a set of places (like in Petri nets). Given a particular behavior, a place can be "fitting", "underfed" (tokens are missing), or "overfed" (tokens are remaining). We define a partial order on places based on their connections. Then we will show various monotonicity properties that can be exploited during process discovery. If a candidate place is underfed, then all "lighter" places are also underfed. If a candidate place is overfed, then all "heavier" places are also overfed. This allows us to prune the search space dramatically. Moreover, we can further reduce the search space by not allowing conflicting or redundant places. These more foundational insights can be used to develop fast process mining algorithms producing places with a guaranteed quality level.

Keywords: Process mining · Process discovery · Petri nets · BPM

1 Introduction

It is a pleasure to contribute to this Festschrift honoring Farhad Arbab's contributions to computer science. Farhad worked on different topics in the broader field of formal methods and software engineering. However, he is best known for his work in coordination models and languages. Often concurrency and composition played an important role in his work. The Reo coordination language is the piece de resistance of Farhad's work. Reo is a channel-based coordination model wherein complex coordinators, called connectors, are composed from simpler ones [10]. The language has been mapped to many other languages [16], including zero-safe nets (a variant for Petri nets) and constraint automata. There is also work on the synthesis of Reo circuits from scenario specifications [21]. Unfortunately, mining techniques to learn Reo models from event data are still missing.

F. de Boer et al. (Eds.): Arbab Festschrift, LNCS 10865, pp. 1–20, 2018.
https://doi.org/10.1007/978-3-319-90089-6_1

In process mining, typically representations such as Petri nets, workflow nets, causal nets, process trees, transition systems, statecharts, and BPMN models are used [3]. Rather than coordinating complex components, these models merely coordinate activities derived from event data.

The goal of Reo is to provide the "glue" between different software components. *In the same way, one could view places in a Petri net as the "glue" between transitions representing activities.* In this sense, places in a Petri net can be viewed as a simple coordination layer. The goal of this paper is to discover sets of places modeling the underlying process such that (1) this can be done quickly (handling event logs with millions of events) and (2) that places have a well-defined minimal quality level.

Event data are collected in logistics, manufacturing, finance, healthcare, customer relationship management, e-learning, e-government, and many other domains. The events found in these domains typically refer to activities executed by resources at particular times and for a particular case (i.e., process instances). Process mining techniques are able to exploit such data. Here, we focus on *process discovery*, but process mining also includes conformance checking, performance analysis, decision mining, organizational mining, predictions, recommendations, etc.

Over the last two decades, hundreds of process discovery techniques have been proposed [3]. Many of the initial techniques could not cope with infrequent behavior and made very strong assumptions about the completeness of the event log. For example, traditional region-based techniques assume that all possible behavior has been observed (i.e., the log is complete) and that all observed traces are equally important. State-based regions were introduced by Ehrenfeucht and Rozenberg in 1989 and generalized by Cortadella et al. [12,14]. Various authors used state-based regions for process discovery [7,22]. Also, language-based regions have been used for this purpose [11,25]. Over time attention shifted to approaches able to deal with noise and infrequent behavior. Early approaches include heuristic mining, fuzzy mining, and various genetic process mining approaches [15,24]. Since 2010 the speed at which new process discovery techniques are proposed is accelerating. As an example consider the family of inductive mining techniques [17–19].

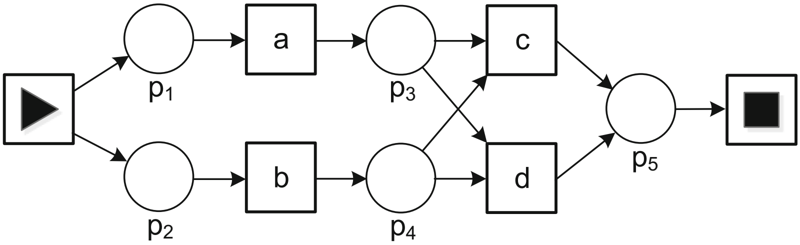


Fig. 1. Process model $P_1 = \{(\{\blacktriangleright\},\{a\}),(\{\blacktriangleright\},\{b\}),(\{a\},\{c,d\}),(\{b\},\{c,d\}),(\{c,d\},\{\blacksquare\})\}$ composed of five places discovered from event log $L_1 = [\langle\blacktriangleright,a,b,c,\blacksquare\rangle^{31}, \langle\blacktriangleright,b,a,c,\blacksquare\rangle^{27}, \langle\blacktriangleright,a,b,d,\blacksquare\rangle^{23}, \langle\blacktriangleright,b,a,d,\blacksquare\rangle^{19}]$.

This paper provides a fresh look at places in a Petri net seen from the viewpoint of process discovery. Each place can be viewed as a *constraint*, limiting the behavior of the Petri net. We use a so-called *open-world assumption*: Any behavior is possible unless explicitly forbidden by one of the places in the model. Consider the process model shown in Fig. 1 which is composed of five places. Transition ▶ models the start of the process and transition ■ marks the end. Place p_1 specifies that activity a can only happen after ▶. Moreover, at the end, the number of occurrences of a should match the number of occurrences of ▶. Since ▶ happens once, also a should also happen precisely once. Place p_3 specifies that activity c and activity d can only happen after a occurred. At the end, the number of occurrences of c and d should match the number of occurrences of a. The goal is to discover models merely composed of places from event data. Each event refers to a case (process instance), activity, and a timestamp. We can group events based on cases and sort events within a case based on the timestamps. This way each case can be presented by a trace $\langle ▶, a, b, c, ■ \rangle$, i.e., a sequence of activities. An event log is a multiset of such traces. The caption of Fig. 1 shows event log L_1 consisting of 100 cases and 500 events referring to six unique activities.

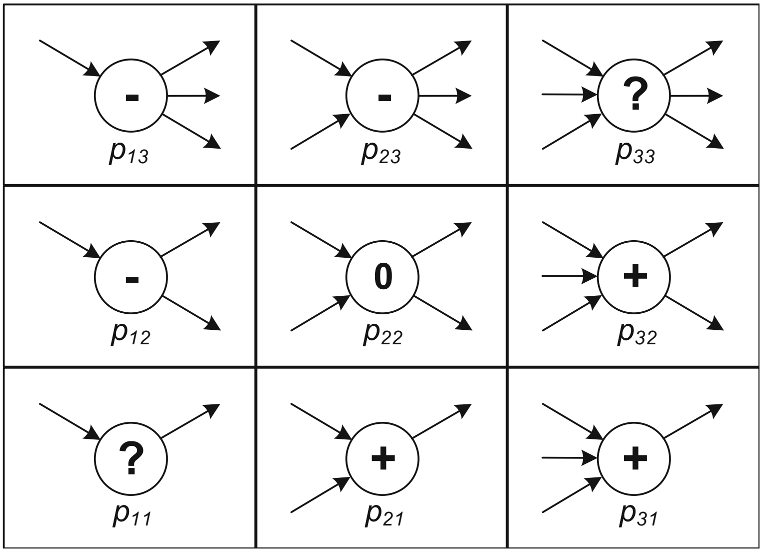


Fig. 2. During replay each of the places p_{32}, p_{21}, and p_{31} will always have at least the number of tokens in p_{22}. Similarly, places p_{13}, p_{23}, and p_{12} will always have at most the number of tokens in p_{22}. Places p_{11} and p_{33} may have more or fewer tokens.

It is far from trivial to discover places from larger event logs referring to many activities. The number of possible places grows exponentially in the number of activities and to evaluate a place one needs to traverse the whole event log. A naive algorithm would need to replay the event log for every possible candidate place. This can be very time-consuming. Moreover, places may be redundant or conflicting. Therefore, we explore relationships among (sets of) places and present several *monotonicity* results. To do this, we define new notions such as "underfed" and "overfed" places and partial orders on (sets of) places based on their input and output transitions.

Figure 2 shows the basic idea. If we replay a trace on a particular place, there could be two problems:

- At some stage, a transition needs to remove a token from the place, but the place is already empty (the place is "underfed").
- At the end of the trace, tokens remain in the place (the place is "overfed").

Note that a place can be "overfed" and "underfed" at the same time. Assume now that we have a place p_{22} with two input transitions and two output transitions (Fig. 2 only shows the corresponding arcs). If this place is perfectly fitting some trace σ (the place is not "underfed" and not "overfed"), then we know that adding an input arc and/or removing an output arc can only make the place "heavier" (indicated by the $+$ sign in Fig. 2). Moreover, removing an input arc and/or adding an output arc can only make the place "lighter" (indicated by the $-$ sign in Fig. 2). We can exploit this simple observation. If place p_{22} is already overfed, then we know that the places p_{32}, p_{21}, and p_{31} also need to be overfed. If place p_{22} is already underfed, then we know that the places p_{13}, p_{23}, and p_{12} also need to be underfed. These monotonicity properties allow us to prune the search space of candidate places. In fact, *the monotonicity results can be exploited by discovery algorithms to speed-up discovery while still producing all places that meet predefined quality criteria.* This paper focuses on the formal foundation of such approaches without providing a specific process discovery technique. Nevertheless, it is quite straightforward to see how the results can be used to speed-up process discovery.

The remainder is organized as follows. Section 2 provides the formal setting by defining behaviors, event logs, process models, and their semantics. In Sect. 3 we relate places using partial orders and prove the first monotonicity results. This is then lifted to quality scores for places (Sects. 4 and 5). We briefly discuss how these monotonicity results can be used for process discovery (Sect. 6). To further prune the set of candidate places we define redundancy and conflict (Sect. 7). Section 8 discusses implications for conformance checking. Section 9 concludes the paper.

2 Behaviors, Event Logs, and Models

To be able to discuss the monotonicity results that can be exploited by process discovery approaches we define key notions such as behaviors, event logs, and places. We also define the semantics of process models based on places using an open-world assumption.

2.1 Behaviors

First, we introduce some basic mathematical notations.

$\mathcal{P}(X) = \{Y \mid Y \subseteq X\}$ is the powerset of set X. $\mathcal{B}(X) = X \to \mathbb{N}$ is the set of all multisets over some set X. For any $B \in \mathcal{B}(X)$: $B(x)$ denotes the number of times element $x \in X$ appears in B. $B_1 = [\,]$, $B_2 = [a, a, b]$, and $B_3 = [a^3, b^2, c]$ are multisets over $X = \{a, b, c\}$. B_1 is the empty multiset, B_2 has three elements, and B_3 has six elements. Note that the ordering of elements is irrelevant. Union ($B_1 \cup B_2$), intersection ($B_1 \cap B_2$), and difference ($B_1 \setminus B_2$) are defined as usual. All operators for sets are generalized to multisets, e.g., $\sum_{x \in [a,b,b,a,c]} x = 2a + 2b + c$.

$\sigma = \langle x_1, x_2, \ldots, x_n \rangle \in X^*$ denotes a sequence over X. $\sigma(i) = a_i$ denotes the i-th element of the sequence. $|\sigma| = n$ is the length of σ and $dom(\sigma) = \{1, \ldots, |\sigma|\}$ is the domain of σ. $\langle\,\rangle$ is the empty sequence, i.e., $|\langle\,\rangle| = 0$ and $dom(\langle\,\rangle) = \emptyset$. $\sigma_1 \cdot \sigma_2$ is the concatenation of two sequences.

Based on the preliminaries we can define the notion of *behavior*. $\langle \blacktriangleright, a, b, c, \blacksquare \rangle$ is an example behavior, i.e., a sequence of activities starting with $\blacktriangleright$ and ending with $\blacksquare$.

Definition 1 (Activities and Behaviors). *$\mathbb{A}$ is the universe of activities (actions, tasks, operations, transaction types, etc.). There are two special activities: $\{\blacktriangleright, \blacksquare\} \subseteq \mathbb{A}$. $\blacktriangleright$ is the unique start activity and $\blacksquare$ is the unique end activity. A behavior $\sigma = \langle a_1, a_2, \ldots, a_n \rangle \in \mathbb{A}^*$ is a sequence of activity names such that $n \geq 2$, $a_1 = \blacktriangleright$, $a_n = \blacksquare$, and for all $1 < i < n$: $a_i \in \mathbb{A} \setminus \{\blacktriangleright, \blacksquare\}$. $\mathbb{B}$ is the set of all possible behaviors.*

In this paper, $\mathbb{A} = \{\blacktriangleright, \blacksquare, a, b, c, d, \ldots\}$ and $\mathbb{B} = \{\langle \blacktriangleright, \blacksquare \rangle, \langle \blacktriangleright, a, \blacksquare \rangle, \langle \blacktriangleright, b, \blacksquare \rangle, \ldots \langle \blacktriangleright, a, b, c, c, a, d, \blacksquare \rangle, \ldots\}$.

2.2 Event Logs

An *event log* can be defined as a multiset of behaviors. Elements of such a multiset are called *traces* and refer to *cases* (i.e., process instance).

Definition 2 (Event Log). *An event log L is a multiset of behaviors, i.e., $L \in \mathcal{B}(\mathbb{B})$. $\sigma \in L$ is called a trace.*

$L_1 = [\langle \blacktriangleright, a, b, c, \blacksquare \rangle^{31}, \langle \blacktriangleright, b, a, c, \blacksquare \rangle^{27}, \langle \blacktriangleright, a, b, d, \blacksquare \rangle^{23}, \langle \blacktriangleright, b, a, d, \blacksquare \rangle^{19}]$ is an example of an event log with 100 traces. For example, 31 cases exhibit the behavior $\langle \blacktriangleright, a, b, c, \blacksquare \rangle$. Typically, an event log has more information. For example, events may have a timestamp, refer to resources, locations, customers, costs, etc. Since we focus on the discovery of the "control-flow backbone" of a process, we can abstract from these optional attributes.

2.3 Using Places to Constrain Behavior

In the context of process mining, a wide variety of modeling languages are used ranging from Petri nets, workflow nets, causal nets, process trees and transition systems to statecharts and BPMN models. In this paper, we use a very "lean" modeling language based on places and an open-world assumption. First, we define $\mathbb{P}$ and $\mathbb{P}^!$ as the set of all possible (through) places.

Definition 3 (Places). *$\mathbb{P} = \mathcal{P}(\mathbb{A}) \times \mathcal{P}(\mathbb{A})$ is the set of all possible places. For any $p = (I, O) \in \mathbb{P}$, $\bullet p = I$ is the set of input activities and $p\bullet = O$ is the set of output activities. $\mathbb{P}^! = (\mathcal{P}(\mathbb{A}) \setminus \{\emptyset\}) \times (\mathcal{P}(\mathbb{A}) \setminus \{\emptyset\})$ is the set of through places, i.e., places having non-empty sets of input and output activities.*

Note that places do not have names, they are fully identified by the input and output activities. Therefore, for any p_1 and p_2, if $\bullet p_1 = \bullet p_2$ and $p_1\bullet = p_2\bullet$, then $p_1 = p_2$. A process model is simply a set of places.

Definition 4 (Process Model). *A set of places $P \subseteq \mathbb{P}$ defines a process model.*

Figure 1 shows the process model $P_1 = \{(\{\blacktriangleright\}, \{a\}), (\{\blacktriangleright\}, \{b\}), (\{a\}, \{c, d\}), (\{b\}, \{c, d\}), (\{c, d\}, \{\blacksquare\})\}$.

Unlike conventional Petri nets, there is (1) no initial marking and (2) not an explicit set of transitions. We do not need an initial marking because behaviors start with the unique start activity $\blacktriangleright$. $T = \bigcup_{p \in P} \bullet p \cup p\bullet$ is the implicit set of transitions (corresponding to the inputs and output of places). However, because we use an open-world assumption, we allow for activities not mentioned in the process model. Places only constrain the behavior of the activities explicitly mentioned. For example, $\langle \blacktriangleright, a, d, d, d, b, e, e, c, \blacksquare \rangle$ is a behavior allowed by process model P_1 (simply ignore activities d and e). In the remainder, we will use the terms transition and activity interchangeably. Whereas the term transition is common in the context of Petri nets, event logs refer to occurrences of activities rather than model elements.

2.4 Behavior Defined by Places

To formalize the semantics of a process model $P \subseteq \mathbb{P}$ we define "underfed", "overfed", and "fitting" places. Given a behavior $\sigma \in \mathbb{B}$, place p is underfed if during the replay of the trace place p "goes negative", i.e., a token needs to be consumed while it has not been produced (yet). Place p is overfed if at the end of replaying a trace, tokens remain in p. Place p is fitting if it is not underfed and not overfed, i.e., place p does not "go negative" and at the end no tokens remain.

Definition 5 (Underfed, Overfed, and Fitting Places). *Let $p \in \mathbb{P}$ be a place and $\sigma = \langle a_1, a_2, \ldots, a_n \rangle \in \mathbb{B}$ a behavior.*

- *$\triangledown_\sigma(p)$ if and only if $|\{1 \leq i < k \mid a_i \in \bullet p\}| < |\{1 \leq i \leq k \mid a_i \in p\bullet\}|$ for some $k \in \{1, 2, \ldots, n\}$ (place p is "underfed"),*
- *$\vartriangle_\sigma(p)$ if and only if $|\{1 \leq i \leq n \mid a_i \in \bullet p\}| > |\{1 \leq i \leq n \mid a_i \in p\bullet\}|$ (place p is "overfed"), and*

- $\Box_\sigma(p)$ *if and only if* $\not\bigtriangledown_\sigma(p)$ *and* $\not\bigtriangleup_\sigma(p)$ *(place* p *is “fitting”, i.e., not “underfed” and not “overfed”).*

Consider trace $\sigma = \langle \blacktriangleright, a, b, c, d, \blacksquare \rangle$ and the five places in Fig. 1. Places p_1 and p_2 are fitting, p_3 and p_4 are underfed (because d occurs when these places empty), and p_5 is overfed (because two tokens are produced and only one is consumed).

As mentioned before, activities not in $\bullet p \cup p \bullet$ have no effect on the evaluation. If $\sigma = \langle \blacktriangleright, a, e, e, b, f, c, f, d, e, \blacksquare \rangle$, then p_1 and p_2 are still fitting, p_3 and p_4 are still underfed, and p_5 is still overfed.

A place can be both underfed and overfed. Consider trace $\sigma = \langle \blacktriangleright, c, a, a, b, b, \blacksquare \rangle$ and the five places in Fig. 1. Places p_1 and p_2 are underfed, p_3 and p_4 are both underfed and overfed (tokens are missing when c occurs and at the end tokens remain), and p_5 is fitting.

The above notions can be generalized to sets of places. Therefore, it is possible to say that a model $P \subseteq \mathbb{P}$ is fitting ($\Box_\sigma(P)$), underfed ($\bigtriangledown_\sigma(P)$), or overfed ($\bigtriangleup_\sigma(P)$).

Definition 6. *Let* $P \subseteq \mathbb{P}$ *be a set of places and* $\sigma \in \mathbb{B}$ *a behavior.*

- $\bigtriangledown_\sigma(P)$ *if and only if there exists a place* $p \in P$ *such that* $\bigtriangledown_\sigma(p)$*,*
- $\bigtriangleup_\sigma(P)$ *if and only if there exists a place* $p \in P$ *such that* $\bigtriangleup_\sigma(p)$*, and*
- $\Box_\sigma(P)$ *if and only if* $\Box_\sigma(p)$ *for all* $p \in P$*.*

Using $\bigtriangledown_\sigma(P)$, $\bigtriangleup_\sigma(P)$, and $\Box_\sigma(P)$ we can compute all fitting, underfed, and overfed behaviors. The set of fitting behaviors $\mathit{fit}(P)$ precisely defines the semantics of a process model $P \subseteq \mathbb{P}$.

Definition 7 (Model Behavior). *Let* $P \subseteq \mathbb{P}$ *be a set of places.*

- $\mathit{fit}(P) = \{\sigma \in \mathbb{B} \mid \Box_\sigma(P)\}$ *is the set of fitting behaviors,*
- $\mathit{neg}(P) = \{\sigma \in \mathbb{B} \mid \bigtriangledown_\sigma(P)\}$ *is the set of underfed behaviors, and*
- $\mathit{pos}(P) = \{\sigma \in \mathbb{B} \mid \bigtriangleup_\sigma(P)\}$ *is the set of overfed behaviors.*

We use the following shorthands: $\mathit{fit}(p) = \mathit{fit}(\{p\})$, $\mathit{neg}(p) = \mathit{neg}(\{p\})$, $\mathit{pos}(p) = \mathit{pos}(\{p\})$ for any place p. Note that $\mathit{fit}(P) = \mathbb{B} \setminus (\mathit{neg}(P) \cup \mathit{pos}(P))$.

Figure 3 shows another example illustrating the declarative nature of places. $P_2 = \{(\{\blacktriangleright, a\}, \{a, b\}), (\{a\}, \{c\}), (\{b, c\}, \{c, \blacksquare\})\}$ has three places allowing for any behavior satisfying the following constraints: (1) b occurs precisely once, (2) a occurs any number of times, but only before b, (3) c occurs any number of times, but only after b, and (4) a and c occur the same number of times. Note that the model in Fig. 3 only constrains activities a, b, and c and therefore also allows for behaviors like $\langle \blacktriangleright, d, b, e, f, \blacksquare \rangle$ and $\langle \blacktriangleright, a, d, a, d, b, e, c, c, e, \blacksquare \rangle$.

2.5 Mapping to Petri Nets

Traditional *Petri nets* are described by tuple $N = (S, T, F)$ where S is the set of places, T is the set of transitions, and $F \subseteq (S \times T) \cup (T \times S)$ the set

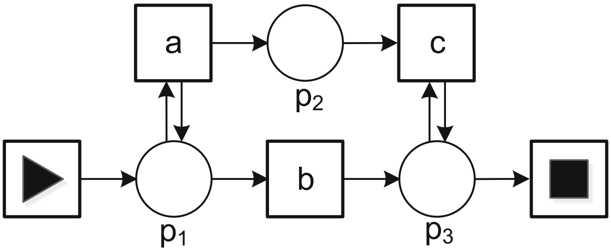


Fig. 3. Process model $P_2 = \{(\{\blacktriangleright, a\}, \{a, b\}), (\{a\}, \{c\}), (\{b, c\}, \{c, \blacksquare\})\}$ composed of three places discovered from event log $L_1 = [\langle \blacktriangleright, b, \blacksquare \rangle^{49}, \langle \blacktriangleright, a, b, c, \blacksquare \rangle^{31}, \langle \blacktriangleright, a, a, b, c, c, \blacksquare \rangle^{12}, \langle \blacktriangleright, a, a, a, b, c, c, c, \blacksquare \rangle^{5}, \langle \blacktriangleright, a, a, a, a, b, c, c, c, c, \blacksquare \rangle^{1}, \langle \blacktriangleright, a, a, a, a, a, b, c, c, c, c, c, \blacksquare \rangle^{2}]$.

of arcs [13]. A *system net* $SN = (S, T, F, M_{init}, M_{final})$ has an initial and a final marking [1]. The *behavior* of a system net corresponds to the set of traces starting in the initial marking M_{init} and ending in the final marking M_{final} [1]. The models used in this paper can be converted to a system net using the following conversion. Given a set of places $P \subseteq \mathbb{P}$ (in the sense of Definition 4), we construct the system net $SN = (S, T, F, M_{init}, M_{final})$ with: $S = P \cup \{i, q, f\}$, $T = \mathbb{A}$, $F = \{(i, \blacktriangleright), (\blacksquare, f)\} \cup \{(t, p) \in T \times P \mid t \in \bullet p\} \cup \{(p, t) \in P \times T \mid t \in p\bullet\} \cup \{(t, q) \mid t \in T \setminus \{\blacksquare\}\} \cup \{(q, t) \mid t \in T \setminus \{\blacktriangleright\}\}$, $M_{init} = [i]$, and $M_{final} = [f]$. The set of traces starting in M_{init} and ending in M_{final} is precisely the set $\mathit{fit}(P)$ (see Definition 7). Moreover, note that SN is a so-called *workflow net* [5]. The workflow net does not need to be sound, but we only consider firing sequences starting in marking $[i]$ and ending in marking $[f]$.

It is also possible to translate any system net (including workflow nets) with initial and final markings into an equivalent model $P \subseteq \mathbb{P}$.

We use the simple representation using merely places and no initial and final markings to be able to succinctly express a range of properties and monotonicity results without considering markings.

3 Relating Places and Monotonicity

The ultimate goal is to discover places from event logs. However, the goal of this paper is not to propose a concrete discovery approach. Instead, we reason about properties of (sets of) places that can be exploited by discovery techniques.

Definition 8 (Place Notations). *Let $p_1 = (I_1, O_1) \in \mathbb{P}$ and $p_2 = (I_2, O_2) \in \mathbb{P}$ be two places. These places can be combined to form new places:*

- $p_1 \sqcap p_2 = (I_1 \cap I_2, O_1 \cap O_2) \in \mathbb{P}$,
- $p_1 \sqcup p_2 = (I_1 \cup I_2, O_1 \cup O_2) \in \mathbb{P}$,
- $p_1 \otimes p_2 = ((I_1 \cup I_2) \setminus (I_1 \cap I_2), (O_1 \cup O_2) \setminus (O_1 \cap O_2)) \in \mathbb{P}$.

Places p_1 and p_2 can be related in different ways:

- $p_1 = p_2$ if and only if $I_1 = I_2$ and $O_1 = O_2$ (equality),
- $p_1 \parallel p_2$ if and only if $p_1 \sqcap p_2 = (\emptyset, \emptyset)$ (non-overlapping),
- $p_1 \sqsubset p_2$ if and only if $I_1 \subseteq I_2$, $O_1 \subseteq O_2$, and $p_1 \neq p_2$ (proper subset), and
- $p_1 \div p_2$ if and only if $p_1 \neq p_2$, $p_1 \not\parallel p_2$, $p_1 \not\sqsubset p_2$ and $p_1 \not\sqsupset p_2$ (incomparable).

We would like to avoid discovering places that are a combination of places already in the model. Consider for example adding place $p_r = (\{\blacktriangleright, c, d\}, \{a, \blacksquare\})$ to the five places in Fig. 1. This place would be *redundant*, because $p_r = p_1 \sqcup p_5$. Indeed, adding p_r would not change the set of fitting behaviors and only complicate the model. A set of places is *non-redundant* if none of its places can be derived from the rest.

Definition 9 (Redundant). *Place $p \in \mathbb{P}$ is redundant with respect to a set of places $P \subseteq \mathbb{P}$ (notation $P \Rightarrow p$) if there is a non-empty subset $P' = \{p_1, p_2, \ldots, p_n\} \subseteq P$ such that $p_i \parallel p_j$ for any $1 \leq i < j \leq n$ and $p = (p_1 \sqcup p_2 \sqcup \ldots \sqcup p_n)$.*

For two sets of places $P_1 \subseteq \mathbb{P}$ and $P_2 \subseteq \mathbb{P}$: $P_1 \Rightarrow P_2$ if and only if $\forall_{p_2 \in P_2}\ P_1 \Rightarrow p_2$ (i.e., P_2 is “implied” by P_1).

A set of places $P \subseteq \mathbb{P}$ is non-redundant if and only if it is impossible to split P in two disjoint non-empty subsets P_1 and P_2 such that $P_1 \Rightarrow P_2$.

Adding input transitions to a place can only lead to more tokens in the place. Therefore, a place that is overfed by many traces in the event log will also be overfed by these traces after adding the input transitions. Adding output transitions to a place can only lead to fewer tokens in the place. Therefore, a place that is underfed by many traces in the event log will also be underfed by these traces after adding the output transitions. This information can be used to prune the search space of discovery algorithms. Therefore, we define a partial order on places and use this to prove monotonicity results that can be exploited during process discovery.

Definition 10 (Weighing Places). *Let $p_1 = (I_1, O_1) \in \mathbb{P}$ and $p_2 = (I_2, O_2) \in \mathbb{P}$ be two places.*

- *$p_1 \preceq p_2$ if and only if $I_1 \subseteq I_2$ and $O_2 \subseteq O_1$ (i.e., p_1 is at least as “light” as p_2) and*
- *$p_1 \succeq p_2$ if and only if $I_2 \subseteq I_1$ and $O_1 \subseteq O_2$ (i.e., p_1 is at least as “heavy” as p_2).*

Note that $p_1 \preceq p_2$ if and only if $p_2 \succeq p_1$. It is easy to see that $\preceq$ defines a partial order. The relation is reflexive ($p \preceq p$), antisymmetric ($p_1 \preceq p_2$ and $p_2 \preceq p_1$ implies $p_1 = p_2$), and transitive ($p_1 \preceq p_2$ and $p_2 \preceq p_3$ implies $p_1 \preceq p_3$).

Definition 11 (Weighing Sets of Places). *Let $P_1 \subseteq \mathbb{P}$ and $P_2 \subseteq \mathbb{P}$ be two sets of places.*

- *$P_1 \preceq P_2$ if and only if $\forall_{p_1 \in P_1}\ \exists_{p_2 \in P_2}\ p_1 \preceq p_2$ (i.e., P_1 is at least as “light” as P_2) and*
- *$P_1 \succeq P_2$ if and only if $\forall_{p_1 \in P_1}\ \exists_{p_2 \in P_2}\ p_1 \succeq p_2$ (i.e., P_1 is at least as “heavy” as P_2).*

Note that $P_1 \preceq P_2$ is *not* equivalent to $P_2 \succeq P_1$. Let $P_1 = \{(\{a\},\{b,c\})\}$ and $P_2 = \{(\{a\},\{b\}),(\{a\},\{d\})\}$. $P_1 \preceq P_2$ because $(\{a\},\{b,c\}) \preceq (\{a\},\{b\})$. However, $P_2 \not\succeq P_1$, because $(\{a\},\{d\}) \not\succeq (\{a\},\{b,c\})$. Both $\preceq$ and $\succeq$ (for sets of places) are reflexive and transitive, but not antisymmetric. Consider $P_3 = \{(\{a,c\},\{b\}),(\{a\},\{b\}),(\{a\},\{b,d\})\}$ and $P_4 = \{(\{a,c\},\{b\}),(\{a\},\{b,d\})\}$. $P_3 \preceq P_4$ and $P_4 \preceq P_3$, but $P_3 \neq P_4$. Also, $P_3 \succeq P_4$ and $P_4 \succeq P_3$, but $P_3 \neq P_4$. Hence, $\preceq$ and $\succeq$ are not antisymmetric.

The above notations and insights allow us to provide very general monotonicity results.

Theorem 1 (Monotonicity Results). *Let $P_1 \subseteq \mathbb{P}$ and $P_2 \subseteq \mathbb{P}$ be two sets of places.*

- *$P_1 \preceq P_2$ implies $pos(P_1) \subseteq pos(P_2)$,*
- *$P_1 \succeq P_2$ implies $neg(P_1) \subseteq neg(P_2)$,*
- *$P_1 \Rightarrow P_2$ implies $fit(P_1) \subseteq fit(P_2)$.*

Proof. If $p_1 \preceq p_2$, then while replaying a trace σ, p_1 cannot have more tokens than p_2, but p_2 can have more tokens than p_1 if the right transitions are activated. Therefore, if $\triangle_\sigma(p_1)$, then $\triangle_\sigma(p_2)$, and if $\triangledown_\sigma(p_2)$, then $\triangledown_\sigma(p_1)$.

Using this insight we prove that $P_1 \preceq P_2$ implies $pos(P_1) \subseteq pos(P_2)$. Assume $P_1 \preceq P_2$, i.e., $\forall_{p_1 \in P_1} \exists_{p_2 \in P_2}\ p_1 \preceq p_2$. We need to prove that for any $\sigma \in \mathbb{B}$: $\exists_{p_1 \in P_1} \triangle_\sigma(p_1)$ implies $\exists_{p_2 \in P_2} \triangle_\sigma(p_2)$. Take a p_1 such that $\triangle_\sigma(p_1)$. There exists a $p_2 \in P_2$ such that $p_1 \preceq p_2$. Place p_2 can only have more tokens than p_1 (and not fewer). Hence, $\triangle_\sigma(p_2)$.

Similarly, we can prove that $P_1 \succeq P_2$ implies $neg(P_1) \subseteq neg(P_2)$.

$P_1 \Rightarrow P_2$ means that all places in P_2 correspond to combinations of places in P_1. Therefore, adding these places does not change the behavior, i.e., $fit(P_1) = fit(P_1 \cup P_2)$. Removing places from $P_1 \cup P_2$ can only result in more fitting traces. Hence, $fit(P_1) = fit(P_1 \cup P_2) \subseteq fit(P_2)$. □

4 Scoring Places

Theorem 1 can be exploited by process discovery algorithms. If a place is underfed (overfed), it does not make sense to consider lighter (heavier) places. Therefore, monotonicity results allow for quickly pruning the search space. To illustrate this, we define concrete quality characteristics for individual places.

One could simply count the fraction of cases having problems. However, some activities may occur infrequently. Places that are only connected to these low-frequency activities have many fitting traces by definition (the place is rarely involved in the execution of a case). In other words, "random places" only connected to low-frequency activities will always have a good score. Therefore, we also consider the "relative" scores of places by only considering traces that actually consume/produce tokens from/for the place under investigation. A trace "activates" place p if it contains an activity in $\bullet p \cup p\bullet$.

Definition 12 (Activation). *Let $p \in \mathbb{P}$ be a place.*

- $act_\sigma(p) = \exists_{a \in \sigma}\ a \in (\bullet p \cup p\bullet)$ *denotes whether the place has been activated in a trace* $\sigma \in \mathbb{B}$, *i.e., a token was consumed or produced for* p *in* σ.
- $act_L(p) = \exists_{\sigma \in L}\ act_\sigma(p)$ *denotes whether place* p *has been activated in an event log* $L \in \mathcal{B}(\mathbb{B})$.

Definition 13 (Place Scores). *Let* $L \in \mathcal{B}(\mathbb{B})$ *be an event log and* $\tau \in [0,1]$ *a threshold. For any place* $p \in \mathbb{P}$ *such that* $act_L(p)$, *we define the following scores:*

- $\#^{\triangledown}_{freq,L}(p) = \frac{|[\sigma \in L | \triangledown_\sigma(p)]|}{|L|}$ *is the fraction of traces for which* p *is underfed,*
- $\#^{\triangle}_{freq,L}(p) = \frac{|[\sigma \in L | \triangle_\sigma(p)]|}{|L|}$ *is the fraction of traces for which* p *is overfed,*
- $\#^{\square}_{freq,L}(p) = \frac{|[\sigma \in L | \square_\sigma(p)]|}{|L|}$ *is the fraction of fitting traces,*
- $\#^{\triangledown}_{rel,L}(p) = \frac{|[\sigma \in L | \triangledown_\sigma(p)\ \wedge\ act_\sigma(p)]|}{|[\sigma \in L | act_\sigma(p)]|} = \frac{|[\sigma \in L | \triangledown_\sigma(p)]|}{|[\sigma \in L | act_\sigma(p)]|}$ *is the fraction of activating traces for which* p *is underfed,*
- $\#^{\triangle}_{rel,L}(p) = \frac{|[\sigma \in L | \triangle_\sigma(p)\ \wedge\ act_\sigma(p)]|}{|[\sigma \in L | act_\sigma(p)]|} = \frac{|[\sigma \in L | \triangle_\sigma(p)]|}{|[\sigma \in L | act_\sigma(p)]|}$ *is the fraction of activating traces for which* p *is overfed,*
- $\#^{\square}_{rel,L}(p) = \frac{|[\sigma \in L | \square_\sigma(p)\ \wedge\ act_\sigma(p)]|}{|[\sigma \in L | act_\sigma(p)]|}$ *is the fraction of activated traces that are also fitting,*
- $\triangledown^{\tau}_{freq,L}(p)$ *if and only if* $\#^{\triangledown}_{freq,L}(p) > \tau$,
- $\triangle^{\tau}_{freq,L}(p)$ *if and only if* $\#^{\triangle}_{freq,L}(p) > \tau$,
- $\square^{\tau}_{freq,L}(p)$ *if and only if* $\#^{\square}_{freq,L}(p) \geq \tau$,
- $\triangledown^{\tau}_{rel,L}(p)$ *if and only if* $\#^{\triangledown}_{rel,L}(p) > \tau$,
- $\triangle^{\tau}_{rel,L}(p)$ *if and only if* $\#^{\triangle}_{rel,L}(p) > \tau$,
- $\square^{\tau}_{rel,L}(p)$ *if and only if* $\#^{\square}_{rel,L}(p) \geq \tau$.

For a discovered place we would like $\#^{\square}_{freq,L}(p)$ and $\#^{\square}_{rel,L}(p)$ to be as high as possible. A place p is perfectly fitting log L if $\#^{\square}_{freq,L}(p) = \#^{\square}_{rel,L}(p) = 1$. If $\square^{0.95}_{rel,L}(p)$, then at least 95% of all traces that activate place p are fitting. If a discovery algorithm only adds places for which $\square^{0.95}_{rel,L}(p)$, then all places have a minimal quality level interpretable by end users (unlike existing approaches that do not provide such a guarantee or "only in the limit").

5 Monotonicity of Place Scores

Since we are interested in places of a certain quality level, e.g., places for which $\square^{0.95}_{rel,L}(p)$ holds, we would like to avoid spending time on the evaluation of places that do not meet the desired quality level. We would like to use Theorem 1 to quickly prune the set of candidate places. We start by listing several observations that directly follow from earlier definitions.

Lemma 1. *Let* $L \in \mathcal{B}(\mathbb{B})$ *be an event log,* $\sigma \in \mathbb{B}$ *a trace, and* $\tau \in [0,1]$ *a threshold. For any place* $p \in \mathbb{P}$ *such that* $act_L(p)$:

- $\triangledown_\sigma(p)$ *implies* $act_\sigma(p)$,
- $\triangle_\sigma(p)$ *implies* $act_\sigma(p)$,

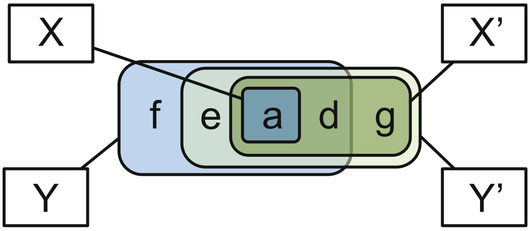


Fig. 4. Visualization of the sets used in Lemma 2. In Theorem 2: $X = [\sigma \in L \mid \triangle_\sigma(p_1)]$, $Y = [\sigma \in L \mid \mathit{act}_\sigma(p_1)]$, $X' = [\sigma \in L \mid \triangle_\sigma(p_2)]$, and $Y' = [\sigma \in L \mid \mathit{act}_\sigma(p_2)]$.

- $\#^{\square}_{\mathit{freq},L}(p) \leq 1 - \#^{\triangledown}_{\mathit{freq},L}(p)$,
- $\#^{\square}_{\mathit{freq},L}(p) \leq 1 - \#^{\triangle}_{\mathit{freq},L}(p)$,
- $\#^{\square}_{\mathit{freq},L}(p) \geq 1 - (\#^{\triangledown}_{\mathit{freq},L}(p) + \#^{\triangle}_{\mathit{freq},L}(p))$,
- $\#^{\square}_{\mathit{rel},L}(p) \leq 1 - \#^{\triangledown}_{\mathit{rel},L}(p)$,
- $\#^{\square}_{\mathit{rel},L}(p) \leq 1 - \#^{\triangle}_{\mathit{rel},L}(p)$,
- $\#^{\square}_{\mathit{rel},L}(p) \geq 1 - (\#^{\triangledown}_{\mathit{rel},L}(p) + \#^{\triangle}_{\mathit{rel},L}(p))$,
- $\square^{\tau}_{\mathit{freq},L}(p)$ *implies* $\not\triangle^{1-\tau}_{\mathit{freq},L}(p)$,
- $\square^{\tau}_{\mathit{freq},L}(p)$ *implies* $\not\triangledown^{1-\tau}_{\mathit{freq},L}(p)$,
- $\square^{\tau}_{\mathit{rel},L}(p)$ *implies* $\not\triangle^{1-\tau}_{\mathit{rel},L}(p)$,
- $\square^{\tau}_{\mathit{rel},L}(p)$ *implies* $\not\triangledown^{1-\tau}_{\mathit{rel},L}(p)$,
- $\not\triangledown^{\tau}_{\mathit{freq},L}(p)$ *and* $\not\triangle^{\tau}_{\mathit{freq},L}(p)$ *implies* $\square^{1-2\times\tau}_{\mathit{freq},L}(p)$, *and*
- $\not\triangledown^{\tau}_{\mathit{rel},L}(p)$ *and* $\not\triangle^{\tau}_{\mathit{rel},L}(p)$ *implies* $\square^{1-2\times\tau}_{\mathit{rel},L}(p)$.

Proof. Note that $\triangledown_\sigma(p)$ implies $\not\triangle_\sigma(p)$, $\triangle_\sigma(p)$ implies $\not\triangledown_\sigma(p)$, and $\square_\sigma(p)$ implies $\not\triangledown_\sigma(p)$ and $\not\triangle_\sigma(p)$. These insights can be used to verify the properties listed.

Consider for example the last property. Assume $\not\triangledown^{\tau}_{\mathit{rel},L}(p)$ and $\not\triangle^{\tau}_{\mathit{rel},L}(p)$. Since $\#^{\triangledown}_{\mathit{rel},L}(p) \leq \tau$ and $\#^{\triangle}_{\mathit{rel},L}(p) \leq \tau$, we know $\#^{\square}_{\mathit{rel},L}(p) \geq 1 - (\#^{\triangledown}_{\mathit{rel},L}(p) + \#^{\triangle}_{\mathit{rel},L}(p)) \geq 1 - (\tau + \tau)$. Hence, $\square^{1-2\times\tau}_{\mathit{rel},L}(p)$. □

Before we show monotonicity with respect to the place scores, we first prove the following lemma.

Lemma 2. *Let X, Y, X', and Y' be sets such that $Y \neq \emptyset$, $Y' \neq \emptyset$, $X \subseteq Y$, $X' \subseteq Y'$, $X \subseteq X'$, and $Y' \setminus Y \subseteq X'$.*

$$\frac{|X|}{|Y|} \leq \frac{|X'|}{|Y'|}$$

Proof. Let $A = Y \cup Y'$, $a = |X \cap X'|$, $b = |X \cap (Y' \setminus X')|$, $c = |X \cap (A \setminus Y')|$, $d = |(Y \setminus X) \cap X'|$, $e = |(Y \setminus X) \cap (Y' \setminus X')|$, $f = |(Y \setminus X) \cap (A \setminus Y')|$, $g = |(A \setminus Y) \cap X'|$, $h = |(A \setminus Y) \cap (Y' \setminus X')|$, and $i = |(A \setminus Y) \cap (A \setminus Y')|$ (see Fig. 4).

Because $X \subseteq X'$, $b = c = 0$. Because $Y' \setminus Y \subseteq X'$, $h = 0$. Also $i = 0$. Hence, $|X| = a$, $|Y| = a + d + e + f$, $|X'| = a + d + g$, $|Y'| = a + d + e + g$.

$$\frac{|X|}{|Y|} = \frac{a}{a+d+e+f} \leq \frac{a}{a+d+e} \leq \frac{a+g}{a+d+e+g} \leq \frac{a+d+g}{a+d+e+g} = \frac{|X'|}{|Y'|}$$

Note that $\frac{a}{a+d+e} \leq \frac{a+g}{a+d+e+g}$ because $a(a+d+e+g) = a^2 + ad + ae + ag \leq a^2 + ad + ae + ag + dg + eg = (a+g)(a+d+e)$. □

Recall that our goal is to dismiss candidate places that are overfed or underfed as soon as possible. Given a threshold τ we would like to avoid checking the quality of places for which $\triangle^{\tau}_{freq,L}(p)$, $\triangledown^{\tau}_{freq,L}(p)$, $\triangle^{\tau}_{rel,L}(p)$, or $\triangledown^{\tau}_{rel,L}(p)$. Using the partial order on places, we can exploit the following monotonicity result.

Theorem 2 (Monotonicity). *Let $L \in \mathcal{B}(\mathbb{B})$ be a non-empty event log and let $\tau \in [0,1]$ be some threshold. For any two places $p_1, p_2 \in \mathbb{P}$ such that $p_1 \preceq p_2$:*

- *If $\triangle^{\tau}_{freq,L}(p_1)$, then $\triangle^{\tau}_{freq,L}(p_2)$.*
- *If $\triangledown^{\tau}_{freq,L}(p_2)$, then $\triangledown^{\tau}_{freq,L}(p_1)$.*

Moreover, if $act_L(p_1)$ and $act_L(p_2)$, then these findings also apply to the relative notion.

- *If $\triangle^{\tau}_{rel,L}(p_1)$, then $\triangle^{\tau}_{rel,L}(p_2)$.*
- *If $\triangledown^{\tau}_{rel,L}(p_2)$, then $\triangledown^{\tau}_{rel,L}(p_1)$.*

Proof. Assume $\triangle^{\tau}_{freq,L}(p_1)$. Hence, $\#^{\triangle}_{freq,L}(p_1) = \frac{|\,[\sigma \in L | \triangle_{\sigma}(p_1)]\,|}{|L|} \geq \tau$. Since $\triangle_{\sigma}(p_1)$ implies $\triangle_{\sigma}(p_2)$ for any $\sigma \in L$, $\frac{|\,[\sigma \in L | \triangle_{\sigma}(p_2)]\,|}{|L|} \geq \frac{|\,[\sigma \in L | \triangle_{\sigma}(p_1)]\,|}{|L|}$. Hence, $\triangle^{\tau}_{freq,L}(p_2)$. Similarly, we can show that $\triangledown^{\tau}_{freq,L}(p_2)$ implies $\triangledown^{\tau}_{freq,L}(p_1)$.

Assume $\triangle^{\tau}_{rel,L}(p_1)$. Hence, $\#^{\triangle}_{rel,L}(p_1) = \frac{|\,[\sigma \in L | \triangle_{\sigma}(p_1)]\,|}{|\,[\sigma \in L | act_{\sigma}(p_1)]\,|} \geq \tau$. Using Lemma 2 we show that $\#^{\triangle}_{rel,L}(p_2) = \frac{|\,[\sigma \in L | \triangle_{\sigma}(p_2)]\,|}{|\,[\sigma \in L | act_{\sigma}(p_2)]\,|} \geq \tau$. Let $X = [\sigma \in L \mid \triangle_{\sigma}(p_1)]$, $Y = [\sigma \in L \mid act_{\sigma}(p_1)]$, $X' = [\sigma \in L \mid \triangle_{\sigma}(p_2)]$, and $Y' = [\sigma \in L \mid act_{\sigma}(p_2)]$. X, X', Y, and Y' are multisets. However, for simplicity assume that each case is uniquely identifiable so that we can treat these as sets. One can use case identifiers to identify traces even when they are identical. To apply Lemma 2, we first check the conditions: $Y \neq \emptyset$ because $act_L(p_1)$, $Y' \neq \emptyset$ because $act_L(p_2)$, $X \subseteq Y$ because $\triangle_{\sigma}(p_1)$ implies $act_{\sigma}(p_1)$, $X' \subseteq Y'$ because $\triangle_{\sigma}(p_2)$ implies $act_{\sigma}(p_2)$, $X \subseteq X'$ because $\triangle_{\sigma}(p_1)$ implies $\triangle_{\sigma}(p_2)$, and $Y' \setminus Y \subseteq X'$, because if $act_{\sigma}(p_2)$ and not $act_{\sigma}(p_1)$, then $\triangle_{\sigma}(p_2)$. The last observation holds because $p_1 \preceq p_2$, $act_{\sigma}(p_2)$ and not $act_{\sigma}(p_1)$ implies that a token was put in p_2 and it was not removed. Note that all output arcs of p_2 are also output arcs of p_1. Hence, the token put in p_2 cannot be removed. Therefore, $\triangle_{\sigma}(p_2)$. Applying Lemma 2 shows that $\#^{\triangle}_{rel,L}(p_1) \leq \#^{\triangle}_{rel,L}(p_2)$, proving that $\triangle^{\tau}_{rel,L}(p_2)$. Similarly, we can show that $\triangledown^{\tau}_{rel,L}(p_2)$ implies $\triangledown^{\tau}_{rel,L}(p_1)$. □

6 Exploiting Monotonicity During Discovery

The goal of this paper is not to provide a particular discovery algorithm. However, Theorem 2 provides the basis for Apriori-style algorithms [8,9,20]. Such algorithms are used in frequent item-set mining, association rule learning, sequence mining, and episode mining. The basic idea of such algorithms is to avoid spending time on "hopeless candidate patterns" by dramatically pruning the search space. For example, in a supermarket, the number of customers buying products A, B, and C is smaller than (1) the number of the customers buying products A and B, (2) the number of the customers buying products A and C, and (3) the number of the customers buying products B and C. Hence, if one of the latter three groups ($\{A, B\}$, $\{A, C\}$, or $\{B, C\}$) is infrequent, then by definition also the former group ($\{A, B, C\}$) is infrequent. Obviously, we can use the monotonicity results presented in this paper in a similar fashion.

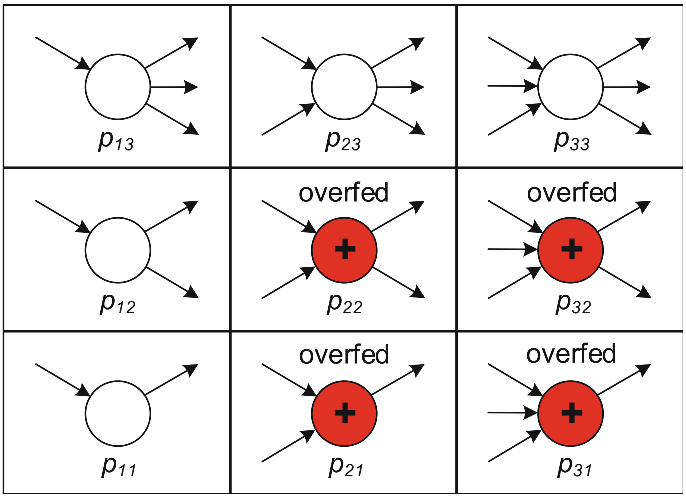


Fig. 5. If place p_{22} is already overfed, then we know that the places p_{32}, p_{21}, and p_{31} also need to be overfed.

Figure 5 sketches the situation where we have evaluated a place p_{22} and the place turned out to be overfed, i.e., at the end of a trace tokens remain. Obviously, the place remains overfed when we remove an output arc or add an input arc. Therefore, by definition, p_{32}, p_{21}, and p_{31} also need to be overfed. Figure 6 describes to opposite situation. If place p_{22} "goes negative" when replaying a trace (i.e., the place is underfed), then the place remains underfed when we remove an input arc or add an output arc. Hence, p_{13}, p_{23}, and p_{12} also need to be underfed.

We can use these insights to prune the search space of candidate places. Assume we have 80 candidate places as shown in Fig. 7(a). We can pick a random

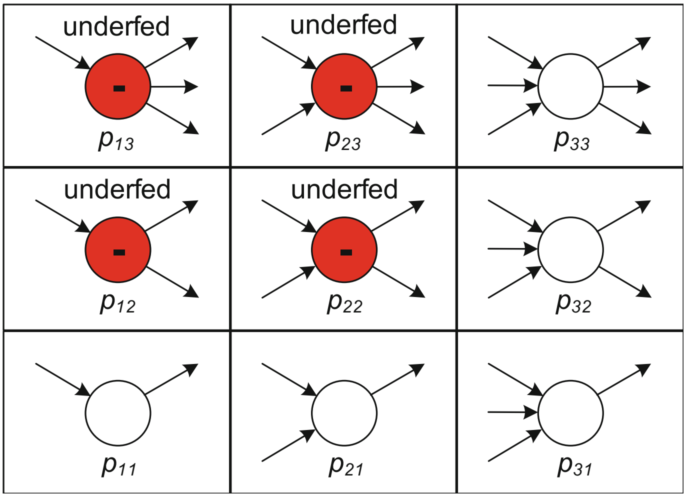
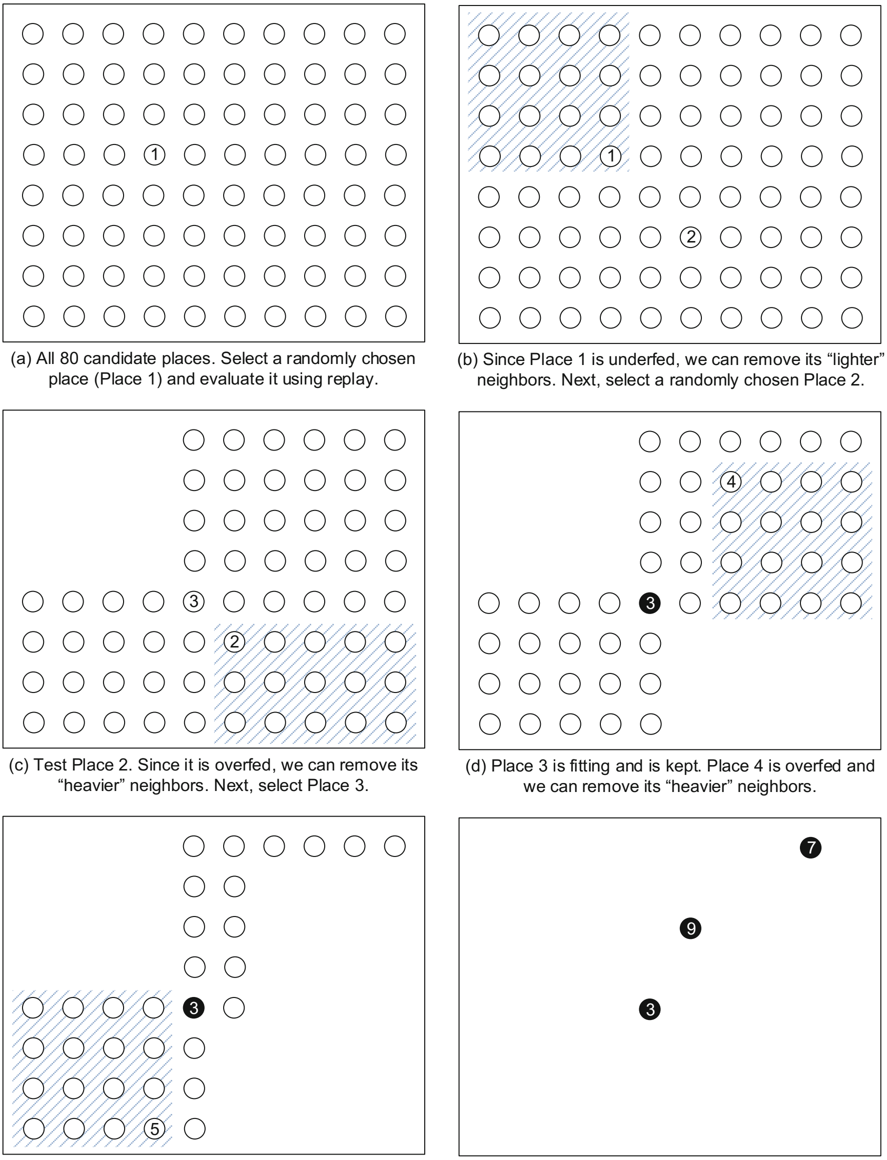


Fig. 6. If place p_{22} is already underfed, then we know that the places p_{13}, p_{23}, and p_{12} also need to be underfed.

candidate place, say Place 1. If this place is underfed according to some criterion (e.g., in more than 10% of the traces the place does not have enough tokens at some stage), then we can identify lighter places that must have the same problem. Assume that Place 1 is indeed underfed and has the lighter neighboring places highlighted in Fig. 7(b). As a result, we can remove 16 candidate places by just evaluating Place 1. Then we pick the next random candidate place, say Place 2. If this place is overfed (e.g., in more than 10% of the traces Place 2 was not empty at the end), then we can identify all heavier places that must have the same problem. These are removed. Figure 7(c) shows that we can remove 15 candidate places by just evaluating Place 2. The next randomly selected candidate place turns out to be fitting (e.g., in 90% of the traces Place 3 was empty at the end and in 90% of the traces there were sufficient tokens), i.e., Place 3 is not underfed and not overfed. In the next step, we remove another 16 candidate places because Place 4 is overfed (Fig. 7(d)). Then we remove another set of places because Place 5 is underfed (Fig. 7(e)). We can repeat the process until there no candidate places left. Figure 7(f) shows the remaining three places. Note that an evaluated place can be both underfitting and overfitting. When encountering such places, the search space can be pruned in two directions (remove all lighter and heavier places). As sketched in Fig. 7, it will often be the case that only a fraction of the candidate places needs to be evaluated using replay techniques.

Figure 7 only sketches the idea and places are selected randomly. One can also think of smarter strategies. For example, one can start with places having just a few connections. Let p_c be a candidate place having n input and output activities, i.e., $n = |\bullet p_c| + |p_c\bullet|$ is the number of arcs. Let $A^{\triangle} = \{p \in \mathbb{P} \mid |\bullet p| + |p\bullet| < n \ \wedge \ p \succeq p_c\}$ and $A^{\triangledown} = \{p \in \mathbb{P} \mid |\bullet p| + |p\bullet| < n \ \wedge \ p \preceq p_c\}$ be the



(a) All 80 candidate places. Select a randomly chosen place (Place 1) and evaluate it using replay.

(b) Since Place 1 is underfed, we can remove its “lighter” neighbors. Next, select a randomly chosen Place 2.

(c) Test Place 2. Since it is overfed, we can remove its “heavier” neighbors. Next, select Place 3.

(d) Place 3 is fitting and is kept. Place 4 is overfed and we can remove its “heavier” neighbors.

(e) Place 5 is underfed and we can remove its “lighter” neighbors.

(f) The process is repeated until there are no unexplored candidate places left. At the end three fitting places remain.

Fig. 7. A process discovery algorithm could simply evaluate all places and only keep those that are fitting (i.e., meet certain quality criteria). However, the monotonicity results in Theorem 2 show that there are many candidate places that we do not need to check. This is the key to discovering places efficiently.

heavier and lighter ancestors of p_c. If for any $p \in A^{\triangle}$, $\triangledown^{\tau}_{\mathit{freq},L}(p)$ or $\triangledown^{\tau}_{\mathit{rel},L}(p)$, then we know that $\triangledown^{\tau}_{\mathit{freq},L}(p_c)$ or $\triangledown^{\tau}_{\mathit{rel},L}(p_c)$. If for any $p \in A^{\triangledown}$, $\triangle^{\tau}_{\mathit{freq},L}(p)$ or $\triangle^{\tau}_{\mathit{rel},L}(p)$, then we know that $\triangle^{\tau}_{\mathit{freq},L}(p_c)$ or $\triangle^{\tau}_{\mathit{rel},L}(p_c)$. These properties can be used to avoid certain checks.

There are many more ways to speedup the search process further:

- Suppose that we consider a place to be overfed when at least 10% of the traces have remaining tokens. This means that we can abort the place evaluation when we have found 10% of traces having problems (for poorly fitting places this may be reached quickly).
- A similar strategy can be used for underfed places. Moreover, the replay of a trace can be aborted when the first problem is encountered.
- If we know the frequencies of all activities in the log, we can do an initial check to see whether the sum of the frequencies of the input activities approximately matches the sum of the frequencies of the input activities (this is also used in [6]). Such aggregate information can be used to guide the pruning process. There can even be guarantees, provided that we can make assumptions about the distribution of activities over traces or bound the trace length.

In short, there are many ways to exploit the monotonicity results provided in this paper.

7 Further Pruning of the Search Space

Next to avoiding checks for places that are obviously “too light” or “too heavy”, we would also like to avoid adding redundant and conflicting places.

In Theorem 1, we showed that $P_1 \Rightarrow P_2$ implies $\mathit{fit}(P_1) \subseteq \mathit{fit}(P_2)$. Hence, adding redundant places does not limit the set of fitting traces and therefore only complicates the process model. This can be exploited while constructing a process model. If two non-overlapping places p_1 and p_2 have been added, one should not consider adding place $p = p_1 \sqcup p_2$.

Moreover, we would also like to avoid adding conflicting places. If two places are in conflict (notation $p_1\#p_2$), then there are traces that could never fit both places.

Definition 14 (Conflict). *Let $p_1, p_2 \in \mathbb{P}$ be two places. $p_1 \prec p_2$ if and only if $p_1 \preceq p_2$ and $p_1 \neq p_2$. $p_1 \succ p_2$ if and only if $p_1 \succeq p_2$ and $p_1 \neq p_2$. $p_1\#p_2$ if and only if $p_1 \prec p_2$ and $p_1 \succ p_2$.*

Theorem 3. *Let $p_1, p_2 \in \mathbb{P}$ be two places and $\sigma \in \mathbb{B}$ a trace. If $\Box_\sigma(p_1)$, $p_1\#p_2$, and $act_\sigma(p_1 \otimes p_2)$, then $\not\Box_\sigma(p_2)$.*

Proof. Assume $\Box_\sigma(p_1)$, $p_1\#p_2$, and $act_\sigma(p_1 \otimes p_2)$. Hence, $p_1 \prec p_2$ or $p_1 \succ p_2$. If $p_1 \prec p_2$, then the number of tokens in p_2 is always at least the number of tokens in p_1 for any sequence, including σ. In fact, in σ there is at least one additional token produced or a token was not consumed (because $act_\sigma(p_1 \otimes p_2)$). Because p_1 ends empty, p_2 must have a remaining token at the end. Hence, σ cannot be fitting ($\not\square_\sigma(p_2)$). If $p_1 \succ p_2$, then the number of tokens in p_2 is always at most the number of tokens in p_1. In fact, there is at least one additional token consumed or a token was not produced (because $act_\sigma(p_1 \otimes p_2)$). Because p_1 ends empty, p_2 must have a missing token at the end. Hence, σ cannot be fitting. □

Consider places $p_1 = (\{a\}, \{b, c\})$ and $p_2 = (\{a, d\}, \{b\})$. Obviously, $p_1 \prec p_2$. For a trace involving c and/or d activities, it can never be the case that both places are fitting. Since p_1 and p_2 disagree on the allowed behavior, one would not like to add both to the same process model. Also this property can be exploited during discovery.

8 How About Conformance Checking?

Process mining is not limited to process discovery and includes conformance checking, model repair, performance analysis, decision mining, and organizational mining. Moreover, also predictive and prescriptive analytics are supported by process mining tools and techniques [3]. Viewing a process model as merely a collection of independent places may also help to expedite these other analysis tasks. Consider for example conformance checking which involves detecting and diagnosing both differences and commonalities between an event log and a process model [4]. Typically, four dimensions are distinguished: fitness, precision, generalization, and simplicity. State-of-the-art techniques use so-called alignments or token-based replay [3]. However, these techniques often do not have the monotonicity properties one expects [23]. For example, removing a place should never result in a better precision or lower fitness. In [2] a probabilistic angle is added to these questions, also revealing obvious problems related to existing conformance measures. The monotonicity results presented in this paper may provide a fresh look on conformance problems. First of all, it could be good to check places individually. Second, there are ways to quickly analyze whether a place is overfed and/or underfed.

9 Conclusion

For this Festschrift celebrating Farhad Arbab's achievements in coordination models and languages, I decided to focus on the discovery of a very simple coordination model: "places". We like to learn such coordination structures from observed behaviors.

The process models in this paper are fully described by places. For the semantics, we employ an open-world assumption and special start and end activities. This yields a representation very suitable for process mining. Given a particular behavior, a place can be "fitting", "underfed" (tokens are missing), or "overfed" (tokens are remaining). We would like to discover fitting places from event data satisfying predefined quality criteria. This is not a trivial task and for large event logs this easily becomes intractable. Therefore, we studied monotonicity properties in the context of event logs. For example, if place p_1 is "lighter" than place p_2 (i.e., $p_1 \preceq p_2$) and 5% of the activated traces produce too few tokens for p_2 ($\bigtriangledown^{0.05}_{rel,L}(p_2)$), then the same traces also produce too few tokens for p_1 (i.e., $\bigtriangledown^{0.05}_{rel,L}(p_1)$). This helps to prune the set of candidate places. Moreover, also notions like redundancy and conflict can be used to reduce the search space further. These properties allow for new Apriori-style algorithms. The insights could also be used to speed-up the discovery of hybrid process models [6].

This contribution did not show how to synthesize Reo circuits from event data. However, this remains an interesting question and I encourage the Reo community to look into this. Finally, I would like to wish Farhad all the best and hope that he will remain working on the "science of coordination" after his "coordination of science" activities at CWI have ended.

References

1. van der Aalst, W.M.P.: Decomposing Petri nets for process mining: a generic approach. Distrib. Parallel Databases **31**(4), 471–507 (2013)
2. van der Aalst, W.M.P.: Mediating between modeled and observed behavior: the quest for the "Right" process. In: IEEE International Conference on Research Challenges in Information Science (RCIS 2013), pp. 31–43. IEEE Computing Society (2013)
3. van der Aalst, W.M.P.: Process Mining: Data Science in Action. Springer, Berlin (2016). https://doi.org/10.1007/978-3-662-49851-4
4. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.: Replaying history on process models for conformance checking and performance analysis. WIREs Data Mining Knowl. Discov. **2**(2), 182–192 (2012)
5. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of workflow nets: classification, decidability, and analysis. Formal Aspects Comput. **23**(3), 333–363 (2011)
6. van der Aalst, W.M.P., De Masellis, R., Di Francescomarino, C., Ghidini, C.: Learning hybrid process models from events. In: Carmona, J., Engels, G., Kumar, A. (eds.) BPM 2017. LNCS, vol. 10445, pp. 59–76. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65000-5_4
7. van der Aalst, W.M.P., Rubin, V., Verbeek, H.M.W., van Dongen, B.F., Kindler, E., Günther, C.W.: Process mining: a two-step approach to balance between underfitting and overfitting. Softw. Syst. Model. **9**(1), 87–111 (2010)
8. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proceedings of the 20th International Conference on Very Large Data Bases (VLDB), pp. 487–499. Morgan Kaufmann Publishers Inc., Santiago de Chile (1994)
9. Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proceedings of the 11th International Conference on Data Engineering (ICDE 1995), pp. 3–14. IEEE Computer Society (1995)
10. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
11. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process mining based on regions of languages. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 375–383. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75183-0_27

12. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Deriving Petri nets from finite transition systems. IEEE Trans. Comput. **47**(8), 859–882 (1998)
13. Desel, J., Esparza, J.: Free Choice Petri Nets, Volume 40 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge (1995)
14. Ehrenfeucht, A., Rozenberg, G.: Partial (set) 2-structures - part 1 and part 2. Acta Inform. **27**(4), 315–368 (1989)
15. Günther, C.W., van der Aalst, W.M.P.: Fuzzy mining: adaptive process simplification based on multi-perspective metrics. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) International Conference on Business Process Management (BPM 2007). LNCS, vol. 4714, pp. 328–343. Springer-Verlag, Berlin (2007). https://doi.org/10.1007/978-3-540-75183-0_24
16. Jongmans, S.S.T.Q., Arbab, F.: Overview of thirty semantic formalisms for Reo. Sci. Ann. Comput. Sci. **22**(1), 201–251 (2012)
17. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs containing infrequent behaviour. In: Lohmann, N., Song, M., Wohed, P. (eds.) BPM 2013. LNBIP, vol. 171, pp. 66–78. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06257-0_6
18. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from incomplete event logs. In: Ciardo, G., Kindler, E. (eds.) PETRI NETS 2014. LNCS, vol. 8489, pp. 91–110. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07734-5_6
19. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Scalable process discovery with guarantees. In: Gaaloul, K., Schmidt, R., Nurcan, S., Guerreiro, S., Ma, Q. (eds.) CAISE 2015. LNBIP, vol. 214, pp. 85–101. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19237-6_6
20. Mannila, H., Toivonen, H., Verkamo, A.I.: Discovery of frequent episodes in event sequences. Data Mining Knowl. Discov. **1**(3), 259–289 (1997)
21. Meng, S., Arbab, F., Baier, C.: Synthesis of Reo circuits from scenario-based interaction specifications. Sci. Comput. Program. **76**(8), 651–680 (2011)
22. Sole, M., Carmona, J.: Process mining from a basis of regions. In: Lilius, J., Penczek, W. (eds.) Applications and Theory of Petri Nets 2010. LNCS, vol. 6128, pp. 226–245. Springer, Berlin (2010). https://doi.org/10.1007/978-3-642-13675-7_14
23. Tax, N., Lu, X., Sidorova, N., Fahland, D., van der Aalst, W.M.P.: The imprecisions of precision measures in process mining. Inf. Process. Lett. **135**, 1–8 (2018)
24. Weijters, A.J.M.M., van der Aalst, W.M.P.: Rediscovering workflow models from event-based data using little thumb. Integrated Comput.-Aided Eng. **10**(2), 151–162 (2003)
25. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process discovery using integer linear programming. Fundamenta Inform. **94**, 387–412 (2010)

Self-stabilization Through the Lens of Game Theory

Krzysztof R. Apt[1,2](✉) and Ehsan Shoja[3]

[1] CWI, Amsterdam, The Netherlands
apt@cwi.nl
[2] MIMUW, University of Warsaw, Warsaw, Poland
[3] Sharif University of Technology, Tehran, Iran

Abstract. In 1974 Dijkstra introduced the seminal concept of self-stabilization that turned out to be one of the main approaches to fault-tolerant computing. We show here how his three solutions can be formalized and reasoned about using the concepts of game theory. We also determine the precise number of steps needed to reach self-stabilization in his first solution.

1 Introduction

In 1974 Dijkstra introduced in a two-page article [10] the notion of self-stabilization. The paper was completely ignored until 1983, when Leslie Lamport stressed its importance in his invited talk at the ACM Symposium on Principles of Distributed Computing (PODC), published a year later as [21]. Things have changed since then. According to Google Scholar Dijkstra's paper has been by now cited more than 2300 times. It became one of the main approaches to fault tolerant computing. An early survey was published in 1993 as [26], while the research on the subject until 2000 was summarized in the book [13]. In 2002 Dijkstra's paper won the PODC influential paper award (renamed in 2003 to Dijkstra Prize). The literature on the subject initiated by it continues to grow. There are annual Self-Stabilizing Systems Workshops, the 18th edition of which took part in 2016.

The idea proposed by Dijkstra is very simple. Consider a distributed system viewed as a network of machines. Each machine has a local state and can change it autonomously by inspecting its local state and the local states of its neighbours. Some global states are identified as *legitimate*. A distributed system is called self-stabilizing if it satisfies the following three properties (the terminology is from [5]):

closure: starting from an arbitrary global state, the system is guaranteed to reach a legitimate state,
stability: once a legitimate state is reached, the system remains in it forever,
fairness: in every infinite sequence of moves every machine is selected infinitely often.

F. de Boer et al. (Eds.): Arbab Festschrift, LNCS 10865, pp. 21–37, 2018.
https://doi.org/10.1007/978-3-319-90089-6_2

Dijkstra proposed in [10] three solutions to self-stabilization in which, respectively, n, four and three state machines were used, where n is the number of machines. The proofs were provided respectively in [7] (republished as [11]), [8,9] (republished with small modifications as [12]). In his solutions a legitimate state is identified with the one in which exactly one machine can change its state.

In this paper we show how Dijkstra's solutions to self-stabilization can be naturally formulated using the standard concepts of strategic games, notably the concept of an improvement path. Also we show how one can reason about them using game-theoretic terms. We focus on Dijkstra's first solution but the same approach can be adopted to other solutions.

The connections between self-stabilization and game theory were noticed before. We discuss the relevant references in the final section. The analysis of the original Dijkstra's solutions using game theory is to our knowledge new.

This paper connects two unrelated areas, each of which has developed its own well-established notation and terminology. To avoid possible confusion, let us clarify that in what follows S_i denotes a set of strategies of a player in a strategic game, while the letter S denotes a variable in a solution to the self-stabilization problem. Further, the notion of a state in the self-stabilization refers to the range of a variable and not to an assignment of values to all variables, as is customary in the area of program semantics.

2 Preliminaries

A ***strategic game*** $\mathcal{G} = (S_1, \ldots, S_n, p_1, \ldots, p_n)$ for $n > 1$ players consists of a non-empty set S_i of ***strategies*** and a ***payoff function*** $p_i : S_1 \times \cdots \times S_n \to \mathbb{R}$, for each player i. We denote $S_1 \times \cdots \times S_n$ by S, call each element $s \in S$ a ***joint strategy*** and abbreviate the sequence $(s_j)_{j \neq i}$ to s_{-i}. Occasionally we write (s_i, s_{-i}) instead of s. We call a strategy s_i of player i a ***best response*** to a joint strategy s_{-i} of his opponents if for all $s'_i \in S_i$, $p_i(s_i, s_{-i}) \geq p_i(s'_i, s_{-i})$. A joint strategy s is called a ***Nash equilibrium*** if each s_i is a best response to s_{-i}. (In the literature these equilibria are often called *pure Nash equilibria* to distinguish them from Nash equilibria in mixed strategies. The latter ones have no use in this paper.)

Further, we call a strategy s'_i of player i a ***better response*** given a joint strategy s if $p_i(s'_i, s_{-i}) > p_i(s_i, s_{-i})$. We call $s \to s'$ an ***improvement step*** (abbreviated to a ***step***) if $s' = (s'_i, s_{-i})$ for some better response s'_i of player i given s. So $p_i(s') > p_i(s)$.

An ***improvement path*** is a maximal sequence

$$s^1 \to s^2 \to \ldots \to s^k \to \ldots$$

such that each $s^i \to s^{i+1}$ is an improvement step.

In the next section we consider specific strategic games on directed graphs. Fix a finite directed graph G. We say that a node j is a ***neighbour*** of the node i in G if there is an edge $j \to i$ in G. Let N_i denote the set of all neighbours of node i in the graph G. We now consider a strategic game in which each player is a node in G. Fix a non-empty set of strategies C that we call ***colours***.

We divide the players in two categories: those who play a coordination game and those who play an anti-coordination game. More specifically,

- the players are the nodes of G,
- the set of strategies of player (node) i is a set of colours $A(i)$ such that $A(i) \subseteq C$,
- if the player plays the coordination game, then his payoff function is defined by
$$p_i(s) = |\{j \in N_i \mid s_i = s_j\}|,$$
- if the player plays the anti-coordination game, then his payoff function is defined by
$$p_i(s) = |\{j \in N_i \mid s_i \neq s_j\}|.$$

So each node simultaneously chooses a colour and the payoff to the player who plays the *coordination game* is the number of its neighbours that chose its colour, while the payoff to the player who plays the *anti-coordination game* is the number of its neighbours that chose a different colour.

The games on directed graphs in which all players were playing the coordination game were studied in [4]. Corresponding games on undirected graphs were considered in [2] and on weighted undirected graphs in [25]. In turn, the games in which some players played the coordination game while other players played the anti-coordination game were studied (in a more general context of weighted hypergraphs) in [28]. If the underlying (weighted) graph is undirected the game always has a Nash equilibrium, which is not the case if the graph is directed. The absence of Nash equilibria is crucial in the context of this paper.

We now move on to the subject of this paper and introduce the following concepts concerning improvement paths.

Definition 1. *Fix a strategic game.*

- *A joint strategy is* ***legitimate*** *if exactly one player does not play a best response in it.*
- *An improvement path ensures*
  - ***closure*** *if some joint strategy in it is legitimate,*
  - ***stability*** *if the successors of the legitimate joint strategies in it are legitimate,*
  - ***fairness*** *if every player is selected in it infinitely often,*
  - ***self-stabilization (in*** k ***steps)*** *if every player is selected in it infinitely often and from a certain point (**after** k **steps**) each joint strategy in it is legitimate.*
- *A game* ***admits closure/stability/fairness*** *if it is ensured by every improvement path in it.*
- *A game* ***admits self-stabilization*** *(**in** k **steps**) if it is ensured by every improvement path in it (**in** k **steps**).*

For a more refined analysis we shall need the concept of a scheduler.

Definition 2.

- *A* ***scheduler*** *is a function f that given a joint strategy s that is not a Nash equilibrium and a player i who does not hold in s a best response selects a strategy $f(s,i)$ for i that is a better response given s.*
- *Consider a scheduler f. An improvement path*

$$s^1 \rightarrow s^2 \rightarrow \ldots \rightarrow s^k \rightarrow \ldots,$$

is ***generated by*** *f if for each $k \geq 1$, if s^k is not a Nash equilibrium, then for some $i \in \{1, \ldots, n\}$, $s^{k+1} = (f(s^k, i), s^k_{-i})$.*
- *A scheduler f* ***ensures self-stabilization*** *(**in** k **steps**) if every improvement path generated by it ensures self-stabilization (**in** k **steps**).*

So a game admits self-stabilization (in k steps) if every scheduler ensures self-stabilization (in k steps). Schedulers in the context of strategic games were extensively considered in [3], though they selected a player and not his strategy. The ones used here correspond in the terminology of [3] to the state-based schedulers.

3 Dijkstra's First Solution

We start by recalling the first solution to the self-stabilization problem given in [10]. We assume a directed ring of n machines, each having a local variable and a program. The variables assume the values from the set $\{0, \ldots, k-1\}$, where $k \geq n$ and $\oplus$ stands for addition modulo k. Each program consists of a single rule of the form

$$P \rightarrow A$$

where P is a condition, called a *priviledge*, on the local variables of the machine and its predecessor in the ring, and A is an assignment to the local variable. The variable of a considered machine is denoted by S and the variable of its predecessor by L.

The program for machine 1 is given by the rule

$$L = S \ \rightarrow \ S := S \oplus 1$$

and for the other machines by the rule

$$L \neq S \ \rightarrow \ S := L.$$

One assumes that each time a machine is selected, its priviledge is true. Dijkstra proved in [11] (that originally appeared as [7]) that starting from an arbitrary initial situation any sequence of machine selections leads to a situation in which

- exactly one priviledge is true,
- this property remains true forever.

Moreover, every machine is selected in this sequence infinitely often.

In the terminology introduced in the Introduction the above system of machines is self-stabilizing.

We can model the above solution by means of the following strategic game $\mathcal{G}$ on a directed ring involving n players:

- each player has the same set C of strategies (called colours), where $|C| \geq 2$,
- exactly one player plays the anti-coordination game on the ring,
- all other players play the coordination game on the ring.

To fix notation we assume that it is player 1 who plays the anti-coordination game. So the payoff functions are simply:

$$p_1(s) := \begin{cases} 0 & \text{if } s_1 = s_n \\ 1 & \text{otherwise} \end{cases}$$

and for $i \neq 1$

$$p_i(s) := \begin{cases} 0 & \text{if } s_i \neq s_{i-1} \\ 1 & \text{otherwise} \end{cases}$$

We arrange the colours in C in a cyclic order and given a colour c we denote its successor in this order by c'. The following result provides a game-theoretic account of the above solution to the self-stabilization problem.

Theorem 1. *Consider the game $\mathcal{G}$. Suppose that $n \geq 3$ and $|C| \geq n$. Let f be a scheduler such that*

$$f(s, 1) = s'_1.$$

Then f ensures self-stabilization in $\mathcal{G}$.

Thus the only restriction on the scheduler f is that for player 1 it selects the next colour in the cyclic order on C (as s'_1 denotes the successor of s_1).

Proof. There is a 1–1 correspondence between the maximal sequences of moves of the machines in Dijkstra's solution and the improvement paths generated by the schedulers satisfying the stated condition. □

We shall return to the above result in Sect. 6. It is useful to point out why we did not incorporate the specific choice of the strategies into the payoff functions and used a scheduler instead. This alternative would call for selecting $\{0, \ldots, k-1\}$ as the set of strategies for each player and using the following payoff function for player 1, where $\oplus$ stands for addition modulo k:

$$p_1(s) := \begin{cases} 0 & \text{if } s_1 \neq s_n \oplus 1 \\ 1 & \text{otherwise} \end{cases}$$

However, the resulting game would then correspond to a setup in which the program for machine 1 is

$$S \neq L \oplus 1 \; \rightarrow \; S := L \oplus 1.$$

Moreover, the resulting game does not admit self-stabilization (and a fortiori the resulting programs for the machines do not form a solution for self-stabilization). Indeed, assume three players and $k = 3$, so that the strategies of the players are 0, 1, 2. Then the following infinite improvement path does not ensure closure:

$$(200 \to 220 \to 120 \to 122 \to 112 \to 012 \to 011 \to 001 \to 201 \to)^*,$$

where each joint strategy is displayed as a string of three numbers from $\{0, 1, 2\}$ and * stands for the infinite repetition of the exhibited prefix of an improvement path.

4 Dijkstra's Three-State Solution

Next we discuss Dijkstra's three-state solution to the self-stabilization problem. We follow here the presentation he gave in [12], where he provided a particularly elegant correctness proof.

There are n machines arranged in an undirected ring, the first one called the *bottom* machine, the last one called the *top* machine, and the other machines called *normal*.

The condition of each rule is now on the local variables of the machine and its two neighbours. The variable of a considered machine is denoted by S, of its left neighbour by L and of its right neighbour by R. All variables range over the set $\{0, 1, 2\}$ and $\oplus$ stands for addition modulo 3.

The program for the bottom machine is given by the rule

$$S \oplus 1 = R \ \rightarrow\ S := S \oplus 2,$$

for each normal machine by the rule

$$L = S \oplus 1 \vee S \oplus 1 = R \ \rightarrow\ S := S \oplus 1,$$

and for the top machine by the rule

$$L = R \wedge S \neq R \oplus 1 \ \rightarrow\ S := R \oplus 1.$$

Dijkstra proved that the above system of machines is self-stabilizing.

This solution can be represented and reasoned about using strategic games, though these games are not anymore coordination or anti-coordination games. First note that, in contrast to the case of Dijkstra's first solution, this solution cannot be modeled using strategic games with 0/1 payoffs. To see it assume $n = 3$ and consider the global state of the system described by $(2, 1, 0)$. Then the priviledge of machine 2 is true, since $L = S \oplus 1$, as $2 = 1 \oplus 1$. After machine 2 is selected the global state changes to $(2, 2, 0)$. In this state the priviledge of machine 2 is again true, since $S \oplus 1 = R$, as $2 \oplus 1 = 0$. So in the improvement path of the corresponding strategic game player 2 can be selected twice in succession. This can be modelled only using at least three payoff values.

To capture such a possibility we need to analyze when a machine can be selected twice in succession. This can happen when successively $L = S \oplus 1$ and $S \oplus 1 = R$ are true or successively $S \oplus 1 = R$ and $L = S \oplus 1$ are true. Taking into account the action of the assignment $S := S \oplus 1$ the first possibility means that initially $L = S \oplus 1 \land S \oplus 2 = R$ is true and the second possibility that initially $S \oplus 1 = R \land L = S \oplus 2$ is true. These two options can be rewritten as $S \oplus 1 \in \{L, R\} \land S \oplus 2 \in \{L, R\}$.

To complete this analysis note that a machine can be selected only once in succession, when initially $L = S \oplus 1 \land S \oplus 2 \neq R$ is true or $S \oplus 1 = R \land L \neq S \oplus 2$ is true, which can be rewritten as $S \oplus 1 \in \{L, R\} \land S \oplus 2 \notin \{L, R\}$.

Translating it into a game-theoretic notation that uses indices we are brought into the following strategic game $\mathcal{G}$ for n players. Each player has $\{0, 1, 2\}$ as the set of strategies. The payoff functions are defined as follows, where we assume that player 1 corresponds to the bottom machine and player n to the top machine:

$$p_1(s) := \begin{cases} 0 & \text{if } s_1 \oplus 1 = s_2 \\ 1 & \text{otherwise} \end{cases}$$

for $1 < i < n$

$$p_i(s) := \begin{cases} 0 & \text{if } s_i \oplus 1 \in \{s_{i-1}, s_{i+1}\} \land s_i \oplus 2 \in \{s_{i-1}, s_{i+1}\} \\ 1 & \text{if } s_i \oplus 1 \in \{s_{i-1}, s_{i+1}\} \land s_i \oplus 2 \notin \{s_{i-1}, s_{i+1}\} \\ 2 & \text{otherwise} \end{cases}$$

$$p_n(s) := \begin{cases} 0 & \text{if } s_1 = s_{n-1} \land s_n \neq s_1 \oplus 1 \\ 1 & \text{otherwise} \end{cases}$$

Dijkstra's result concerning the above system of three-state machines is captured by the following theorem.

Theorem 2. *Consider the above game $\mathcal{G}$. Suppose that $n \geq 3$. Let f be a scheduler such that*

$$\begin{aligned} &f(s, 1) = s_1 \oplus 2, \\ &f(s, i) = s_i \oplus 1, \ \textit{where } 1 < i < n, \\ &f(s, n) = s_1 \oplus 1. \end{aligned}$$

Then f ensures self-stabilization in $\mathcal{G}$.

Proof. Every maximal sequence of moves of the machines in Dijkstra's three-state solution corresponds to an improvement path generated by a scheduler satisfying the stated conditions. Conversely, every improvement path generated by a scheduler satisfying the stated conditions corresponds to a maximal sequence of moves of the machines in Dijkstra's three-state solution with each improvement step that results for a player i in the payoff increase by 2 mapped to two consecutive moves of machine i. □

5 A Four-State Solution

Finally, we consider a four-state solution. Instead of Dijkstra's solution that uses two Boolean variables per machine we consider a modified solution due to [16] that uses per machine a single variable that can take four values. We assume the set up and terminology of the previous section, with the following differences.

The variable of machine 1 now ranges over $\{1, 3\}$, of machine n over $\{0, 2\}$. and all other variables range over $\{0, 1, 2, 3\}$. Further, $\oplus$ stands now for addition modulo 4.

The program for the bottom machine is given by the rule

$$S \oplus 1 = R \; \rightarrow \; S := S \oplus 2,$$

for each normal machine by the rule

$$L = S \oplus 1 \lor S \oplus 1 = R \; \rightarrow \; S := S \oplus 1,$$

and for the top machine by the rule

$$L = S \oplus 1 \; \rightarrow \; S := S \oplus 2.$$

Following the considerations of the previous section this solution can be modeled by the following strategic game $\mathcal{G}$ for n players. The sets of strategies are as follows: for player 1: $\{1, 3\}$, for player n: $\{0, 2\}$, and for all other players: $\{0, 1, 2, 3\}$.

The payoff functions are defined as follows, where we assume that player 1 corresponds to the bottom machine and player n to the top machine:

$$p_1(s) := \begin{cases} 0 & \text{if } s_1 \oplus 1 = s_2 \\ 1 & \text{otherwise} \end{cases}$$

for $1 < i < n$

$$p_i(s) := \begin{cases} 0 & \text{if } s_i \oplus 1 \in \{s_{i-1}, s_{i+1}\} \land s_i \oplus 2 \in \{s_{i-1}, s_{i+1}\} \\ 1 & \text{if } s_i \oplus 1 \in \{s_{i-1}, s_{i+1}\}) \land s_i \oplus 2 \notin \{s_{i-1}, s_{i+1}\} \\ 2 & \text{otherwise} \end{cases}$$

$$p_n(s) := \begin{cases} 0 & \text{if } s_n \oplus 1 = s_{n-1} \\ 1 & \text{otherwise} \end{cases}$$

The reason for using three values in the payoff functions p_i, where $1 < i < n$, is as in the previous section. The corresponding result concerning self-stabilization of the above system of four-state machines is now captured by the following game-theoretic theorem.

Theorem 3. *Consider the above game $\mathcal{G}$. Suppose that $n \geq 3$. Let f be a scheduler such that*

$$\begin{aligned} &f(s, 1) = s_1 \oplus 2, \\ &f(s, i) = s_i \oplus 1, \; \textit{where } 1 < i < n, \\ &f(s, n) = s_n \oplus 2. \end{aligned}$$

Then f ensures self-stabilization in $\mathcal{G}$.

Proof. The same as the proof of Theorem 2. □

6 A Game-Theoretic Analysis of the First Solution

We now analyze in detail the strategic game $\mathcal{G}$ introduced in Sect. 3 with the aim of proving a stronger result about the first solution to self-stabilization. We begin with the following observation.

Note 1. The game $\mathcal{G}$ admits no Nash equilibria.

Proof. Suppose otherwise. Let s be a Nash equilibrium of $\mathcal{G}$. Then every player $i \neq 1$ holds in s the colour of its predecessor. Hence all players hold in s the same colour, in particular players 1 and n. But then player 1 does not hold in s a best response, which yields a contradiction. □

Corollary 1. *The game $\mathcal{G}$ admits stability.*

Proof. Suppose $s \rightarrow s'$ is an improvement step in the game $\mathcal{G}$ and that s is legitimate. Then by the definition of the game either s' is legitimate or is a Nash equilibrium. So the claim follows by Note 1. □

We shall use below the following observation.

Note 2. Consider a coordination game on a chain of n players in which each player has the same set of strategies. Then all improvement paths in this game are of length $\leq \frac{n(n-1)}{2}$. Further, improvement paths of length $\frac{n(n-1)}{2}$ exist.

Proof. Suppose the chain is $1 \rightarrow 2 \rightarrow \ldots \rightarrow n$. Consider an improvement path ξ. Each player i can adopt in ξ at most $i-1$ colours, namely the strategies held by his predecessors in the chain. So each player i can be involved in at most $i-1$ improvement steps. Consequently the length of ξ is bound by $\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2}$.

To establish the second claim take an initial joint strategy s in which all colours differ. Then the required number of steps is achieved by scheduling the players in the 'rightmost first' order, so

$$(n), (n-1, n), (n-2, n-1, n), \ldots, (2, 3, \ldots, n),$$

where to increase readability we separated the consecutive phases using brackets. □

Theorem 4. *The game $\mathcal{G}$ admits fairness.*

Proof. Consider an improvement path ξ. We first prove that player 1 is infinitely often selected in ξ. Suppose otherwise. By Note 1 ξ is infinite, so from some moment on player 1 is never selected in the infinite suffix ϕ of ξ. Break the ring by removing the link between players n and 1 and consider the resulting coordination game on the chain $1 \to 2 \to \ldots \to n$. Then ϕ is an infinite improvement path in this game, which contradicts Note 2.

Note now that if some player i is finitely often selected in ξ, then so is its successor. Together with the above conclusion this implies successively that players $n, n-1, \ldots, 2$ are infinitely often selected in ξ. □

So to prove that $\mathcal{G}$ admits self-stabilization we only need to check that it admits closure. However, this holds only for games with two or three players. In fact, we have the following result.

Theorem 5. *Consider the game $\mathcal{G}$.*

(i) If $n = 2$ then $\mathcal{G}$ admits self-stabilization in 0 steps.
(ii) If $n = 3$ then $\mathcal{G}$ admits self-stabilization in 2 steps.
(iii) If $n > 3$ then $\mathcal{G}$ does not admit self-stabilization.

Proof. For simplicity we view each joint strategy as a string over the set of colours that we denote by the initial letters of the alphabet. Different letters stand for different colours.

(i) In this case every joint strategy is legitimate.
(ii) For brevity we say that a joint strategy s is an *i-strategy*, where $0 \leq i \leq 2$, if exactly i players hold in s a best response. The only 0-strategy is of the form aba. We reach from it in one step a 1-strategy cba (assuming $|C| > 2$) or a 2-strategy bba, aaa or abb.

So consider now an arbitrary 1-strategy. If it is player 1 who plays the best response, then s is of the form acb (so in this case $|C| > 2$). Then the only possible improvement steps are $acb \to aab$ or $acb \to acc$. In both cases we reach a 2-strategy in one step.

If it is player 2 who plays the best response, then s is of the form aaa or aab, which contradicts the fact that s is a 1-strategy. Finally, if it is player 3 who plays the best response, then s is of the form baa or aaa, which also contradicts the fact that s is a 1-strategy.

We conclude that a legitimate joint strategy is always reached in at most 2 steps.

(iii) Assume that $n > 3$. Then the following infinite improvement path does not ensure closure:

$$\begin{aligned}(bba^{n-4}ab \to aba^{n-4}ab \to aba^{n-4}aa \to^* abb^{n-4}ba \to \\ aab^{n-4}ba \to bab^{n-4}ba \to bab^{n-4}bb \to^* baa^{n-4}ab \to)^*,\end{aligned}$$

where each inner * stands for an appropriate sequence of $n-4$ improvement steps, while the outer * stands for the infinite repetition of the exhibited prefix of an improvement path. □

The above result explains the need for a scheduler. As before we assume a cyclic order on the set of colours and denote the successor of colour c by c'. The following result improves upon Theorem 1. The differences are discussed after the proof.

Theorem 6. *Consider the game $\mathcal{G}$. Suppose that $n \geq 3$ and $|C| \geq n-1$. Let f be a scheduler such that*

$$f(s,1) = s'_1.$$

Then f ensures self-stabilization in $\mathcal{G}$ in $\frac{1}{2}(3n+1)(n-2)$ steps.

Proof. We split the proof in two parts. The slightly unusual naming of joint strategies in Part 1 will become clear in Part 2.

Part 1: self-stabilization.

Consider an improvement path ξ generated by the scheduler f that starts in a joint strategy s. Call a joint strategy *lean* if the players $2, \ldots, n$ hold in it at most $n-2$ different colours. We now establish a number of claims about ξ.

Claim 1. A lean joint strategy appears in ξ.

Proof. By Theorem 4 eventually some player $i \in \{3, \ldots, n\}$ is selected in ξ. The resulting joint strategy becomes then lean. □

Let s'' be the first lean joint strategy in ξ. Call a colour *fresh* in ξ if it is not held in s'' by any player $i \neq 1$. Fresh colours exist since $|C| \geq n-1$. Let c be the first fresh colour that follows, in the cyclic order on C, the colours that are held in s'' by players $i \neq 1$.

Claim 2. Player 1 eventually introduces in ξ the colour c.

Proof. By the definition of the scheduler and Theorem 4. □

Claim 3. Player 1 eventually introduces in ξ the successor c' of the colour c.

Proof. By the definition of the scheduler and Theorem 4. □

Consider now the joint strategies s^1 and s^5 resulting from the steps described in Claims 2 and 3. Let

$$s^4 \rightarrow s^5$$

be the last step of the segment $s^1 \rightarrow^* s^5$ of ξ. So $s^1_1 = s^4_1 = s^4_n = c$ and $s^5_1 = c'$.

Take now a joint strategy s^6 from the segment $s^1 \rightarrow^* s^5$, different from s^1 and s^5. In s^6 player 1 is not selected. Moreover, by the definition of the game, each better response of a player different than 1 is the colour of his predecessor. So only player 1 can introduce in ξ colour c.

This implies by induction that each time some player i switches in s^6 to the colour c, all players $1, \ldots, i-1$ hold in s^6 the colour c. So the only possibility that player n holds the colour c in s^4 is that all players hold in s^4 the colour c. Informally, the colour c 'travelled the whole ring'. So s^4 is a legitimate joint strategy. Hence by Corollary 1 and Theorem 4 the scheduler f ensures self-stabilization.

Part 2: computing the bound.

Recall that s'' is the first lean joint strategy in ξ. Let s' be the first joint strategy in the segment $s \to^* s''$ of ξ such that in the segment $s' \to^* s''$ player 1 is not selected. We first determine the maximum number of steps in the prefix $s \to^* s'$ of ξ. Since s'' is the first lean joint strategy in ξ, in the prefix $s \to^* s'$ only players 1 and 2 are selected. Moreover, by the choice of s' the last step in this prefix involves player 1. Further, player 1 can be selected the second time only after player n has been selected and no player can be selected twice in succession. These constraints leave only two possible schedulings that yield $s \to^* s'$, namely 1 and 2, 1.

However, the prefix $s \to^* s'$ cannot have 2 steps. Indeed, otherwise it would have the form

$$(c_1, c_2, \ldots, c_n) \to (c_1, c_1, c_3, \ldots, c_n) \to (c'_n, c_1, c_3, \ldots, c_n),$$

where $c_1 = c_n$. So $(c_1, c_1, c_3, \ldots, c_n)$ is lean, which contradicts the choice of s'' as the first lean joint strategy in ξ. Consequently the prefix $s \to^* s'$ can have at most 1 step.

Let now ξ' be the suffix of ξ that starts in s'. We now determine the number of steps in ξ' that yield self-stabilization. We can assume that it takes in ξ at least three steps to reach s^5, as otherwise the bound holds. Consider the last three steps in ξ that lead to s^5:

$$s^2 \to s^3 \to s^4 \to s^5.$$

We noticed already that in s^4 all players hold the colour c. Also, the last n steps in ξ that lead to s^4 consist of switching to the colour c. Hence s^2 is of the form $(c, \ldots, c, a, b)$, where $a \neq c$ and $b \neq c$.

Case 1 $a = b$.

Then s^2 is legitimate. We first compute the number of steps in the prefix χ of ξ' leading from s' to s^4. Consider some player i. In χ he can be involved in two types of steps:

- in which he switches to a colour held in s' by one his predecessors $1, \ldots, i-1$,
- in which he switches to a colour introduced in χ by player 1 (to identify such steps in χ we can 'mark' such colours in some way).

The first possibility leads to at most $i - 1$ steps, while the second one to at most $n - 2$ steps since starting from the lean joint strategy s'' (and hence from s') player 1 can change his colour in χ at most $n - 2$ times. This means that the total number of steps in χ is at most

$$\sum_{i=1}^{n}(i - 1 + n - 2) = \frac{n(n-1)}{2} + n(n-2).$$

Deducting 2 for the steps $s^2 \rightarrow s^3 \rightarrow s^4$ we get the bound $\frac{n(n-1)}{2} + n(n-2) - 2$ on the number of steps in ξ' that yield self-stabilization.

Case 2 $a \neq b$.

Then s^2 is not legitimate but s^3 is, so we need to compute the number of steps in ξ' leading from s' to s^3. To this end we modify ξ' to another improvement path ψ by replacing the step $s^2 \rightarrow s^3$ by

$$s^2 \rightarrow (c, \ldots, c, a, a) \rightarrow s^3$$

and apply the reasoning from Case 1 to ψ. This yields the above bound on the number of steps in ψ needed to reach $(c, \ldots, c, a, a)$ and hence the same bound on the number of steps in ξ' leading from s' to s^3.

We noticed already that the prefix $s \rightarrow^* s'$ can have at most 1 step, so we conclude that ξ ensures self-stabilization in $\frac{n(n-1)}{2} + n(n-2) - 2 + 1 = \frac{1}{2}(3n+1)(n-2)$ steps. □

The original bound of [10] on the number of colours was $|C| \geq n$. The authors of [15] noticed that it can be lowered to $|C| \geq n-1$ and that it is optimal in the sense that for $|C| = n-2$ the claim of the theorem does not hold. The latter observation was established by noting that starting from the joint strategy

$$c_2 c_1 c_{n-2} \ldots c_2 c_1$$

the counterclockwise scheduling of the players combined with the selecting of the colours in the assumed cyclic order by player 1 generates an infinite improvement path which does not yield self-stabilization. The fact that self-stabilization can be reached in $\mathcal{O}(n^2)$ steps when $|C| \geq n$ was established in [22]. Finally, Theorem 5 shows that the use of a scheduler in Theorem 6 is necessary.

Next, we show that $\frac{1}{2}(3n+1)(n-2)$ is also a lower bound.

Example 1. Consider the game $\mathcal{G}$ for n players with $|C| \geq n-1$. Assume the cyclic order $c_1 \rightarrow c_2 \rightarrow \cdots \rightarrow c_{n-1} \rightarrow \ldots$ on C. So if $|C| = n-1$, then $c'_{n-1} = c_1$ and otherwise $c'_{n-1} = c_n$.

Then the following prefix of an improvement path is generated by every scheduler mentioned in Theorem 6 and ends in a legitimate joint strategy:

$$\begin{array}{rcll}
c_1 c_{n-1} c_{n-2} \ldots c_1 & \rightarrow & & \\
c_2 c_{n-1} c_{n-2} \ldots c_1 & \xrightarrow{n-1 \text{ steps}} & c_2 c_2 c_{n-1} c_{n-2} \ldots c_2 & \rightarrow \\
c_3 c_2 c_{n-1} c_{n-2} \ldots c_2 & \xrightarrow{n-1 \text{ steps}} & c_3 c_3 c_2 c_{n-1} \ldots c_3 & \rightarrow \\
c_4 c_3 c_2 c_{n-1} \ldots c_3 & \xrightarrow{n-1 \text{ steps}} & c_4 c_4 c_3 c_2 c_{n-1} \ldots c_4 & \rightarrow \\
c_5 \ldots c_2 c_{n-1} \ldots c_4 & \xrightarrow{n-1 \text{ steps}} & c_5 c_5 \ldots c_2 c_{n-1} \ldots c_5 & \rightarrow \\
 & \vdots & & \\
c_{n-1} \ldots c_2 c_{n-1} c_{n-2} & \xrightarrow{n-1 \text{ steps}} & c_{n-1} c_{n-1} \ldots c_2 c_{n-1} & \rightarrow \\
c'_{n-1} c_{n-1} \ldots c_2 c_{n-1} & \xrightarrow{\frac{n(n-1)}{2} - 2 \text{ steps}} & c'_{n-1} c'_{n-1} \ldots c'_{n-1} c_{n-1} c_{n-1}. &
\end{array}$$

The number of steps in the last line needs to be clarified since the scheduling used in the proof of Note 2 yields already after $\frac{n(n-1)}{2} - (n-1)$ steps the legitimate joint strategy $c'_{n-1}c_{n-1}\ldots c_{n-1}c_{n-1}c_{n-1}$, so 'too early'. Therefore we modify this scheduling to

$$(n), (n-1, n), (n-2, n-1, n), \ldots, (3, 4, \ldots, n-1), (2, 3, \ldots n-2, n, n-1, n).$$

This way we ensure that the legitimate joint strategy is reached only after $\frac{n(n-1)}{2} - 2$ steps. Alternatively, we could use the scheduling

$$\begin{gathered}(n, n-1, \ldots, 2), (n, n-1, \ldots, 3), \ldots, (n, n-1, n-2), (n, n-1), (n), \\ (n, n-1, \ldots, 2), (n, n-1, \ldots, 3), \ldots, (n, n-1, n-2), (n).\end{gathered}$$

The first and the last two lines consist in total of $1+\frac{n(n-1)}{2}-2$, so $\frac{(n+1)(n-2)}{2}$ steps, while each of the remaining $n-2$ lines consists of n steps. Therefore the total number of steps to reach $c'_{n-1}c'_{n-1}\ldots c'_{n-1}c_{n-1}c_{n-1}$ equals $\frac{(n+1)(n-2)}{2} + n(n-2) = \frac{1}{2}(3n+1)(n-2)$. Note that no other listed joint strategy is legitimate. □

7 Related Work and Discussion

Starting from [19], a paper that relates secret sharing and multiparty communication protocols to game theory, a growing literature keeps revealing rich connections between game theory and distributed computing. For a short overview of the early connections see Sect. 4 of [18].

Let us mention a couple of more recent examples. The authors of [1] provide a game-theoretic analysis of the leader election algorithms on a number of networks for both the synchronous case and the asynchronous case. In turn, [14] provides a framework in which the processes and the environment of a distributed system are viewed as players in an extensive game, in which implementations are interpreted as strategies with an implementation being correct if the corresponding strategy is winning.

To discuss the papers about connections between game theory and self-stabilization note first that we followed here the original Dijkstra's definition of a legitimate global state as the one in which exactly one machine can change its state. If we view a legitimate global state as the one in which no machine can change its state and drop the fairness assumption then we enter the area of *self-stabilizing algorithms*. An early example of such an algorithm is the one introduced in [27] that computes a maximal independent set (MIS).

Probably the first paper that noted the connection between the self-stabilizing algorithms and game theory is [6], where the notion of a *selfish stabilization* is introduced. The authors attached to each node of a graph a cost function (a customary alternative to the payoff functions in the definition of strategic games) to derive a simple self-stabilizing algorithm that constructs a spanning tree in a final state corresponding to a Nash equilibrium of the underlying strategic game. In turn, the authors of [20] related self-stabilization to *uncoupled dynamics*, a procedure used in game theory to reach a Nash equilibrium in situations when players do not know each others' payoff functions.

Recently, the authors of [29] observed that self-stabilizing algorithms that compute a maximal weighted independent set (MWIS) and MIS can be analyzed using game-theoretic tools. To relate this work to ours recall that in our setup we defined a legitimate joint strategy as the one in which exactly one player does not play a best response. Consider now an alternative definition that equates the legitimate joint strategy with a Nash equilibrium. We need now to recall the following definition due to [24]. We say that a strategic game has the ***finite improvement property*** (***FIP***) if every improvement path is finite.

The authors of [29] found that the self-stabilizing algorithms that compute a MWIS and a MIS correspond to natural strategic games on graphs that have the FIP. The computations of such an algorithm then correspond to the (necessarily finite) improvement paths in the corresponding game. They also noticed that if a game on a graph has the FIP then after an appropriate translation to a distributed system a self-stabilizing algorithm is obtained. Indeed, the FIP ensures the closure property, while the stability is immediate. These observations also clarify the set up of the just discussed papers [6,20].

We conclude this discussion of relations between self-stabilization and game theory by the following remark. The author of [17] introduced the concept of a ***weak self-stabilization*** which guarantees that a distributed system reaches a legitimate state only by *some* (and thus not necessarily all) sequence of moves. This concept can be easily incorporated into our framework by stipulating that a game ***admits weak self-stabilization*** if from every initial joint strategy some improvement path ensures self-stabilization. Schedulers that ensure self-stabilization obviously establish weak self-stabilization. This property naturally corresponds to the class of weakly acyclic games introduced in [23,30]. They are defined by the following weakening of the FIP: a game is ***weakly acyclic*** if for every initial joint strategy there exists a finite improvement path that starts in it. For a thorough analysis of weakly acyclic games see [3] from which we adopted the concept of a scheduler.

Acknowledgment. We acknowledge useful comments of Mohammad Izadi, Zoi Terzopoulou and Peter van Emde Boas. First author was partially supported by the NCN grant nr 2014/13/B/ST6/01807.

References

1. Abraham, I., Dolev, D., Halpern, J.Y.: Distributed protocols for leader election: a game-theoretic perspective. In: Afek, Y. (ed.) DISC 2013. LNCS, vol. 8205, pp. 61–75. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41527-2_5
2. Apt, K.R., de Keijzer, B., Rahn, M., Schäfer, G., Simon, S.: Coordination games on graphs. Int. J. Game Theory **46**(3), 851–877 (2017)
3. Apt, K.R., Simon, S.: A classification of weakly acyclic games. Theory Decis. **78**(4), 501–524 (2015)
4. Apt, K.R., Simon, S., Wojtczak, D.: Coordination games on directed graphs. In: Proceedings of the 15th Conference on Theoretical Aspects of Rationality and Knowledge (TARK 2015). EPTCS, vol. 215, pp. 67–80 (2016)

5. Arora, A., Gouda, M.: Closure and convergence: a foundation of fault-tolerant computing. IEEE Trans. Softw. Eng. **19**(11), 1015–1027 (1993)
6. Dasgupta, A., Ghosh, S., Tixeuil, S.: Selfish stabilization. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 231–243. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-540-49823-0_16
7. Dijkstra, E.W.: Self-stabilization in spite of distributed control, October 1973. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD391.PDF
8. Dijkstra, E.W.: Self-stabilization with four-state machines, October 1973. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD392.PDF
9. Dijkstra, E.W.: Self-stabilization with three-state machines, November 1973. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD396.PDF
10. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11), 643–644 (1974)
11. Dijkstra, E.W.: Self-stabilization in spite of distributed control. In: Selected Writings on Computing: A Personal Perspective, pp. 41–46. Springer, New York (1982). https://doi.org/10.1007/978-1-4612-5695-3_7
12. Dijkstra, E.W.: A belated proof of self-stabilization. Distrib. Comput. **1**(1), 5–6 (1986). https://www.cs.utexas.edu/users/EWD/ewd09xx/EWD922.PDF
13. Dolev, S.: Self-stabilization. MIT Press, Cambridge (2000)
14. Finkbeiner, B., Olderog, E.: Petri games: synthesis of distributed systems with causal memory. Inf. Comput. **253**, 181–203 (2017)
15. Fokkink, W., Hoepman, J., Pang, J.: A note on k-state self-stabilization in a ring with $k = n$. Nord. J. Comput. **12**(1), 18–26 (2005)
16. Ghosh, S.: An alternative solution to a problem on self-stabilization. ACM Trans. Program. Lang. Syst. **15**(4), 735–742 (1993)
17. Gouda, M.G.: The theory of weak stabilization. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 114–123. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45438-1_8
18. Halpern, J.Y.: Computer science and game theory: a brief survey. CoRR 2007 (2007). http://arxiv.org/abs/cs/0703148
19. Halpern, J.Y., Teague, V.: Rational secret sharing and multiparty computation: extended abstract. In: Proceedings of the 36th Annual ACM Symposium on Theory of Computing, pp. 623–632. ACM (2004)
20. Jaggard, A.D., Lutz, N., Schapira, M., Wright, R.N.: Self-stabilizing uncoupled dynamics. In: Lavi, R. (ed.) SAGT 2014. LNCS, vol. 8768, pp. 74–85. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44803-8_7
21. Lamport, L.: Solved problems, unsolved problems and non-problems in concurrency (invited address). In: Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, pp. 1–11 (1984)
22. Manku, G.S.: A simple proof for $\mathcal{O}(n^2)$ convergence of Dijkstra's self-stabilization protocol (2005, Unpublished)
23. Milchtaich, I.: Congestion games with player-specific payoff functions. Games Econ. Behav. **13**, 111–124 (1996)
24. Monderer, D., Shapley, L.S.: Potential games. Games Econ. Behav. **14**, 124–143 (1996)
25. Rahn, M., Schäfer, G.: Efficient equilibria in polymatrix coordination games. In: Italiano, G.F., Pighizzini, G., Sannella, D.T. (eds.) MFCS 2015. LNCS, vol. 9235, pp. 529–541. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48054-0_44
26. Schneider, M.: Self-stabilization. ACM Comput. Surv. **25**(1), 45–67 (1993)

27. Shukla, S.K., Rosenkrantz, D.J., Ravi, S.S., et al.: Observations on self-stabilizing graph algorithms for anonymous networks. In: Proceedings of the Second Workshop on Self-stabilizing Systems, vol. 7, p. 15 (1995)
28. Simon, S., Wojtczak, D.: Synchronisation games on hypergraphs. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2017), pp. 402–408. IJCAI/AAAI Press (2017)
29. Yen, L.-H., Huang, J.-Y., Turau, V.: Designing self-stabilizing systems using game theory. ACM Trans. Auton. Adapt. Syst. **11**(3), 18:1–18:27 (2016)
30. Young, H.P.: The evolution of conventions. Econometrica **61**(1), 57–84 (1993)

Energy-Utility Analysis of Probabilistic Systems with Exogenous Coordination

Christel Baier(✉), Philipp Chrszon(✉), Clemens Dubslaff(✉), Joachim Klein(✉), and Sascha Klüppelholz(✉)

Faculty of Computer Science, Technische Universität Dresden, Dresden, Germany
{christel.baier,philipp.chrszon,clemens.dubslaff,joachim.klein, sascha.klueppelholz}@tu-dresden.de

Abstract. We present an extension of the popular probabilistic model checker Prism with multi-actions that enables the modeling of complex coordination between stochastic components in an exogenous manner. This is supported by tooling that allows the use of the exogenous coordination language Reo for specifying the coordination glue code. The tool provides an automatic compilation feature for translating a Reo network of channels into Prism's guarded command language. Additionally, the tool supports the translation of reward monitoring components that can be attached to the Reo network to assign rewards or cost to activity within the coordination network. The semantics of the translated model is then based on weighted Markov decision processes that yield the basis, e.g., for a quantitative analysis using Prism. Feasibility of the approach is shown by a quantitative analysis of an energy-aware network system example modeled with a role-based modeling approach in Reo.

1 Introduction

In recent decades, many algorithms, logics and tools have been developed for the formal modeling and analysis of probabilistic systems, combining techniques introduced by the model-checking community with methods for the analysis of stochastic models (see, e.g., [12,16,22]). A widely used model is provided by Markov decision processes (MDPs), which represent probabilistic systems with non-determinism, suitable to model, e.g., concurrency, adversarial behavior or control. To allow for quantitative information attached to the states or transitions, MDPs are often augmented with rewards (sometimes also interpreted as costs). Rewards are useful, e.g., to reason about energy, waiting times or other costs, as well as utility, such as the number of successful completions of a task. Popular model checkers such as Prism [33,42] or Storm [18] can then be used to

The authors have been supported by the DFG through the Collaborative Research Center SFB 912 – HAEC, the Excellence Initiative by the German Federal and State Governments (cluster of excellence cfAED), the Research Training Group RoSI (GRK 1907), the DFG-projects BA-1679/11-1 and BA-1679/12-1, and the 5G Lab Germany.

F. de Boer et al. (Eds.): Arbab Festschrift, LNCS 10865, pp. 38–56, 2018.
https://doi.org/10.1007/978-3-319-90089-6_3

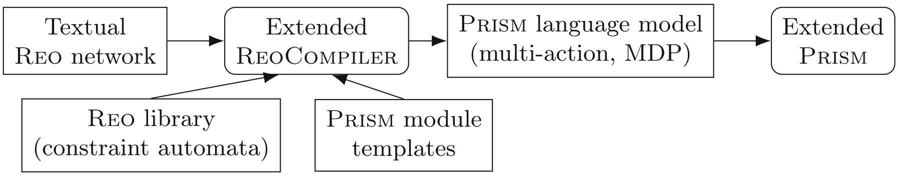


Fig. 1. Using the extended ReoCompiler to generate Prism language models

establish formal guarantees on the expected extremal (maximal/minimal) accumulated rewards and for the analysis of the trade-off between multiple accumulated rewards, e.g., comparing the required energy and the utility gained until reaching a goal for the various ways the non-determinism in the MDP can be resolved (see, e.g., [10,11,20,21]).

For modeling of stochastic systems, a common formalism is the Prism input language, a guarded-command language with probabilistic language features inspired by reactive modules [1]. It allows modeling a system by parallel composition of independent modules that can synchronize over shared actions and is particularly suitable for a symbolic encoding using, e.g., multi-terminal binary decision diagrams (MTBDDs) [42]. However, in practice, modeling complex coordination between the modules can be cumbersome and may require hard-coding the various synchronization possibilities in each module manually. It would therefore be desirable to model the coordination exogenously, i.e., the individual components of the system expose their willingness for synchronization via a well-defined interface to the outside, but do not need to be aware of the concrete connections to the other parts of the system. This facilitates a separation of concerns between computation and coordination, providing modeling flexibility and the ability to easily switch between coordination variants.

A preeminent advocate and example for this exogenous approach is the Reo language [2], a modeling formalism that allows for coordination patterns to be modeled compositionally as a network of channels. There are a wide variety of semantics for Reo [27] and, due to its generality, it can be useful in a wide range of contexts [3,5,8,28,30,31,45]. In the context of (non-probabilistic) model checking of systems described or coordinated by a Reo network, the operational semantics provided by constraint automata [14] proved to be versatile [8,30,31].

Contributions. We present an extension of the Prism input language and provide tool support that permits the use of multi-actions and suitable parallel composition operators that facilitate the exogenous modeling of coordination (Sect. 3). With an underlying MDP-based semantics, the parallel composition operators are derived from a data-abstract variant of *simple probabilistic constraint automata* (spCA) [7]. Here, probabilistic choice can influence the choice of successor state, but does not directly apply to the selection of enabled actions and is thus compatible with the MDP formalism.

Having provided the technical base for exogenous coordination, we are then interested to leverage Reo for the coordination of Prism modules. To achieve this, we have extended the ReoCompiler [47] with support for Prism as a new target language (Sect. 4). This enables the automatic generation of a Prism language model description from a textual description of a Reo network that coordinates Prism modules exogenously (see Fig. 1). To attach rewards to activity of the components and the network, we introduce the concept of reward monitors and provide tool support. This allows the quantitative analysis of the performance and of trade-offs for different scheduling and coordination strategies using Prism's variety of analysis backends (probabilistic model checking using explicit and symbolic engines as well as statistical model checking).

Our main focus is on the use of non-probabilistic Reo networks (with a constraint-automata-based operational semantics) for the coordination of probabilistic Prism modules. However, due to the compatibility with the spCA and MDP semantics, it is also possible to describe and use probabilistic channels by providing their operational behavior in the spCA semantics as Prism modules and incorporate those into a Reo network.

To demonstrate the feasibility of this exogenous modeling approach for the analysis of non-trivial stochastic systems, we consider a case study of a peer-to-peer network with compute nodes that can either play the role of a server, a client, or a relay in the computer network (Sect. 5). For this, we apply the role-based modeling approach using Reo as suggested in [17]. Role binding and role playing, as well as the communication protocol for the file transfer is constructed and coordinated via a network of Reo channels and connectors. We consider variants where the network topology is replaced and where a particular strategy is employed by switching to a different role-playing coordinator. We demonstrate the analysis of several queries that can be used to illuminate the trade-offs in the strategies. Our extensions of Prism and the ReoCompiler, as well as additional material is available at https://wwwtcs.inf.tu-dresden.de/ALGI/PUB/FA18.

Related Work. Apart from Reo, there is a variety of coordination languages, surveyed, e.g., in [41]. For our case study (Sect. 5), we rely on the models incorporating the concept of roles. Although roles are intuitive and commonly understood, there is no generally accepted definition of roles [48]. We follow the Dresden approach towards roles [32] and rely on our modeling framework for role-based systems using Reo presented in [17].

Several approaches extending Reo with stochastic component connectors have been presented in the literature, providing semantics in terms of simple probabilistic constraint automata [7], continuous-time constraint automata [15] quantitative intensional automata [4], stochastic Reo automata [38], stochastic timed automata for Reo [36], and probabilistic timed constraint automata [23], to mention a few. All these approaches above have in common that no direct tool support exists for these models and practical use is mainly justified by providing translations to continuous-time Markov chains (CTMCs) or interactive Markov chains (IMCs) [25]. For instance, case studies have been carried out in [4,38,39], based on IMC and CTMC representations of stochastic Reo automata and computing steady-state probabilities using Prism. In this line, REO2MC, a tool chain to automatically generate CTMC semantics from quantitative intensional automata was presented in [6]. Avoiding intermediate semantics, [40] presented a direct IMC semantics for stochastic REO and provides tool support using the model checkers CADP and IMCA. Using PRISM, they also performed quantitative analysis on CTMCs generated from the IMC semantics, including reward-based properties in the case study of [39].

Concerning modeling formalisms for stochastic systems, there is a variety of other approaches departing from the state-based models such as Markov chains or Markov decision processes we employ in this paper, e.g., stochastic Petri nets [37] or the stochastic process algebra PEPA [26].

2 Preliminaries

In this section we provide a brief overview to the PRISM input language, Markov decision processes (MDPs) as underlying semantics and the quantitative measures that can be addressed using probabilistic model checking. For details on PRISM we refer, e.g., to [42,43]. Details on MDPs and probabilistic model checking can, e.g., be found in [13,29,46] and the tutorial [19]. In the later sections of the paper we assume the reader to be familiar with the core concepts of REO. For further details we refer, e.g., to [2,14].

Markov Decision Processes. A *Markov decision process* (MDP) is a tuple $\mathcal{M} = (S, \mathfrak{Act}, P, \mathit{Rew})$ where S is a finite set of states, $\mathfrak{Act}$ a finite set of actions, $P\colon S \times \mathfrak{Act} \times S \to [0,1] \cap \mathbb{Q}$ is the transition probability function and Rew is a set of reward functions $\mathit{rew}_i\colon S \times \mathfrak{Act} \to \mathbb{N}$. We require that $\sum_{s' \in S} P(s, \alpha, s') \in \{0, 1\}$ for all $(s, \alpha) \in S \times \mathfrak{Act}$. We denote by $\mathfrak{Act}(s)$ the set of actions that are enabled in s, i.e., $\alpha \in \mathfrak{Act}(s)$ iff $P(s, \alpha, s') > 0$ for some $s' \in S$. The paths of $\mathcal{M}$ are finite or infinite sequences $s_0\, \alpha_0\, s_1\, \alpha_1\, s_2\, \alpha_2 \ldots$ where states and actions alternate such that $P(s_i, \alpha_i, s_{i+1}) > 0$ for all $i \geqslant 0$. Intuitively, in each step first the non-determinism between the enabled actions is resolved and then the successor state is chosen according to the probability distribution. If $\pi{=}s_0\, \alpha_0\, s_1\, \alpha_1\, s_2\, \alpha_2 \ldots \alpha_{k-1}\, s_k$ is a finite path, then $\mathit{rew}(\pi){=}\mathit{rew}(s_0, \alpha_0) + \mathit{rew}(s_1, \alpha_1) + \ldots + \mathit{rew}(s_{k-1}, \alpha_{k-1})$ denotes the accumulated reward along π. A *(randomized) scheduler* for $\mathcal{M}$, often also called policy or adversary, is a function σ that assigns to each finite path π a probability distribution over $\mathfrak{Act}(\mathit{last}(\pi))$ resolving the non-determinism in the MDP, where $\mathit{last}(\pi)$ is the last state of π.

The PRISM Input Language. We provide a brief, informal overview of the PRISM modeling language (which is also used by other tools and alternative model checkers such as STORM) and its MDP-based semantics. In particular, we concentrate on the features that are used for the synchronization of the individual modules, as our work presented in this paper extends them with features for multi-action synchronization.

A PRISM language model description generally consists of a set of *modules* $M_1, \ldots, M_n$. Each module can be seen as an independent process with local state

```
module foo
  s : [0..4] init 0;

  [act1] s = 0  →  1/2 : (s′ = 0)  +  1/2 : (s′ = 1);
  []     s < 4  →  1/4 : (s′ = 0)  +  3/4 : (s′ = s + 1);
endmodule
```

Fig. 2. A simple Prism module

variables, which can be either Boolean or can take values from a fixed integer range. The values of these state variables can only be updated from within the module, but can be read from other modules. Therefore, state variable names have to be unique across all the modules in the system. In addition to the local variables inside the modules, one can also declare global variables, which can be updated from any module with certain restrictions that ensure the absence of conflicting updates. In addition to modules, Prism allows defining reward structures that assign costs to either states or transitions.

The global state space of the composed MDP then consists of the Cartesian product of the local variables of all the modules, as well as of the global variables. Thus, each state in the MDP corresponds to a particular variable valuation. The step-wise behavior of a module M_i is specified by a set of *guarded commands* C_i, where each command c_j consists of an action a_j, a state guard g_j and an update specification u_j. The state guard, a Boolean expression over the variable valuations of all variables (global and local in any module), determines whether a command is *locally enabled* in a module. The update specification describes a probability distribution over the updates to the variable valuations. The action of a command allows for the synchronization between modules. A command becomes *globally enabled* only if all synchronization partners provide corresponding locally enabled commands. In standard Prism, the action consists of an action name or it can be left empty. The latter corresponds to an internal action that can happen at any time the state guard evaluates to `true`. Such actions never synchronize with other actions. Consider the example in Fig. 2. Here, the Prism module has a single variable `s` with possible values $0, \ldots, 4$ and two guarded commands. The first, with action `act1`, is enabled if variable $\mathtt{s} = 0$ and, upon execution, will set the value of variable `s` to either 0 or 1, each with probability $1/2$. The second guarded command specifies an internal action, which is enabled as long as $\mathtt{s} < 4$ and, upon execution, will reset the value of `s` to 0 with probability $1/4$ or increment the value of `s` by 1 with probability $3/4$. Each module M_i has an action alphabet Act_i, which consists of all the actions that are mentioned in the commands of module M_i.

The composed MDP arises from the parallel composition of the modules $M_1, \ldots, M_n$. Prism supports several process-algebra operators that allow fine-grained control over the order and synchronization type used in the parallel composition [43,44]. The parallel composition operator $M_1 \parallel M_2$, which is used by default, synchronizes commands in M_1 and M_2 that have actions which occur in both action alphabets Act_1 and Act_2. Thus, a command in M_1 with action $a \in Act_1 \cap Act_2$ is only enabled in some state in $M_1 \parallel M_2$ if there exists at least one command in M_2 with action a that is enabled as well. In this case, each enabled a-command in M_1 can be executed with each enabled a-command in M_2. On the other hand, if there is no enabled a-command in M_2, then none of the a-commands in M_1 are enabled. Those commands with actions outside of $Act_1 \cap Act_2$, as well as those without an action, can be executed only by themselves, i.e., in an interleaved manner. In addition to this default parallel composition operator, PRISM supports an operator $M_1 \,|||\, M_2$, which does not allow any synchronization and instead composes the commands of M_1 and M_2 in an interleaved manner, as well as a composition operator $M_1 \,|Act|\, M_2$ that allows for specifying the set of actions Act over which synchronization happens directly. Thus, $M_1 \parallel M_2$ is equivalent to $M_1 \,|Act_1 \cap Act_2|\, M_2$ and $M_1 \,|||\, M_2$ is equivalent to $M_1 \,|\varnothing|\, M_2$, i.e., using the empty set as the synchronizing alphabet. The action alphabet of the composition of M_1 and M_2 is obtained as the union of Act_1 and Act_2. Additionally, there is an operator that supports *hiding* of actions, i.e., turning some named actions into internal, empty actions and removing the actions from the action alphabet, as well as an operator for *renaming* actions.

Quantitative Analysis. Probabilistic model checkers such as PRISM and STORM can be used for the automated analysis of MDPs, for example answering questions such as "What is the maximal (minimal) probability for reaching some goal state, ranging over all schedulers?". Observing the rewards in the MDP, which can for example be used to model costs, energy, utility, etc., such tools also support a reward-based analysis, e.g., computing the maximal (minimal) expected accumulated reward until some goal is reached. Here, a trade-off analysis between multiple reward functions is of particular interest, for example using multi-objective analysis [20,21] or analysis of an energy-utility trade-off [10,11].

3 Exogenous Coordination with PRISM

We have extended PRISM's guarded command language with features that facilitate the modeling of more complex coordination schemes, in particular exogenous coordination. Most importantly, we have conservatively extended the PRISM language to support multi-actions. Although multi-actions arise rather naturally in REO connectors coordinating the activity and communication of components, till now there has been no support for in PRISM.

Extending the PRISM Language with Multi-actions. A command in our extension comprises a (possibly empty) *set* of actions α, a state guard, and an update specification. The actions α can either occur in a closed form, denoted by `[`α`]` or an open form, denoted by `]`α`[`. Intuitively, a closed multi-action indicates that no further action can be added during composition and yield a multi-action $\alpha' \subseteq \alpha$, while an open multi-action allows the composition with other actions to form a multi-action $\alpha' \supseteq \alpha$. Note that this extension is conservative in the sense that if α occurs only in closed form and contains at most one action, every

$$(1)\quad \frac{[\alpha_1] : g_1 \to u_1 \in C_1 \quad \wedge \quad [\alpha_2] : g_2 \to u_2 \in C_2 \quad \wedge \quad \alpha_1 = \alpha_2 \ \wedge \ \alpha_1 \cap Act \neq \varnothing}{[\alpha_1 \cup \alpha_2] : g_1 \wedge g_2 \to u_1 \cdot u_2 \in C_{1||2}}$$

$$(2a)\quad \frac{[\alpha_1] : g_1 \to u_1 \in C_1 \ \wedge \ \alpha_1 \cap Act = \varnothing}{[\alpha_1] : g_1 \to u_1 \in C_{1||2}} \qquad (2b)\quad \frac{[\alpha_2] : g_2 \to u_2 \in C_2 \ \wedge \ \alpha_2 \cap Act = \varnothing}{[\alpha_2] : g_2 \to u_2 \in C_{1||2}}$$

$$(3)\quad \frac{]\alpha_1[: g_1 \to u_1 \in C_1 \quad \wedge \quad]\alpha_2[: g_2 \to u_2 \in C_2 \quad \wedge \quad \alpha_1 \cap Act = \alpha_2 \cap Act}{]\alpha_1 \cup \alpha_2[: g_1 \wedge g_2 \to u_1 \cdot u_2 \in C_{1||2}}$$

$$(4a)\quad \frac{]\alpha_1[: g_1 \to u_1 \in C_1 \ \wedge \ \alpha_1 \cap Act = \varnothing}{]\alpha_1[: g_1 \to u_1 \in C_{1||2}} \qquad (4b)\quad \frac{]\alpha_2[: g_2 \to u_2 \in C_2 \ \wedge \ \alpha_2 \cap Act = \varnothing}{]\alpha_2[: g_2 \to u_2 \in C_{1||2}}$$

$$(5a)\quad \frac{]\alpha_1[: g_1 \to u_1 \in C_1 \ \wedge \ [\alpha_2] : g_2 \to u_2 \in C_2 \ \wedge \ \alpha_2 \neq \varnothing \ \wedge \ \alpha_1 = \alpha_2 \cap Act}{[\alpha_1 \cup \alpha_2] : g_1 \wedge g_2 \to u_1 \cdot u_2 \in C_{1||2}}$$

$$(5b)\quad \frac{[\alpha_1] : g_1 \to u_1 \in C_1 \ \wedge \ ]\alpha_2[: g_2 \to u_2 \in C_2 \ \wedge \ \alpha_1 \neq \varnothing \ \wedge \ \alpha_2 = \alpha_1 \cap Act}{[\alpha_1 \cup \alpha_2] : g_1 \wedge g_2 \to u_1 \cdot u_2 \in C_{1||2}}$$

Fig. 3. SOS rules for the parallel composition of the commands of two modules, synchronizing over the action alphabet *Act*.

command is as in standard Prism. As before, the action alphabet Act_i of module M_i is obtained from the set of actions that occur in any of M_i's commands.

Using the well-known SOS notation, we now provide the rules for the $M_1|Act|M_2$ parallel composition operator (see Fig. 3) that supports multi-actions. As noted above, $M_1 \parallel M_2$ can be obtained by using $Act = Act_1 \cap Act_2$ as the synchronization alphabet. In Fig. 3, we denote by $[\alpha] : g \to u \in C_i$ that there is a command in module M_i with closed multi-action α, state guard g and update specification u. Similarly, $]\alpha[: g \to u \in C_i$ denotes the same command albeit with open multi-action α. In the bottom part of the rules, $C_{1||2}$ stands for the commands in the composed module $M_1 \parallel M_2$. Furthermore, $u_1 \cdot u_2$ stands for the combined update specification obtained from u_1 and u_2 by using their product distribution, just as in the standard Prism semantics. For instance, the combined update specification $u_1 \cdot u_2$ for $u_1 = {}^1\!/_2 : (s'{=}0) + {}^1\!/_2 : (s'{=}1)$ and $u_2 = {}^1\!/_3 : (t'{=}0) + {}^2\!/_3 : (t'{=}1)$ would be

$${}^1\!/_6 : (s'{=}0, t'{=}0) + {}^2\!/_6 : (s'{=}1, t'{=}0) + {}^1\!/_6 : (s'{=}0, t'{=}1) + {}^2\!/_6 : (s'{=}1, t'{=}1).$$

We now provide some intuitive explanations for the composition rules. Rule (1) concerns the synchronization of two commands with closed multi-actions. As both are closed, it is not possible to add additional actions, which implies that $\alpha_1 = \alpha_2 = \alpha_1 \cup \alpha_2$. The condition $\alpha_1 \cap Act \neq \varnothing$ ensures that there is at least one action available for synchronization. All commands with closed actions that do not have any synchronizing action are handled by the symmetrical rules (2a) and (2b). This includes the handling of the closed empty multi-action, clearly excluded from the scope of rule (1). Altogether, rules (1), (2a) and (2b) collapse to the standard composition operator of Prism whenever the multi-actions are singletons or empty, thus preserving the standard Prism semantics whenever neither multi-actions nor open actions are used.

Rules (3), (4a) and (4b) deal with the composition of commands with open multi-actions. Rule (3) allows the parallel execution of two commands whenever their actions agree on the synchronized action alphabet. Note that there is no restriction on the non-emptiness of $\alpha_1 \cap Act$ and $\alpha_2 \cap Act$. Thus, two open commands that do not have actions in the synchronization alphabet and are therefore "unrelated" can be executed in parallel. Likewise, by rules (4a) and (4b), those actions can also be executed without synchronization.

Rules (5a) and (5b) deal with the parallel composition of open and closed commands. For (5a), the condition $\alpha_2 \neq \varnothing$ ensures that closed, empty actions never synchronize, while $\alpha_1 = \alpha_2 \cap Act$ ensures that $\alpha_1 \subseteq \alpha_2$, i.e., α_1 introduces no new actions, and that α_1 agrees with α_2 on the synchronizing actions. Rule (5b) is the symmetric rule to rule (5a).

The three rules (3), (4a) and (4b) correspond to the product rules for a data-abstract *simple probabilistic constraint automaton* as presented in [7]. The other rules in Fig. 3 can be similarly seen as variants of those product rules, adapted for closed commands and the mixture of closed and open commands in a natural, backward compatible fashion. Note that our parallel composition is commutative and associative, i.e., for modules M_1, M_2, and M_3 we have that the semantics of $M_1 \parallel M_2$ is isomorphic to the semantics of $M_2 \parallel M_1$ and likewise, the semantics of $M_1 \parallel (M_2 \parallel M_3)$ is isomorphic to the semantics of $(M_1 \parallel M_2) \parallel M_3$. The proof of this statement is straightforward but tedious and is provided in the extended version of this paper [9].

In the translation from the PRISM language model description to the underlying MDP, the set of actions in the MDP then corresponds to the powerset of action names that appear in the model description. That is, for an action alphabet Act of the composed system, the set of actions in the MDP is then $\mathfrak{Act} = 2^{Act}$, i.e., each action in the MDP is a subset of Act.

Reward structures in PRISM can be used to assign rewards to state-action pairs in the MDP, by declaring reward values for states satisfying a state guard and a specific, single action name. We have extended the declarations of reward structures with support for multi-action specifications, i.e., of the form $\texttt{[}\alpha\texttt{]}$ and $\texttt{]}\alpha\texttt{[}$ in the definitions of reward structures, where α is a set of actions from the action alphabet Act of the composed system. The reward value is then assigned to state-action pairs in the MDP with matching actions. A specification $\texttt{[}\alpha\texttt{]}$ matches exactly the action $\beta \in \mathfrak{Act}$ in the MDP iff $\alpha = \beta$. For $\texttt{]}\alpha\texttt{[}$, all actions $\beta \in \mathfrak{Act}$ that satisfy $\alpha \subseteq \beta$ match and are assigned the reward value. This can be used, e.g., to assign a reward whenever a particular action name is active, irregardless of which other actions in the system are active simultaneously.

Further Extensions of the PRISM Language. We have extended the PRISM language with additional features that simplify exogenous and compositional model design. PRISM supports a mechanism to take one module and obtain an additional instance. As the variable names and action names of a module live in the same namespace (even local variables of a module can be read from other modules), this requires renaming all module variables and possibly the action names in case synchronization between instances should be avoided. For example, the PRISM statement

```
module M2 = M1 [s1 = t1, s1 = t2, a1 = a2] endmodule
```

constructs module M2 as a copy of M1, renaming the state variables s1 and s2 to t1 and t2, respectively, as well as renaming action a1 to a2. As this kind of statement requires detailed knowledge of the variable names for the module, we have extended the syntax to allow for *rule-based renaming.* For example, the statement

```
module M2 = M1 (varprefix = M1_)[a1 = a2] endmodule
```

would automatically rename a variable s in M1 to M1_s, which makes it easy to ensure global uniqueness of variable names. Additionally, we support rules with varsuffix (adding a given suffix to the variable names), actionprefix and actionsuffix (similarly renaming the individual action names occurring in a module). Rule-based renaming is being performed first, then the additional, explicitly given, renamings are performed. Note that, however, every state variable and action can only be renamed once. With this automatic renaming, Prism's module renaming statement can be seen as the *instantiation* of a module. However, in standard Prism, every module definition that appears in the input file is automatically instantiated. This makes it impossible to provide a library of module templates, of which only a subset is actually instantiated. To remedy this issue, we allow a module definition to be marked as *template.* Such a template module will not be instantiated automatically, but is available for instantiation via module renaming.

A Simple Example of Exogenous Coordination. As an example for exogenous coordination, consider a simple setting with three producer modules and three consumer modules. Each of them has a certain probability in each step to become broken. In each step, exactly one of the non-broken producers shall synchronize with one of the non-broken consumers, until eventually almost surely all have failed. In the standard Prism language, we have to hard-code one command for each synchronization choice in each module, e.g., by using actions $\mathtt{p}_\mathtt{i}\mathtt{c}_\mathtt{j}$ to synchronize producer i with consumer j. With our extension of Prism, we can model exogenous coordination: Each producer and consumer module has a single action, which is suitably synchronized by some glue code modules, e.g., a merger module that nondeterministically selects one of the producers and is chained to a router module that nondeterministically selects one of the consumers. In the extended version of this paper [9], we provide a detailed description of both approaches. It is readily apparent that the second, exogenous approach provides far greater flexibility and separation of concerns, making it easy to replace the coordination glue code by alternative variants, specializations, etc.

Prism Implementation. We have extended Prism with support for handling multi-actions and for dealing with the other proposed language extensions, both in Prism's explicit engine (where an explicit, graph-based model representation is built) and in the (semi-)symbolic engines (where a symbolic model representation [42] is used). For the explicit engine, this mostly consists of the handling of the parallel composition according to the rules of Fig. 3 during model construction. For a given variable valuation, we can easily determine the commands that are locally enabled in each module of the system. Then, we have to determine the possibilities for synchronized and independent execution of commands from different modules, maintaining data structures to speed-up the lookup of potential synchronizing commands.

For the symbolic engine of Prism, which is based on a symbolic representation of the model via *multi-terminal binary decision diagrams* (MTBDDs) [42], the transition structure of an MDP is encoded using MTBDD variables for the nondeterministic choices, as well as variables for encoding the states and successor states, mapping to the probability for a given transition in the MDP. As Prism already encodes each action name in the model description by one variable, adapting the encoding to multi-actions is rather straightforward. Likewise, the various composition rules in Fig. 3 can be elegantly formulated as symbolic operations on the MTBDD representation for each module. One complication however is the encoding of local nondeterminism within a module, i.e., to distinguish which of multiple commands with the same multi-action that are enabled simultaneously are actually executed. The encoding used by standard Prism (a binary encoding of an integer for the various local nondeterministic choices) is not convenient for a fully symbolic composition, therefore we changed this encoding. In our extension, each command in the model corresponds to an MTBDD variable that denotes whether this command is actually active or inactive in a given step.

4 Reo for Exogenous Coordination within Prism

Having extended Prism by the infrastructure for the exogenous coordination of probabilistic components, we are now interested in a framework for the convenient modeling of the coordination glue code. For this, the channel-based coordination language Reo [2] provides an elegant and compositional modeling approach, where the coordination glue code for components is specified using a Reo network of channels that can be used to model a plethora of coordination patterns. For this, both stateless channels such as synchronous channels (ensuring that activity at their channel ends happens simultaneously), asynchronous channels (ensuring the non-synchronicity/mutual exclusion at their channel ends), lossy channels or transformer channels, as well as stateful ones such as FIFO channels (which can accept a token or data and pass it on later) are used, mediated by network nodes that coordinate the activity of the connected channels. Additionally, ready-made or user-defined circuits can be used as building blocks to model common coordination patterns, such as a sequencer that ensures that certain activity happens one after the other. With constraint-automata-based operational semantics [7,14] for Reo, the behavior of the whole network can be obtained from the automata-based descriptions of the individual parts (channels and nodes) in a compositional manner by a series of product operations.

To allow the use of Reo as the coordination glue code of Prism components, we make use of the ReoCompiler tool developed at the Centrum Wiskunde

& Informatica, Amsterdam [47]. Among others, the ReoCompiler supports the convenient textual specification of Reo networks, providing the glue code for components. Then, it allows the compilation of the glue code to a target language (such as Java). When combined with definitions of the components in the target language (e.g., a Java class implementing the component's behavior), the coordinated system can then be executed. The external components interface with the Reo network via input and output ports.

Prism as a Target Language of the ReoCompiler. We have extended the existing ReoCompiler with support for the Prism language. In particular, we provide a translation from the constraint-automata-like intermediate compilation result for the glue code to the Prism language. This relies on the extension for multi-actions and the product operator for modules presented in Sect. 3, which allows the encoding of the operational semantics via (data-abstract) simple probabilistic constraint automata [7]. Together with the flexible module instantiation from module templates, the generated Prism language model description properly instantiates the various (Prism-based) components and connects with the generated coordination glue code. Here, Prism's components actions are exported to the network as input/output ports.

The constraint-automata semantics for the Reo channels supports the transfer of data, i.e., ports or nodes in the network are not only active or not, but may have some observable data value. As we are mainly interested in the data-abstract coordination of Prism components, i.e., an action either fires or not but carries no data, we treat the Reo network as using a singleton data domain. As sometimes attaching data to actions is natural for certain modeling tasks, we however provide basic tooling as well to emulate actions carrying data by encoding the different data values as variants of the actions, i.e., for an action $\mathtt{a}$ there are variants $\mathtt{a}_1$, $\mathtt{a}_2, \ldots$ corresponding to the data values $1, 2, \ldots$.

We support two orthogonal approaches to the compilation. In the first, *monolithic* approach, the whole Reo network comprising the glue code, i.e., all parts of the network except for the "native" Prism components are compiled into a single protocol module. This compilation relies on the composition of all the channels and nodes within the ReoCompiler. In a second, *compositional* approach, the ReoCompiler is used to generate a Prism language file where all the individual channels and nodes of the Reo network are translated to individual Prism modules and where the composition of the behavior is performed during Prism's translation from the model description to the concrete MDP. Here, we crucially rely on the fact that we can readily translate the ReoCompiler's internal representation of Reo networks into a Prism module. It should be noted that both approaches have a minor difference in the underlying semantics: The composition inside the ReoCompiler relies on classical interleaving for independent (unsynchronized) parts of the Reo network. Then, the generated code realizes a sequential implementation that simulates the parallel execution of these independent parts. In the compositional approach, unrelated actions can synchronize (cf. rule (3) in Fig. 3, with $Act = \varnothing$) and thus are executed in parallel. This can, e.g., be observed for chains of FIFO channels. The monolithic approach can be useful to hide the internal complexity of a Reo network from Prism, while the compositional approach provides more insight into the parts and the structure of the Reo network at the level of the model checker.

Further Extensions to the ReoCompiler. We have extended the ReoCompiler with some other features that are useful in the context of the quantitative analysis of the generated models. First, towards model checking it is often required to refer to the content of a state variable, e.g., for one of the Prism components or the state of a FIFO channel in the Reo network. As the focus of the ReoCompiler is more on generating executable code where the component names are largely irrelevant, it only generates unique, but not necessarily stable names. We have added syntax and support for providing a name during component instantiation, which results in a predictable name for the state variables of the component instances (and memory cells for stateful channels) in the generated Prism language file.

Another common requirement in probabilistic model checking is the ability to assign *rewards* or *costs* to the model, for example to model the energy consumption or to track the achieved utility on completion of a task. In Prism, such rewards can be attached to states (e.g., for every step spent in a state, a certain reward is accumulated) as well to transitions (e.g., for a step with certain actions, a given reward is accumulated). As the names of the actions (i.e., ports and nodes) generated during the network composition process are not necessarily stable or predictable, e.g., due to the application of the Reo hiding operator, we have extended ReoCompiler with support for *reward monitors*. Here, a reward monitor is a special component with a given set of input ports which can be attached to the network using the standard Reo channels and operations. The reward-monitor definition then specifies the rewards that are assigned whenever certain of the input ports of the monitor are active. We support two variants, local and global monitors. A local monitor tracks a reward on its own, resulting in a single reward structure in the generated Prism file. In contrast, a global monitor carries a label that ensures that, if there are multiple monitors with the same label, the reward from all of those monitors is collected in a single Prism reward structure. This allows, e.g., attaching a dedicated reward monitor to each component that records the energy consumption when there is port activity, with all those rewards being added together to yield the overall energy consumption of the system in each step.

Additionally, we have added the ability to include Prism language snippets from external files into the generated Prism language file, allowing the convenient inclusion of the module templates for the Prism components that may be instantiated in the generated model description, as well as auxiliary definitions that commonly arise during the modeling with Prism, such as constant definitions, the definition of state labels as well as additional reward structures.

Example. In the extended version of this paper [9], we provide a detailed description how the coordination glue code in the producer/consumer example can be elegantly modeled using a Reo network, with automatic generation of the corresponding Prism language model description via our extended version of the ReoCompiler. Here, we can model the coordination using Reo channels and an exclusive router, which provides the desired coordination in a compositional manner.

Other Model Semantics. In addition to the MDP semantics, it is also possible to generate a model description for a discrete-time Markov chain (DTMC). Here, Prism resolves all nondeterminism in the MDP model uniformly. Moreover, it is possible as well to select probabilistic timed automata (PTA) semantics [35], where the Prism modules may additionally contain special clock variables, clock invariants and the commands can contain clock guards and clock resets. For these PTA models, our extension for multi-actions and the related composition operators supports the analysis via Prism's digital clock engine, which internally transforms the PTA into an MDP [34]. The adaption of Prism's other PTA analysis engines remains part of future work.

5 Application: Energy-Aware Network System

In this section, we present a peer-to-peer file transfer case study that leverages the extensions to Prism and the Reo tool support to model the complex coordination between the components of the system. The model is inspired by a case study presented in [24]. The network system consists of several *stations* or nodes interconnected via a network with some fixed topology, e.g. a ring or star topology. Each of the stations can store files. We do not consider the actual contents of these files in our model and rather represent them using an abstract index. A station may request a file from another station connected to the network. Then, the file is transfered between the stations using a peer-to-peer approach, i.e., without a central entity handling the transfer.

In a file transfer, each participating station may act in one of three different *roles*. The station that initiated the request and will receive the file plays the role of the *client*. Conversely, a station that has a local copy of the requested file can act as a *server*. Since a file transfer can also happen between stations that are not directly connected, but via one or more hops, the stations in between client and server play the role of a *relay*. A relay station retransmits incoming requests and file data to its neighboring stations. As a file transfer may be initiated between any two stations on the network, each station may dynamically play one of the three roles: server, client or relay.

To model such a system, we employed the role-based modeling approach proposed in [17]. Within this approach, the dynamically changing behaviors, i.e., the roles, are separated from the static core functionality. The main idea is to encapsulate the role behaviors into *role components*. These role components are then bound to their player using a Reo connector. This *binding connector* enables the player component to dynamically enact the role behavior.

Figure 4 depicts the binding connector between a station and its roles in detail. Here, $\bullet$ denotes standard Reo nodes with nondeterministic merging on the input side and replication (simultaneous activity) on the output side, while $\otimes$ denotes an x-router node, where there is a nondeterministic choice on the

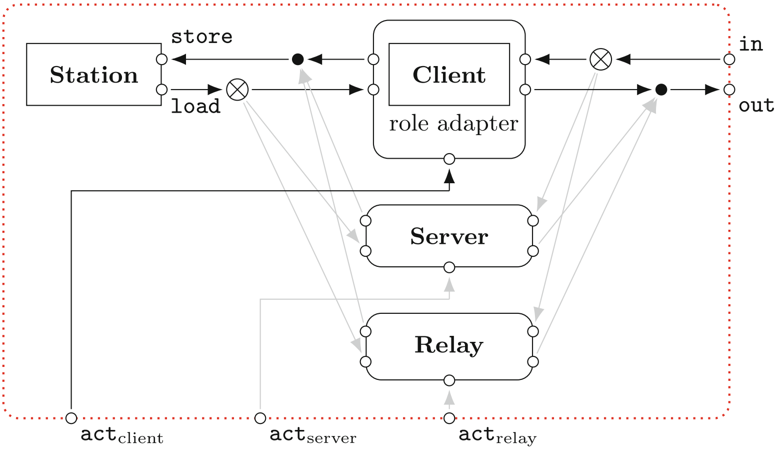


Fig. 4. A station component and its bound role components

output side. The station component as well as the role components are modeled as Prism modules. Each of the role components is wrapped in a role adapter (shown in detail for the client role). This adapter adds one port that allows enabling or disabling the role. Internally, this port is synchronized with the `in` and `out` ports of the role component, thus blocking the `act` port will effectively disable the role. The connector between the station component and the role components ensures that each role can retrieve the file stored on the local station or replace it with another one. The other part of the connector attaches the roles to the network, allowing each of them to send or receive requests or file data. We use the same binding connector for all stations within the network. The network itself is also realized as a Reo network which connects the `in` and `out` ports of the stations according to the network topology.

The `act` ports of a station's roles allow the *role-playing coordinator* to enable and disable role behaviors dynamically. The role-playing coordinator enforces that all stations act according to the peer-to-peer file-transfer protocol. The core component of a station can generate a request for a certain file according to a probabilistic distribution. This request is buffered by the coordinator. Eventually, the coordinator allows the station to play the client role to send the request into the network. Another station will then receive this request. In case this station has the requested file, the server role will be enabled, which in turn fetches the file and sends it back. If the station does not have the file, the relay role will be played and the request is sent to the neighboring stations. For simplicity, the global coordination will also ensure that only one file transfer happens simultaneously.

Our approach allows us to vary the coordination without modifying the Prism modules "implementing" the station core component and the role components. The connector modeling the network can be changed to different topologies. The remaining nondeterminism in the system stands for the different strategies that may be employed to achieve certain objectives. This concerns, e.g., the role-playing assignments for the different stations and the choice which of the pending requests will be processed next. By attaching different coordination, we can thus explore the effect of a particular strategy, for example replacing the nondeterministic choice of the next request by a uniform random choice. The coordinator could also be augmented to ensure that no file is "forgotten" by the network by blocking requests that would overwrite the last copy of a file.

We have analyzed the file-transfer model with three stations. In particular, we have considered four variants of the model by using a ring or chain topology of the network connector and by either using a nondeterministic choice or a probabilistic choice of the next request to be processed.[1] For the analysis, we have added reward monitors and state rewards to the model. *Energy* is consumed on network activity, i.e., whenever one or more of the `in` and `out` ports of the network connector are active. Furthermore, a *penalty* (negative utility) is associated with pending requests that have not yet been processed. We have used Prism to analyse the model variants, among others asking for

(a) the minimal/maximal probability that eventually station 1 receives its requested file,
(b) the minimal/maximal probability that eventually all stations have a file,
(c) the minimal expected time until the file requested by station 1 is delivered,
(d) the minimal expected time until all stations have received a file,
(e) the maximal probability to deliver a file to station 1 with less than x penalty,
(f) the maximal probability for delivering a file using a given energy budget without overstepping the penalty threshold, and
(g) the minimal energy required such that a file is delivered to station 1 with a probability greater than 0.9 without a penalty violation.

The analysis results for the queries (c) to (g) are presented in Table 1. The results for (c) show that in a ring topology, the requested file is delivered faster than in the chain topology. This is as expected, since in the chain topology, we always need one hop to transfer a file between the two outer stations of the network, while in the ring topology a direct transfer without hops is always possible. The same argument also applies to (d). The results for (e) show the difference between the random scheduling and the optimal scheduling of the next file transfer. Generally, the random scheduling collects a higher penalty, which means that pending transfers are kept waiting longer. The reward-bounded reachability probability (f) and the quantile [10] query (g) illuminate the trade-off between early processing of a request and thus consuming less energy, or waiting for another request to arrive thereby collecting a penalty for pending requests. Comparing the minimal energy consumption in (g) for the nondeterministic and random selection of the requests, we see that the nondeterministic choice uses less energy. This is as expected, because the nondeterministic selection corresponds to the optimal strategy for choosing the next request.

[1] For further details on the models and experiments, see https://wwwtcs.inf.tu-dresden.de/ALGI/PUB/FA18.

Table 1. Analysis results for the file-transfer model with 3 stations

Variant	(c)	(d)	(e)	(f)	(g)
chain, nondet	4.0	16.15	0.95	0.98	10.0
ring, nondet	2.0	15.94	1.0	0.96	12.0
chain, random	5.87	18.73	0.64	0.65	15.0
ring, random	4.0	18.34	0.78	0.72	12.0

Table 2. Model sizes and analysis times for the file-transfer model with 3 stations

Variant	States	Components	Actions	Time (s)		
				Build	Analysis (f)	Analysis (g)
chain, nondet	4204	108	150	10.7	81.1	58.4
ring, nondet	34164	112	150	90.7	197.8	91.3
chain, random	12612	103	154	10.6	92.0	24.4
ring, random	102492	107	154	62.6	224.5	101.0

Model sizes and the time required for model construction and analysis of instances of queries (f) as well as (g) are presented in Table 2. The number of components consists of the number of channels, Prism modules and Reo nodes in the network. The actions column refers to the number of unique action names within the generated model. Here, the analysis has been carried out using the symbolic engine of Prism and the monolithic approach. The considerable number of components and states is caused by the detailed modeling of the role-playing coordinator. The coordinator divides a file transfer into multiple steps which requires storing request messages in its internal state. The number of states within the random variants is greater than in the nondeterministic variants because the random selection of requests requires additional internal state compared to a nondeterministic selection. The ring topology further increases the number of states since more routes within the network are possible.

6 Conclusions

We have extended the Prism language and the Prism model checker by features that allow an exogenous modeling of the coordination of Prism modules. We believe that, already on its own, these modeling capabilities will be very useful for the modeling of complex case studies. By using our extension of the ReoCompiler, this exogenous approach can additionally leverage the elegant specification of complex coordination patterns by Reo networks and allow the creation of model variants, as seen in our case study.

As future work, we are interested in exploring the full integration of actions with attached data values into Prism. Previous experience with the symbolic encoding of models with data [8] suggest that this would require some effort to ensure a compact symbolic encoding, which is compounded by the fact that good heuristics for the variable ordering from the non-probabilistic setting, such as interleaving the data on the actions with related state variables, may conflict with variable-ordering restrictions designed for efficient probabilistic model checking.

We are also interested in ways to provide the user more feedback during modeling, e.g., by integrating a visualization of the Reo network with animated control flow into Prism's simulation view, which can also be used to explore counter-examples from Prism's non-probabilistic CTL and LTL checkers.

References

1. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods Syst. Des. **15**(1), 7–48 (1999)
2. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**, 329–366 (2004)
3. Arbab, F., Aştefănoaei, L., de Boer, F.S., Dastani, M., Meyer, J.-J., Tinnermeier, N.: Reo connectors as coordination artifacts in 2APL systems. In: Bui, T.D., Ho, T.V., Ha, Q.T. (eds.) PRIMA 2008. LNCS (LNAI), vol. 5357, pp. 42–53. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89674-6_8
4. Arbab, F., Chothia, T., van der Mei, R., Meng, S., Moon, Y.J., Verhoef, C.: From coordination to stochastic models of QoS. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 268–287. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02053-7_14
5. Arbab, F., Kokash, N., Meng, S.: Towards using Reo for compliance-aware business process modeling. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. CCIS, vol. 17, pp. 108–123. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88479-8_9
6. Arbab, F., Meng, S., Moon, Y., Kwiatkowska, M.Z., Qu, H.: Reo2MC: a tool chain for performance analysis of coordination models. In: ESEC/SIGSOFT FSE 2009, pp. 287–288. ACM (2009)
7. Baier, C.: Probabilistic models for Reo connector circuits. J. Univ. Comput. Sci. **11**(10), 1718–1748 (2005)
8. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: Formal verification for components and connectors. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 82–101. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04167-9_5
9. Baier, C., Chrszon, P., Dubslaff, C., Klein, J., Klüppelholz, S.: Energy-utility analysis of probabilistic systems with exogenous coordination (extended version) (2018). http://wwwtcs.inf.tu-dresden.de/ALGI/PUB/FA18/
10. Baier, C., Daum, M., Dubslaff, C., Klein, J., Klüppelholz, S.: Energy-utility quantiles. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 285–299. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06200-6_24
11. Baier, C., Dubslaff, C., Klein, J., Klüppelholz, S., Wunderlich, S.: Probabilistic model checking for energy-utility analysis. In: van Breugel, F., Kashefi, E., Palamidessi, C., Rutten, J. (eds.) Horizons of the Mind. A Tribute to Prakash Panangaden. LNCS, vol. 8464, pp. 96–123. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06880-0_5
12. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.: Model-checking algorithms for continuous-time Markov chains. IEEE Trans. Softw. Eng. **29**(6), 524–541 (2003)

13. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
14. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. **61**(2), 75–113 (2006)
15. Baier, C., Wolf, V.: Stochastic reasoning about channel-based component connectors. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 1–15. Springer, Heidelberg (2006). https://doi.org/10.1007/11767954_1
16. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60692-0_70
17. Chrszon, P., Dubslaff, C., Baier, C., Klein, J., Klüppelholz, S.: Modeling role-based systems with exogenous coordination. In: Ábrahám, E., Bonsangue, M., Johnsen, E.B. (eds.) Theory and Practice of Formal Methods. LNCS, vol. 9660, pp. 122–139. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30734-3_10
18. Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A Storm is coming: a modern probabilistic model checker. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 592–600. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_31
19. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D.: Automated verification techniques for probabilistic systems. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 53–113. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21455-4_3
20. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Quantitative multi-objective verification for probabilistic systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 112–127. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_11
21. Forejt, V., Kwiatkowska, M., Parker, D.: Pareto curves for probabilistic model checking. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, pp. 317–332. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33386-6_25
22. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects Comput. **6**, 512–535 (1994)
23. He, K., Hermanns, H., Chen, Y.: Models of connected things: on priced probabilistic timed Reo. In: 41st IEEE Annual Computer Software and Applications Conference (COMPSAC 2017), vol. 1, pp. 234–243 (2017)
24. Hennicker, R., Klarl, A.: Foundations for ensemble modeling – the Helena approach. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. LNCS, vol. 8373, pp. 359–381. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54624-2_18
25. Hermanns, H.: Interactive Markov chains. Ph.D. thesis, University of Erlangen-Nuremberg, Germany (1999)
26. Hillston, J.: A compositional approach to performance modelling. Ph.D. thesis, University of Edinburgh, UK (1994)
27. Jongmans, S.T.Q., Arbab, F.: Overview of thirty semantic formalisms for Reo. Sci. Ann. Comput. Sci. **22**(1), 201–251 (2012)
28. Jongmans, S.-S.T.Q., Santini, F., Sargolzaei, M., Arbab, F., Afsarmanesh, H.: Automatic code generation for the orchestration of web services with Reo. In: De Paoli, F., Pimentel, E., Zavattaro, G. (eds.) ESOCC 2012. LNCS, vol. 7592, pp. 1–16. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33427-6_1
29. Kallenberg, L.: Markov Decision Processes. Lecture Notes, University of Leiden (2011)

30. Kokash, N., Arbab, F.: Formal design and verification of long-running transactions with extensible coordination tools. IEEE Trans. Serv. Comput. **6**(2), 186–200 (2013)
31. Kokash, N., Krause, C., de Vink, E.: Reo + `mCRL2`: a framework for model-checking dataflow in service compositions. Formal Aspects Comput. **24**(2), 187–216 (2012)
32. Kühn, T., Leuthäuser, M., Götz, S., Seidl, C., Aßmann, U.: A metamodel family for role-based modeling and programming languages. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 141–160. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11245-9_8
33. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
34. Kwiatkowska, M.Z., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. Formal Methods Syst. Des. **29**(1), 33–78 (2006)
35. Kwiatkowska, M.Z., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theoret. Comput. Sci. **282**(1), 101–150 (2002)
36. Li, Y., Zhang, X., Ji, Y., Sun, M.: Capturing stochastic and real-time behavior in Reo connectors. In: Cavalheiro, S., Fiadeiro, J. (eds.) SBMF 2017. LNCS, vol. 10623, pp. 287–304. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70848-5_18
37. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with generalized stochastic Petri nets. SIGMETRICS Perform. Eval. Rev. **26**(2), 2 (1998)
38. Moon, Y., Silva, A., Krause, C., Arbab, F.: A compositional model to reason about end-to-end QoS in stochastic Reo connectors. Sci. Comput. Program. **80**, 3–24 (2014)
39. Moon, Y.-J., Arbab, F., Silva, A., Stam, A., Verhoef, C.: Stochastic Reo: a case study. In: Workshop on Harnessing Theories for Tool Support in Software (TTSS 2011), pp. 90–105 (2011)
40. Oliveira, N., Silva, A., Barbosa, L.S.: Imc$_{\text{reo}}$: interactive Markov chains for stochastic Reo. J. Internet Serv. Inf. Secur. **5**(1), 3–28 (2015)
41. Papadopoulos, G.A., Arbab, F.: Coordination models and languages. Adv. Comput. **46**, 329–400 (1998)
42. Parker, D.: Implementation of symbolic model checking for probabilistic systems. Ph.D. thesis, University of Birmingham (2002)
43. PRISM Manual. http://www.prismmodelchecker.org/manual/
44. PRISM Language Semantics. http://www.prismmodelchecker.org/doc/semantics.pdf
45. Proença, J.: Synchronous coordination of distributed components. Ph.D. thesis, Leiden University (2011)
46. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, New York (1994)
47. The Reo Compiler. https://github.com/ReoLanguage/Reo
48. Zhu, H., Zhou, M.: Roles in information systems: a survey. IEEE Trans. Syst. Man Cybern. Part C **38**(3), 377–396 (2008)

A Note on Reactive Transitions and Reo Connectors

Daniel Figueiredo[1], Manuel A. Martins[1], and Luís S. Barbosa[2,3](✉)

[1] CIDMA—Universidade de Aveiro, Aveiro, Portugal
{daniel.figueiredo,martins}@ua.pt
[2] HASLab—INESC TEC and QUANTALab,
Universidade do Minho, Braga, Portugal
lsb@di.uminho.pt
[3] UNU-EGOV, United Nations University, Guimarães, Portugal

Abstract. The structure of a reactive transition system can to be modified on the fly by e.g. removing, reversing or adding new transitions. The topic has been studied by D. Gabbay and his collaborators in different contexts. In this paper we take their work a step further, introducing a suitable notion of bisimulation and obtaining a Hennessy-Milner theorem with respect to a hybrid logic in which transition properties can be expressed. Our motivation is to provide a characterisation of equivalence for such systems in order to exploit their possible roles in the formal description of software connectors in Reo, either from a behavioural (semantic) or spatial (syntactic) point of view.

1 Introduction

Complex, distributed systems require reliable and yet flexible architectures. A clear separation between typical *loci* of computation (e.g. services or components) and the protocols that manage their interaction is at the heart of the so-called *exogenous coordination* models and, in particular, of Farhad Arbab's outstanding contributions to Software Engineering [1,2]. Actually, Reo connectors mediate interaction, offering a powerful "glue-code" to express such protocols, while maintaining the envisaged separation of concerns. Moreover, Reo connectors are compositional, providing a very flexible approach to software composition. Among languages with a similar purpose, Reo is the only one that allows for propagation of mutual exclusion and synchrony requirements along the connector structure.

Different forms of transition systems have been used as semantic domains for connector behaviour [14], either directly (as in, e.g. constraint [5] or Reo automata [8]), or indirectly through mappings to process algebra formalisms [6,15]. Typically, such systems are then regarded as (variants of) Kripke frames

This paper is dedicated to Farhad Arbab, on the occasion of his retirement, as a tribute of gratitude for his outstanding contributions to the field of systems' interaction and composition, his inspiring attitude, and generosity.

F. de Boer et al. (Eds.): Arbab Festschrift, LNCS 10865, pp. 57–67, 2018.
https://doi.org/10.1007/978-3-319-90089-6_4

upon which (variants of) modal logics are interpreted providing a framework to reason about coordination semantics. This is well-known and will not be further detailed here.

From a different point of view, Reo connectors are syntactically represented as graphs of communication primitives (e.g. channels) whose nodes stand for interaction points. Edges are labelled with channel identifiers and types which classify their behaviour. Again such graphs can be regarded as Kripke frames expressing the spatial structure of coordination patterns. This perspective was introduced by Oliveira and Barbosa [17] when proposing an elementary framework for expressing reconfigurations of the interaction protocols, i.e. of the connector's structure, as discussed, for instance, by Krause [16]. Reconfigurations in that sense may substitute, add or remove communication channels, or move communication interfaces between components, in order to restructure a complex interaction policy. The corresponding modal logic expresses properties of (spatial) connector structure.

From this perspective, the focus is placed on the interconnection structure, with no reference to the connector's emerging behaviour. Examples of structural, or 'syntactic' properties are:

(i) every $\mathsf{fifo_e}$ channel from a node n is connected to at least a lossy channel,
(ii) node i is an output node of the connector.

In [17] it is required that a reconfiguration preserves such properties.

Often structural properties are to be formulated relatively to a particular node in the pattern. An example is given by property *(ii)* above. In general, one may require, for instance, that all the channels incident to a specific node and their interconnections remain unchanged under a reconfiguration. This justifies the choice of hybrid logic [7,9] to express such properties. In general, hybrid logic adds to a modal language the ability to name, or to explicitly refer to specific states of the underlying Kripke structure. This is done through the introduction of propositional symbols of a new sort, called nominals, each of which is true at exactly one possible state. The sentences are then enriched in two directions. On the one hand, nominals are used as simple sentences holding exclusively in the state they name. On the other hand, explicit reference to states is provided by a satisfaction operator @ such that $@_i\phi$ asserts the validity of ϕ at the state named i. In the logic described in next section, properties (i) and (ii) above are written[1] as

$$@_n[\mathsf{fifo_e}]\langle\mathsf{lossy}\rangle\top$$

and

$$@_i[-]\bot$$

respectively, where $[-]\bot$ states the absence of outgoing channels from the node referred by nominal i.

[1] As used in modal logics coming from process algebras, modalities are indexed by sets of labels, with symbol "-" standing for the whole set of those.

The starting point for this paper is that the description of coordination elements at any level (*behavioural* or *spatial*, *semantic* or *syntactic*) may be enriched, and become more expressive if the underlying transition system exhibits a reactive structure. The qualifier *reactive* classifies a frame, or a transition system, in which some transitions may inhibit others to occur. The study of this sort of structures, and of their corresponding logics, goes back to the seminal paper of van Benthem [18] which introduces what was then called *sabotage logic*. In this language crossing a specifically annotated edge would erase an edge from the underlying Kripke frame. Other variants of reactivity encompass different effects, for example creating new edges [3], or reversing their direction [4]. The topic has received some attention along the last 10 years—see e.g. [12] for a detailed account.

This paper illustrates the use of reactive transition systems to specify Reo connectors in both perspectives, behavioural and spatial, mentioned above. However, to make this a useful feature in practice, one needs to have at hand the tools typically used to reason about transitions, in particular a notion of bisimulation, a logic and a Hennessy-Milner-like theorem relating model bisimilarity and logical equivalence. Such is actually the paper's contribution, adding to the theory of reactive frames developed within the modal logic community.

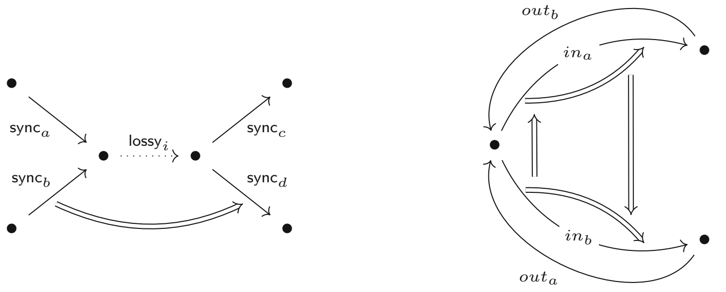
The remaining of the paper is organised as follows. The next section introduces a hybrid logic for reactive transition systems and illustrates the application of such systems to connector modelling. Section 3 contains the core results of the paper, introducing a suitable notion of bisimulation and proving the corresponding Hennessy-Milner theorem for the logic. Finally, Sect. 4 concludes and suggests a few topics for future work.

2 A Hybrid Logic for Reactive Transitions

Figure 1 illustrates how reactive transition systems may be used in the context of connector specification, leading to short and crisp descriptions. On the left hand side is depicted the structure of a merger-broadcaster connector intermediated by a lossy channel; edges in the graph are labelled with the type and identifier of constituent channels. The arrow connecting channel sync_b to sync_d inhibits a transmission on the latter whenever a data token has entered the circuit through the former.

The system on the right, on the other hand, expresses the semantics of a synchronous channel $\leftrightsquigarrow$ whose both ends can either receive or deliver data. Ignoring all the double arrows the behaviour of such a connector would simply be to accept a token at end a (represented by label in_a) and deliver it at b (out_b), or the other way round. As designed, however, a strict alternating discipline is enforced. Actually, crossing transition in_a has two side effects: inhibits itself and, at the same time, removes the inhibition affecting transition in_b which will be selected in the next acceptance round.

A (hybrid) logic to talk about this kind of systems is specified as follows. Note the presence of two kinds of modalities ($\langle a \rangle$ and $\Box_P$) as well as a hybrid satisfaction operator ($@_i$).



(a) Conditional connector structure (b) The semantics of $\leftrightsquigarrow$

Fig. 1. Reactive transition in Reo connectors.

Definition 1. *Given sets Π,* NOM*, and A, of atomic propositions, nominals, and labels respectively, the set of $\mathcal{HL}_r$-*formulas *is defined inductively as follows:*

$$\varphi \ ::= i \,|\, p \,|\, \neg\varphi \,|\, \varphi \vee \varphi \,|\, @_i\varphi \,|\, \langle a\rangle\varphi \,|\, \Diamond_P\varphi$$

for any $i \in$ NOM*, $p \in \Pi$ and $a \in A$. As usual, connectives $\bot$, $\top$, $\wedge$, $\rightarrow$ and $\leftrightarrow$ for $a \in A$ and $\Box_P, [a]$ are introduced as abbreviations.*

As discussed below, formulas are interpreted over paths. Modality $\Diamond_P\varphi$ and its dual, $\Box_P$, are blind for the path taken to arrive to the current state. They are necessary because in a reactive system the accessibility relation may change on the fly. Actually, differently from what happens in classical modal logic, the evaluation of a proposition p at a world w depends on the path taken to reach w. In general, $(W,R), w \models \langle a\rangle p \Leftrightarrow (W,R'), w' \models p$ for some edge w, w' labelled by a, where R' is the accessibility relation obtained after crossing the edge $(w,w') \in R$.

Example 1. Formula $[sync_a][lossy_i]\langle sync_d\rangle\top$ characterises a property valid for the system depicted in Fig. 1(a): the connector allows data to flow through channels $sync_a$, $lossy_i$, and $sync_d$ in sequence. If channel $sync_b$ was taken instead, it would no longer be possible to use $sync_d$ after $lossy_i$. The fact is captured by the formula $[sync_b][lossy_i][sync_d]\bot$. Finally both properties can be combined in one formula resorting to modality $\Diamond_P$: The formula

$$[sync_a][lossy_i]\big(\langle sync_d\rangle\top \wedge \Diamond_P[sync_d]\bot\big)$$

records the possibility of data flowing through channel $sync_d$ for data items coming from $sync_a$, as well as the fact that there exists another possible edge leading to the same state, but which makes impossible to cross $sync_d$.

Formally, let W be a nonempty set of vertices (or states), A a set of labels, and denote by $(A \times W)^*$ the set of all nonempty finite sequences (i.e. *paths*) over $A \times W$. Then,

Definition 2. *A* reactive frame with labels *is a set of finite paths $\Delta \subseteq W \times (A \times W)^*$ such that $(w) \in \Delta$ for any $w \in W$, and $\big(w_1, ((a_2, w_2), ..., (a_n, w_n))\big) \in \Delta$ whenever $\big(w_1, ((a_2, w_2), ..., (a_n, w_n), (a_{n+1}, w_{n+1}))\big) \in \Delta$, for every $n \geq 1$.*

Path composition is denoted by juxtaposition; if no ambiguity arises, w and (a, w) will denote the corresponding singleton paths. Given a set W and a non-empty set of paths $\Delta \subseteq W \times (A \times W)^*$, function $t : \Delta \to W$ returns $t(w) = w$ and $t\big(w_1, ..., (a_n, w_n)\big) = w_n$. In practice, t returns the last state in the path.

Definition 3. *Let Π be a set of atomic propositions. A* reactive model with labels *is a triple (W, Δ, V), where (W, Δ) is a reactive frame with labels, and $V : \Pi \cup \mathrm{NOM} \to 2^{\Delta}$ is a function such that $\lambda \in V(p)$ iff $t(\lambda) \in V(p)$ for any $p \in \Pi, \lambda \in \Delta$ and $|V(i)| = 1$ for any $i \in \mathrm{NOM}$.*

We may now define how to evaluate $\mathcal{HL}_r$-formulas with respect to a reactive model with labels $M = (W, \Delta, V)$, at a path $\lambda \in \Delta$:

Definition 4. *The validity of a $\mathcal{HL}_r$-formula is established recursively:*

- $M, \lambda \models p$ *iff* $\lambda \in V(p)$, *for any* $p \in \Pi$
- $M, \lambda \models i$ *iff* $\lambda \in V(i)$, *for any* $i \in \mathrm{NOM}$
- $M, \lambda \models \neg\varphi$ *iff* $M, \lambda \not\models \varphi$
- $M, \lambda \models \varphi \vee \psi$ *iff* $M, \lambda \models \varphi$ *or* $M, \lambda \models \psi$
- $M, \lambda \models @_i\varphi$ *iff* $M, \gamma \models \varphi$, *where* $V(i) = \{\gamma\}$
- $M, \lambda \models \langle a \rangle\varphi$ *iff* $\exists w \in W,\ \lambda(a, w) \in \Delta$ *and* $M, \lambda(a, w) \models \varphi$
- $M, \lambda \models \Diamond_P\varphi$ *iff* $\exists \gamma \in \Delta,\ M, \gamma \models \varphi$ *and* $t(\lambda) = t(\gamma)$.

Both modalities and the hybrid satisfaction operator are interpreted over paths rather than individual states. For modality $\Diamond_P$ note that a path $\gamma \in \Delta$ is considered accessible from another one $\lambda \in \Delta$ if and only if their final state coincides.

Example 2. Consider the reactive model $M = (W, \Delta, V)$ with labels depicted in Fig. 2. The set of paths Δ is built as follows. From w_4 no move is possible, therefore, $w_4 \in \Delta$. From w_3, one may move to w_4 through an edge labeled by a, thus w_3 and $\big(w_3, (a, w_4)\big) \in \Delta$. Similarly, starting from w_2 leads to w_4, $\big(w_2, (a, w_3)\big)$, $\big(w_2, (a, w_3), (a, w_4)\big) \in \Delta$. Finally, from w_1 one may move to w_3, through an edge labeled by a. However, afterwards, it is not possible to go to w_4 because the edge (w_3, a, w_4) was inhibited when edge (w_1, a, w_3) was crossed. This is the effect represented by the double arrow. Therefore, w_1, $\big(w_1, (a, w_3)\big) \in \Delta$. Δ contains no other paths. Clearly, $M, w_2 \models \langle a \rangle\langle a \rangle\top$ but $M, w_1 \not\models \langle a \rangle\langle a \rangle\top$.

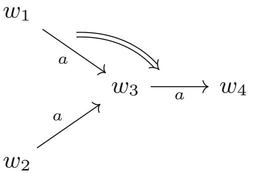


Fig. 2. A reactive frame (W, Δ).

3 Bisimulation for Reactive Models with Labels

This section characterises a notion of bisimulation to compare reactive models with labels. Its relationship with the modal equivalence induced by the logic above is discussed.

Definition 5. *Let (W, Δ, V) and (W', Δ', V') be reactive models over a set of labels A. A relation $\mathcal{S} \subseteq \Delta \times \Delta'$ is an $\mathcal{H}$-bisimulation if and only if*

(nom) *for any $i \in \mathrm{NOM}$, $w \in V(i)$ and $v \in V'(i)$ implies $(w, v) \in \mathcal{S}$*

and, for all $\lambda \in \Delta, \lambda' \in \Delta'$, such that $(\lambda, \lambda') \in \mathcal{S}$,

$(atom)$ *$V(p)(\lambda) = V'(p)(\lambda')$, for all $p \in \Pi \cup \mathrm{NOM}$*
$(A\text{-}zig)$ *$\forall a \in A\Big(\forall w \in W(\lambda(a, w) \in \Delta \Rightarrow \exists w' \in W', \lambda'(a, w') \in \Delta'$ such that $(\lambda(a, w), \lambda'(a, w')) \in \mathcal{S})\Big)$*
$(A\text{-}zag)$ *$\forall a \in A\Big(\forall w' \in W'(\lambda'(a, w') \in \Delta' \Rightarrow \exists w \in W, \lambda(a, w) \in \Delta$ such that $(\lambda(a, w), \lambda'(a, w')) \in \mathcal{S})\Big)$*
$(P\text{-}zig)$ *$\forall \gamma \in \Delta(t(\lambda) = t(\gamma) \Rightarrow \exists \gamma' \in \Delta'\big(t(\lambda') = t(\gamma')$ and $(\gamma, \gamma') \in \mathcal{S})\big)$*
$(P\text{-}zag)$ *$\forall \gamma' \in \Delta'(t(\lambda') = t(\gamma') \Rightarrow \exists \gamma \in \Delta\big(t(\lambda) = t(\gamma)$ and $(\gamma, \gamma') \in \mathcal{S})\big)$*

Example 3. Consider the two systems depicted in Fig. 3. Propositions holding at each particular state are listed between square brackets; no nominals are considered. It is easy to verify that relation $\{\big((w_1), (v_1)\big), \big((w_1, (a, w_2)), (v_1, (a, v_2))\big), \big((w_2), (v_2)\big), \big((w_2, (b, w_3)), (v_2, (b, v_2))\big), \big((w_4), (v_1)\big), \big((w_4, (a, w_3)), (v_1, (a, v_2))\big), \big((w_3), (v_2)\big), \big((w_3, (b, w_2)), (v_2, (b, v_2))\big)\}$ is an $\mathcal{H}$-bisimulation.

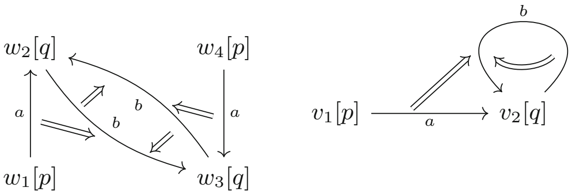


Fig. 3. Two bisimilar models.

As expected, bisimilarity entails modal equivalence.

Theorem 1. *Let (W, Δ, V) and (W', Δ', V') be two reactive models, let $\lambda \in \Delta, \lambda' \in \Delta'$ and let $\mathcal{S} \subseteq \Delta \times \Delta'$ be an $\mathcal{H}$-bisimulation. Then $(\lambda, \lambda') \in \mathcal{S}$ implies $M, \lambda \vDash \varphi \Leftrightarrow M', \lambda' \vDash \varphi$ for every formula $\mathcal{HL}_r$-formula φ.*

Proof. The proof proceeds by induction over the structure of formulas. If $\varphi \in \Pi \cup \mathrm{NOM}$, then $M, \lambda \vDash \varphi \Leftrightarrow M', \lambda' \vDash \varphi$ by definition of $\mathcal{H}$-bisimulation. The non-basic cases are presented below, under the hypothesis that $(\lambda, \lambda') \in \mathcal{S}$.

- $M, \lambda \models \neg\varphi \Leftrightarrow M, \lambda \not\models \varphi \Leftrightarrow M', \lambda' \not\models \varphi \Leftrightarrow M', \lambda' \models \neg\varphi$.
- $M, \lambda \models \varphi \wedge \psi \Leftrightarrow M, \lambda \models \varphi$ and $M, \lambda \models \psi$
  $\Leftrightarrow M', \lambda' \models \varphi$ and $M', \lambda' \models \psi \Leftrightarrow M', \lambda' \models \varphi \wedge \psi$.
- $M, \lambda \models \langle a \rangle \varphi \Rightarrow \exists w \in W$, $\lambda(a, w) \in \Delta$ and $M, \lambda(a, w) \models \varphi$. By definition of bisimulation we conclude that $\exists w' \in W', \lambda'(a, w') \in \Delta'$ such that $\lambda(a, w)$ and $\lambda'(a, w')$ are bisimilar. The induction hypothesis entails $\exists w' \in W'$, $\lambda'(a, w') \in \Delta'$ and $M', \lambda'(a, w') \models \varphi$. Finally, $M', \lambda' \models \langle a \rangle \varphi$. The reciprocal condition is proved analogously.
- $M, \lambda \models \Diamond_P \varphi \Rightarrow \exists \gamma \in \Delta$, $t(\gamma) = t(\lambda)$ and $M, \gamma \models \varphi$. Again, by definition of bisimulation, we conclude that $\exists \gamma' \in \Delta'$ with $t(\gamma') = t(\lambda')$ such that γ' and γ are bisimilar. Thus, by induction, $\exists \gamma' \in \Delta'$ with $t(\gamma') = t(\lambda')$ and $M', \gamma' \models \varphi$, from which we conclude that $M', \lambda' \models \Diamond_P \varphi$. The reciprocal condition is proved analogously.
- $M, \lambda \models @_i \varphi \Leftrightarrow M, \gamma \models \varphi$ such that $V(i) = \{\gamma\}$. From the definition of bisimulation we have $M', \gamma' \models \varphi$ such that $V'(i) = \{\gamma'\}$, and therefore $M', \lambda' \models @_i \varphi$. The reciprocal condition is proved analogously.

□

The reciprocal of Theorem 1 is only valid for a restricted class of models. In classical modal logic a condition stating image-finiteness is usually imposed. We resort here to a slightly more relaxed notion, that of a saturated model [4]. With this restriction, Theorem 2 below explains how an $\mathcal{H}$-bisimulation relating paths indistinguishable by $\mathcal{HL}_r$-formulas can be built. In the sequel, for a relation $Z \subseteq \Delta \times \Delta$ on paths, notation $Z[\lambda]$ abbreviates the set $\{\gamma \in \Delta : \lambda Z \gamma\}$.

Definition 6. *Let Σ be a set of formulas and $M = (W, \Delta, V)$ a reactive model with labels.*

- *Σ is* satisfiable *over a set of paths $\Lambda \subseteq \Delta$ if there is a path $\lambda \in \Lambda$ such that $M, \lambda \models \varphi$ for every $\varphi \in \Sigma$.*
- *Σ is* finitely satisfiable *over a set of paths $\Lambda \subseteq \Delta$ if, for every finite subset $\bar{\Sigma} \subseteq \Sigma$, there is a path $\lambda \in \Lambda$ such that $\lambda \models \varphi$ for every $\varphi \in \bar{\Sigma}$.*
- *A model is* Z-saturated *over a relation $Z \subseteq \Delta \times \Delta$, if, for all λ, every set Σ is satisfiable over $Z[\lambda]$ whenever Σ is finitely satisfiable over $Z[\lambda]$.*

Definition 7. *Given a reactive frame (W, Δ) and a set of labels A, relation $R_a \subseteq \Delta \times \Delta$, for each $a \in A$ is defined by $(\lambda, \gamma) \in R_a$ iff $\exists w \in W, \gamma = \lambda(a, w)$. Similarly, define relation $P \subseteq \Delta \times \Delta$ by $(\lambda, \gamma) \in P$ iff $t(\gamma) = t(\lambda)$.*

In order to complete the Hennessy-Milner theorem for the presented logic, we state and prove the next theorem. It is not proved for all reactive models but comprise an embracing class of reactive models.

Theorem 2. *Let M and M' be two P-saturated and $(R_a)_{a \in A}$-saturated reactive-models with labels. A non-empty relation $\mathcal{S} \subseteq \Delta \times \Delta'$ such that $(\lambda, \lambda') \in \mathcal{S}$ iff for any formula φ, $M, \lambda \models \varphi \Leftrightarrow M', \lambda' \models \varphi$, is an $\mathcal{H}$-bisimulation.*

Proof. Consider an arbitrary $a \in A$, suppose that $(\lambda, \gamma) \in R_a$, for some $\gamma \in \Delta$ and let $Sat(\gamma) = \{\varphi : M, \gamma \models \varphi\}$. Then, for each finite subset $\Sigma' \subseteq Sat(\gamma)$, $M, \lambda \models \langle a \rangle \bigwedge_{\varphi \in \Sigma'} \varphi$ holds and, therefore, $M', \lambda' \models \langle a \rangle \bigwedge_{\varphi \in \Sigma'} \varphi$. This means that $Sat(\gamma)$ is finitely satisfiable over $R_a(\lambda')$, and since M' is R_a-saturated, $Sat(\gamma)$ is satisfied over $R_a(\lambda')$. Thus, there exists a state γ' such that $(\lambda', \gamma') \in R_a$ and $(\gamma, \gamma') \in S$. Analogously, if $(\lambda, \lambda') \in \mathcal{S}$ and $(\lambda', \gamma') \in R_a$, then there exists some $w \in W$ such that $(\lambda w, \lambda' w') \in \mathcal{S}$.

Suppose now that $(\lambda, \gamma) \in P$, for some $\gamma \in \Delta$ and consider $Sat(\gamma) = \{\varphi : M, \gamma \models \varphi\}$. Then, for each finite subset $\Sigma' \subseteq Sat(\gamma)$, $M, \lambda \models \Diamond_P \bigwedge_{\varphi \in \Sigma'} \varphi$ and, therefore, $M', \lambda' \models \Diamond_P \bigwedge_{\varphi \in \Sigma'} \varphi$. This means that $Sat(\gamma)$ is finitely satisfiable over $P_{\lambda'}$, and since M' is P-saturated, $Sat(\gamma)$ is satisfied over $P_{\lambda'}$. Again, there exists a state γ' such that $(\lambda', \gamma') \in P$ and $(\gamma, \gamma') \in S$. Analogously, if $(\lambda, \lambda') \in \mathcal{S}$ and $(\lambda', \gamma') \in P$, then there exists some $\gamma \in \Delta$ such that $(\lambda, \gamma) \in P$ and $(\gamma, \gamma') \in \mathcal{S}$.

Now, let $i \in \mathrm{NOM}$ such that $V(i) = \{\lambda\}$ and $V'(i) = \{\lambda'\}$ for some $\lambda \in \Delta, \lambda' \in \Delta'$. Let $(\gamma, \gamma') \in \mathcal{S} \neq \emptyset$, then, for any $\mathcal{HL}_r$-formula φ, $M, \gamma \models @_i \varphi \Leftrightarrow M', \gamma' \models @_i \varphi$ that semantically implies, $M, \lambda \models \varphi \Leftrightarrow M', \lambda' \models \varphi$. Since φ is arbitrary, $(\lambda, \lambda') \in \mathcal{S}$. Finally, if $(\lambda, \lambda') \in \mathcal{S}$, then we can trivially verify that, $\forall p \in \Pi \cup \mathrm{NOM}$, $M, \lambda \models p \Leftrightarrow M', \lambda' \models p$ by definition.

□

Clearly the theorem would fail for non R_a-saturated models. The following proposition gives a sufficient condition for a model to be R_a-saturated.

Proposition 1. *Let $M = (W, \Delta, V)$ be a reactive model with labels and consider $R_a(\lambda)$ as defined above. If $|R_a(\lambda)| < \infty$, for any $\lambda \in \Delta$, then M is R_a-saturated.*

Proof. Suppose $|R_a(\lambda)| < \infty$, for any $\lambda \in \Delta$, holds for M, but the model is not R_a-saturated. This means that there exists $\lambda \in \Delta$ and a set Σ of formulas such that Σ is finitely satisfiable over $R_a(\lambda)$ but not satisfiable over $R_a(\lambda)$.

Clearly, any formula $\varphi \in \Sigma$, $\{\varphi\}$ is satisfiable over $R_a(\lambda)$ which means that $\langle a \rangle \Sigma = \{\langle a \rangle \varphi : \varphi \in \Sigma\}$ is satisfied in λ. Since $|R_a(\lambda)| < \infty$, every path in $R_a(\lambda)$ can be enumerated as $\gamma_1, \ldots, \gamma_n$. Since Σ is not satisfiable over $R_a(\lambda)$, there is a formula $\varphi_i \in \Sigma$ for each γ_i, $i \in \{1, ..., n\}$, such that φ_i is not satisfied in γ_i. However, for any $i \in \{1, ..., n\}$, $\langle a \rangle \varphi_i$ is satisfied in λ. Thus, the set $\Phi = \{\varphi_1, \ldots, \varphi_n\} \subseteq \Sigma$ is finite and, therefore, satisfiable over $R_a(\lambda)$. This leads to a contradiction since each path $\gamma_i \in R_a(\lambda)$ does not verify $\varphi_i \in \Phi$.

□

An analogous result for relation P is obtained along similar lines, but additionally requiring the absence of cycles in the reactive model.

Our last results establish a connection between reactive models, as discussed in this paper, and the usual Kripke models. We start by making explicit how a reactive model arises from a classical one. Note that in the sequel $NOM = \emptyset$ because nominals, in the logic introduced here, bind paths, rather than states as in standard hybrid logic.

Definition 8. *A Kripke model $K = (W_K, R, V_W)$ induces a reactive model $M = (W, \Delta, V)$ as follows:*

- *$W = W_K$*
- *Δ is the set of all possible paths generated by the accessibility relation R,* i.e.*:*
  - *$(w) \in \Delta$ for any $w \in W$*
  - *For any $n \geq 2$, $(w_0, ..., w_n) \in \Delta$ whenever $\forall i \in \{1, n-1\}, (w_i, w_{i+1}) \in R$*
- *V is defined in order to be coherent with the notion of valuation for models with no reactivity:*
  - *$\forall p \in \Pi$, $\lambda \in V(p)$ iff $t(\lambda) \in V_K(p)$*

Theorem 3. *Let (W_K, R, V_K), (W'_K, R', V'_K) be Kripke models and $\mathcal{B} \subseteq W \times W'$ a bisimulation. Let (W, Δ, V) (respectively, (W', Δ', V')) be the induced reactive model with respect to (W, R, V) (respectively, (W', R', V')). Let the relation $\mathcal{S} \subseteq \Delta \times \Delta'$ be such that $(\lambda, \lambda') \in \mathcal{S}$ iff $t(\lambda)\mathcal{B}t(\lambda')$. Then $\mathcal{S}$ is an $\mathcal{H}-$bisimulation of reactive models (with $NOM = \emptyset$).*

Proof. (nom) Trivial since $NOM = \emptyset$.

$(A - zig)$ Let us consider $\lambda \in \Delta$, $\lambda' \in \Delta'$ and $w \in W$ such that $(\lambda, \lambda') \in \mathcal{S}$ and $\lambda w \in \Delta$. By definition of S, we conclude that $(t(\lambda), t(\lambda')) \in \mathcal{B}$ and $(t(\lambda), w) \in R$. Therefore, since $\mathcal{B}$ is a bisimulation, there exists $w' \in W'$ such that $(t(\lambda'), w') \in R'$ and $(w, w') \in \mathcal{B}$. Thus, $\exists w' \in W', (t(\lambda w), t(\lambda' w')) \in \mathcal{B}$ which implies $\exists w' \in W', (\lambda w, \lambda' w') \in \mathcal{S}$.

$(A - zag)$ Analogous to $A - zig$.

$(P - zig)$ Let us consider $\lambda, \gamma \in \Delta$ and $\lambda' \in \Delta'$ such that $t(\lambda) = t(\gamma)$ and $(\lambda, \lambda') \in \mathcal{S}$. Therefore $(t(\lambda), t(\lambda')) \in \mathcal{B}$ implies $(t(\gamma), t(\lambda')) \in \mathcal{B}$ and, thus, $(\gamma, \lambda') \in \mathcal{S}$. The result follows because, trivially, $t(\lambda') = t(\lambda')$.

$(P - zag)$ Analogous to $P - zag$.

$(atom)$ $M, \lambda \vDash_X p \Leftrightarrow M, t(\lambda) \vDash p$ for any $p \in \Pi$. Thus, if $(\lambda, \lambda') \in \mathcal{S}$, then $M, \lambda \vDash_X p \Leftrightarrow M, t(\lambda) \vDash p \Leftrightarrow M', t(\lambda') \vDash p \Leftrightarrow M', \lambda' \vDash_X p$

□

In the opposite direction a similar result pops out:

Theorem 4. Let (W_K, R, V_K), (W'_K, R', V'_K) be Kripke models and $\mathcal{S} \subseteq \Delta \times \Delta'$ a bisimulation between paths of the corresponding induced reactive models. Define relation $\mathcal{B} \subseteq W \times W'$ by $(w, w') \in \mathcal{B}$ iff there exists $(\lambda, \lambda') \in \mathcal{S}$ such that $t(\lambda) = w$ and $t(\lambda') = w'$. Then $\mathcal{B}$ is a bisimulation between the original Kripke models.

Proof. We prove the *zig* and *zag* conditions, as well as the semantic equivalence between atomic propositions.

(*zig*) Let us suppose $(w, w') \in \mathcal{B}$ and that there exists $v \in W$ such that $(w, v) \in R$. Then, there exists $(\lambda, \lambda') \in \mathcal{S}$ such that $t(\lambda) = w$ and $t(\lambda') = w'$. Furthermore, $\lambda w \in \Delta$. Since $\mathcal{S}$ is a bisimulation, there exists $v' \in W'$ such that $\lambda' w' \in \Delta'$ and $(\lambda w, \lambda' v') \in \mathcal{S}$. Hence, $t(\lambda v) = v$, $t(\lambda' v') = v'$ and, therefore, $v\mathcal{B}v'$.

(*zag*) Analogous to *zig*.

(*atom*) Finally, we note that $M, \lambda \vDash_X p \Leftrightarrow M, t(\lambda) \vDash p$ for any $p \in \Pi$ because $X = \Pi$. If $(w, w') \in \mathcal{B}$, then there exists $(\lambda\, \lambda') \in \mathcal{S}$ such that $t(\lambda) = w$ and $t(\lambda') = w'$. Because $M, \lambda \vDash_X p \Leftrightarrow M', \lambda' \vDash_X p$, we conclude that $M, t(\lambda) \vDash p \Leftrightarrow M', t(\lambda') \vDash p$, *i.e.*, $M, v \vDash p \Leftrightarrow M', v' \vDash p$. □

As explained above, the set of nominals was assumed to be empty in the context in which both theorems were formulated. However, if nominals were considered in the language of reactive models, the second theorem would remain valid, but not the first one, as non bisimilar paths may be bound by the same nominal.

4 Conclusions and Future Work

This paper indicates that *reactive* transition systems may play an interesting role in the formal description of software connectors in Reo, either from a behavioural (semantic) or spatial (syntactic) point of view. In the later sense, we are currently enriching our previous work on connector reconfiguration [17] to handle such reactive spatial descriptions of coordination patterns.

A preliminary step for those developments is a suitable characterisation of equivalence. Such is the focus of the technical contents of the present paper. We introduce a notion of bisimulation for reactive transition systems with labels and establish, under reasonable conditions, a Hennessy-Milner-like result. This adds to the quest of Gabbay and his collaborators for suitable (logic) tools to specify the dynamics of reactivity. We are currently extending our work to *switch graphs* [13], another sort of reactive frames with an interesting potential for describing coordination patterns, namely in the context of analogues to biological systems—a main topic in the first author's doctoral research [10,11].

Acknowledgments. This work is a result of project "SmartEGOV/NORTE-01-0145-FEDER-000037", supported by Norte Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, through the European Regional Development Fund (EFDR). Additional support was provided by the European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project UID/MAT/04106/2013 at CIDMA. The first author is also supported by an Individual Doctoral Grant (reference number PD/BD/114186/2016).

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)

2. Arbab, F.: Abstract behavior types: a foundation model for components and their composition. Sci. Comput. Program. **55**(1–3), 3–52 (2005)
3. Areces, C., Fervari, R., Hoffmann, G.: Swap logic. Logic J. IGPL **22**(2), 309–332 (2014)
4. Areces, C., Fervari, R., Hoffmann, G.: Relation-changing modal operators. Logic J. IGPL **23**(4), 601–627 (2015)
5. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. **61**(2), 75–113 (2006)
6. Barbosa, M.A., Barbosa, L.S.: A perspective on service orchestration. Sci. Comput. Program. **74**(9), 671–687 (2009)
7. Blackburn, P.: Representation, reasoning, and relational structures: a hybrid logic manifesto. Logic J. IGPL **8**(3), 339–365 (2000)
8. Bonsangue, M., Clarke, D., Silva, A.: Automata for context-dependent connectors. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 184–203. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02053-7_10
9. Brauner, T.: Hybrid Logic and its Proof-Theory. Applied Logic Series. Springer, Heidelberg (2010). https://doi.org/10.1007/978-94-007-0002-4
10. Figueiredo, D.: Relating bisimulations with attractors in Boolean network models. In: Botón-Fernández, M., Martín-Vide, C., Santander-Jiménez, S., Vega-Rodríguez, M. (eds.) AlCoB 2016. LNCS, vol. 9702, pp. 17–25. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-38827-4_2
11. Figueiredo, D., Martins, M.A., Chaves, M.: Applying differential dynamic logic to reconfigurable biological networks. Math. Biosci. **291**, 10–20 (2017)
12. Gabbay, D., Marcelino, S.: Modal logics of reactive frames. Stud. Logica **93**(2), 405–446 (2009)
13. Gabbay, D., Marcelino, S.: Global view on reactivity: switch graphs and their logics. Ann. Math. Artif. Intell. **66**(1–4), 1–32 (2012)
14. Jongmans, S.-S.T.Q., Arbab, F.: Overview of thirty semantic formalisms for Reo. Sci. Ann. Comp. Sci. **22**(1), 201–251 (2012)
15. Kokash, N., Krause, C., de Vink, E.P.: Reo + mCRL2: a framework for model-checking dataflow in service compositions. Formal Asp. Comput. **24**(2), 187–216 (2012)
16. Krause, C., Maraikar, Z., Lazovik, A., Arbab, F.: Modeling dynamic reconfigurations in Reo using high-level replacement systems. Sci. Comput. Program. **76**(1), 23–36 (2011)
17. Oliveira, N., Barbosa, L.S.: Reasoning about software reconfigurations: the behavioural and structural perspectives. Sci. Comput. Program. **110**, 78–103 (2015)
18. van Benthem, J.: An essay on sabotage and obstruction. In: Hutter, D., Stephan, W. (eds.) Mechanizing Mathematical Reasoning. LNCS (LNAI), vol. 2605, pp. 268–276. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32254-2_16

Personal Note: Working with Farhad Arbab 1990–2005

Kees Blom(✉)

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Kees.Blom@cwi.nl

From 1990 to 2005 the author had the privilege and pleasure to work as a programmer in Farhad Arbab's research group, initially on the development, implementation and applications of the **Manifold** coordination language. During these years I learned to appreciate his broad scientific knowledge and technical insight, as well as his humor and warm personality.

The work on **Manifold** evolved in a number of moderate scale trials (a dozen or so cooperating computers) in different application fields [1–4,7]. The latter work on 'Chaotic Iteration' raised a question how to effectively implement Distributed Termination Detection (DTD) for sets of asynchronously connected processes without a central supervising process. This proved to be an old problem [5,8] for which many solutions had been proposed under various conditions [11], until recently [9]. Because at that time all existing DTD algorithms had serious drawback in terms of prerequisites and properties, a new experimental DTD algorithm was designed and implemented in **Manifold**. However, these experiments showed that other programming languages such as Java were more suitable for programming this type of algorithms.

This effort resulted in a Java implementation the **BTTFWave** protocol that was used in a number of applications, see [10] where also a short description of this protocol is given. Also, a formal correctness proof of the **BTTFWave** protocol has been presented in [12]. All these experiences may have helped to inspire Farhad in setting up a totally new paradigm for component composition: Reo [6], which in later years proved to be quite fruitful.

It so happened that around that time Farhad was appointed as a professor at Leiden University, and I was asked to join another research group. Although we managed by working in weekends and evening hours to complete specification, implementation and some applications of the **BTTFWave** protocol, Farhad never found the opportunity to publish more extensively on this work.

Hopefully this may change now that Farhad has reached the state of retirement, because in my opinion it is quite an elegant and effective protocol, that deserves a wider audience. In any event, I wish him all the best in years to come.

F. de Boer et al. (Eds.): Arbab Festschrift, LNCS 10865, pp. 68–69, 2018.
https://doi.org/10.1007/978-3-319-90089-6_5

References

1. Blom, C.L., Arbab, F., Hummel, S., Elshoff, I.J.P.: Coordination of a heterogeneous coastal hydrodynamics application in manifold. Technical report SEN-R9833, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, December 1998. https://ir.cwi.nl/pub/4589
2. Bonsanque, M.M., Arbab, F., de Bakker, J.W., Rutten, J.J.M.M., Scutella, A., Zavattaro, G.: A transition system semantics for the control-driven coordination language manifold. Theoret. Comput. Sci. **240**(1), 3–47 (2000)
3. Everaars, C.T.H., Arbab, F., Koren, B.: Dynamic process composition and communication patterns in irregularly structured applications. Concurrency: Practice Exp. **12**(2–3), 157–174 (2000)
4. Monfroy, E.: A coordination-based chaotic iteration algorithm for constraint propagation. In: Proceedings of the 2000 ACM Symposium on Applied Computing, SAC 2000, vol. 1, pp. 262–269. ACM, New York (2000)
5. Dijkstra, E.W.D., Scholten, C.S.: Termination detection for diffusing computations. Inf. Process. Lett. **11**(1), 1–4 (1980)
6. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
7. Arbab, F., Blom, C.L., Burger, F.J., Everaars, C.T.H.: Reusable coordinator modules for massively concurrent applications. Softw.: Practice Exp. **28**(7), 703–735 (1998). Extended version
8. Francez, N.: Distributed termination. ACM Trans. Program. Lang. Syst. **2**(1), 42–55 (1980)
9. Árdal, K.B.: A fault-tolerant variant of the mahapatra-dutt termination detection algorithm. Master thesis, Vrije Universiteit, Amsterdam, 26 August 2017. http://www.cs.vu.nl/~wanf/theses/ardal-mscthesis.pdf
10. Apt, K.R., Arbab, F., Ma, H.: A distributed platform for mechanism design. In: 2008 International Conference on Computational Intelligence for Modelling Control & Automation, pp. 767–772. IEEE (2008)
11. Matocha, J., Camp, T.: A taxonomy of distributed termination detection algorithms. J. Syst. Softw. **43**, 207–221 (1998)
12. Degenhardt, M.: Proving the correctness of the BTTF wave algorithm for distributed termination detection. Master thesis, Vrije Universiteit, Amsterdam, July 2014. http://www.cs.vu.nl/~wanf/theses/degenhardt.pdf

Soft Constraint Automata with Memory

Kasper Dokter[1], Fabio Gadducci[2], and Francesco Santini[3](✉)

[1] Centrum Wiskunde & Informatica, Amsterdam, Netherlands
K.P.C.Dokter@cwi.nl
[2] Dipartimento di Informatica, University of Pisa, Pisa, Italy
fabio.gadducci@unipi.it
[3] Dipartimento di Matematica e Informatica, University of Perugia, Perugia, Italy
francesco.santini@unipg.it

Abstract. In this paper, we revise the notion of *Soft Constraint Automata*, where automata transitions are weighted and consequently each action is associated with a preference value. We first relax the underlying algebraic structure that models preferences, with the purpose to use bipolar preferences (i.e., both positive and negative ones). Then, we equip automata with memory cells, that is, with an internal state to remember and update information from transition to transition. Finally, we revise automata operators, as join and hiding.

Keywords: Constraint automata with memory · Soft constraints

1 Introduction

In the history of Computer Science, many coordination languages have been proposed for the specification and implementation of interaction protocols, in order to let software components communicate. Such formalisms include process calculi, concurrent objects, actors, agents, shared memory, message passing, and more. A distinctive feature of these formalisms is that they are all primarily action-based models that provide constructs for the direct specification of things that interact, rather than a direct specification of interaction (i.e., protocols).

This is one of the main motivations behind the long-running success of the Reo language [2], whose distinctive feature instead is to treat interaction as an explicit first-class concept. Reo comes with its own composition operators, and it allows for specifying more complex interaction protocols by combining simpler, and possibly primitive, protocols. In practice, Reo connectors impose constraints on the order in which the components can exchange data items with each other; even though the basic primitive channels are simple, Reo connectors can actually describe rather complex protocols.

The literature offers tens of different semantics formalisms to express the behaviour of Reo connectors [17]: *co-algebraic*, *colouring*, and other models based on, for instance, constraints and Petri nets. The *operational* models (i.e., automata) are probably the most popular approaches: *Constraint Automata* [8]

F. de Boer et al. (Eds.): Arbab Festschrift, LNCS 10865, pp. 70–85, 2018.
https://doi.org/10.1007/978-3-319-90089-6_6

(*CA*) and (several) related variants, and *Context-sensitive Automata*. Variants of CA consists in *Timed*, *Probabilistic*, *Continuous-time*, *Quantitative Resource-sensitive timed*, and *Transactional* extensions.

In the remainder of this paper, our aim is to both relax and extend one of the chronologically latter variants of CA, hence not included in the survey in [17]: *Soft Constraint Automata* [6], also called *Soft Component Automata* [19, 20] (SCA in both cases). SCA is a state-transition system where transitions are labelled with actions and preferences. Higher-preference transitions typically contribute more towards the goal of the component.

The aim of this paper consists of three sub-goals. First, we relax the definition of the underlying structure that models preferences: instead of semirings as in [6,19,20], we exploit partially ordered monoids (see Sect. 2) to naturally relax soft constraints in order to represent bipolar preferences as labels for automata transitions. In this way, we can express both positive and negative values, e.g., costs and retributions for firing a transition rule.

Second, we extend SCA with a notion of memory (SCAM), as Arbab and co-authors have already accomplished for CA [18]. Each transition of a SCAM can also put a condition on the current data assigned to a finite set of *memory cells*, and update their respective values. Therefore, together with states, memory cells determine the configuration of a connector, and thus can influence the observed behaviour of the component.

The third and last intention is less scientific, but more significant from our side: we feel the need to celebrate Farhad's influential intuition behind Reo, and his far-reaching contribution to many fields of Computer Science as, to name only two of them, Concurrency Theory and Coordination Models and Languages. Indeed, besides personal gratitude,[1] our will is to continue playing with *"Puff's gigantic tail"* for a long time ahead [3].

The outline of the paper is as follows: in Sect. 2 we set the background notions behind the algebraic structure we use for our soft constraints, that is partially ordered monoids; then, in Sect. 3 we just define soft constraint functions. In Sect. 4 we introduce and formally define SCAM, while Sect. 5 summarises the related work. Finally, Sect. 6 wraps up the paper with conclusive thoughts and hints about related future research.

2 Partially Ordered Monoids

The first step is to define an algebraic structure for modelling preferences. It falls into the range of *bipolar* approaches: we refer to [14] for the missing proofs as well as for an introduction and a comparison with other proposals.

[1] Francesco would like to thank Farhad for his precious mentoring during his visit at CWI as "Alain Bensoussan" Fellow during 2011–2012; Fabio for the many meetings and collaborations along the years; and Kasper for his incredible supervision job and the many interesting discussions.

Definition 1 (Orders). *A partial order (PO) is a pair $\langle A, \leq \rangle$ such that A is a set and $\leq \subseteq A \times A$ is a reflexive, transitive, and anti-symmetric relation. A PO is a complete lattice (CL) if any subset of A has a least upper bound (LUB).*

The LUB of a subset $X \subseteq A$ is denoted $\bigvee X$, and it is unique. By definition $\bot = \bigvee \emptyset$ is the bottom of the PO and $\top = \bigvee A$ is the top.

Definition 2 (PO monoids). *A (commutative) monoid is a triple $\langle A, \oplus, \mathbf{0} \rangle$ such that $\oplus : A \times A \to A$ is a commutative and associative function satisfying*

- $\forall a \in A . a \oplus \mathbf{0} = a$, *where* $\mathbf{0} \in A$ *is the* identity *element.*

A partially ordered monoid (POM) is a 4-tuple $\langle A, \leq \oplus, \mathbf{0} \rangle$ such that $\langle A, \leq \rangle$ is a PO and $\langle A, \oplus, \mathbf{0} \rangle$ a monoid. It is monotone if

- $\forall a, b, c \in A.\, a \leq b \implies a \oplus c \leq b \oplus c$

and it is distributive if

- $\forall a \in A. \forall X \subseteq A.\, a \oplus \bigvee X = \bigvee \{ a \oplus x \mid x \in X \}$.

whenever X is finite. A complete lattice monoid (CLM) is a POM such that the underlying order is a CL, it is monotone if the underlying POM is so and it is distributive if the property holds for possibly infinite subsets.

Note that in a distributive POM the $\oplus$ operation is monotone. In the following, we usually use an infix notation $a \oplus b$ for $\oplus(a, b)$.

Remark 1. It is now easy to show that distributive POMs are *tropical* semirings, i.e., semirings with a sum operator $a \oplus b = \bigvee \{a, b\}$ that is idempotent. If $\mathbf{0}$ is also the top of the PO we end up in what are called *absorptive* semirings [16] in the algebra literature, which in turn are known as *c*-semirings in the soft constraint jargon [10]. Combined with monotonicity, imposing $\mathbf{0}$ to be the top means that preferences are negative, i.e., $\forall a, b \in A . a \oplus b \leq a$. Indeed, most better known structures that are used in the soft constraints literature are absorptive semirings whose underlying, distributive POM is actually a CLM: among them we recall the *Boolean* ($\langle \{\mathit{false}, \mathit{true}\}, \to, \wedge, \mathit{true} \rangle$), *Fuzzy* ($\langle [0,1], \leq, \min, 1 \rangle$), *Probabilistic* ($\langle [0,1], \leq, \times, 1 \rangle$), and *Tropical* ($\langle \mathbb{R}^+ \cup \{+\infty\}, \geq, +, 0 \rangle$) semirings (where, in the latter, $\geq$ the inverse of the standard order, thus $+\infty$ is the bottom and 0 the top of the CLM, respectively).

Remark 2. Note that CLMs feature an operator that carries the intuitive meaning of subtraction, which can be defined as $a \ominus b = \bigvee \{ c \mid b \oplus c \leq a \}$. It satisfies the usual property of residuation, namely $b \oplus c \leq a \iff c \leq a \ominus b$: see e.g. [11] for a brief survey on residuation for absorptive semirings.

2.1 Cylindric Algebras

We now introduce two families of operators that we use for modelling suitable operators on automata. They are generalised notions of existential quantifiers and diagonals [22]. For this section, we fix a POM $\mathcal{M} = \langle A, \leq, \oplus, \mathbf{0} \rangle$.

Definition 3 (Cylindrification). *Let V a set of variables. A cylindric operator $\exists$ over $\mathcal{M}$ and V is a family of monotone, identity preserving functions $\exists_x : A \to A$ indexed by elements in V such that for all $a, b \in A$ and $x, y \in V$*

1. $a \leq \exists_x a$
2. $\exists_x (a \oplus \exists_x b) = \exists_x a \oplus \exists_x b$
3. $\exists_x \exists_y a = \exists_y \exists_x a$

The support *of $a \in A$ is the set of variables* $\text{supp}(a) = \{x \in V \mid \exists_x a \neq a\}$.

Preserving identities means that $\exists_x \mathbf{0} = \mathbf{0}$. Combined with item 2, it implies idempotency of $\exists$, i.e., $\exists_x \exists_x a = \exists_x a$, which implies $x \notin \text{supp}(\exists_x a)$. Since $\exists$ is commutative, we denote $\exists_{x_1} \cdots \exists_{x_n} a$ as $\exists_X a$, for $X = \{x_1, \ldots, x_n\} \subseteq V$.

We now fix a set of variables V and a cylindric operator $\exists$ over $\mathcal{M}$ and V.

Definition 4 (Diagonalisation). *A diagonal operator δ for $\exists$ is a commutative function $\delta : V \times V \to A$ such that for all $a \in A$ and $x, y, z \in V$*

1. $\delta_{x,x} = \mathbf{0}$
2. $\delta_{x,y} = \exists_z (\delta_{x,z} \oplus \delta_{z,y})$ *for* $z \notin \{x, y\}$
3. $\delta_{x,y} \oplus \exists_x (a \oplus \delta_{x,y}) \leq a$ *for* $x \neq y$

We use a subscript notation, as $\delta_{x,y}$ for $\delta(x, y)$. Axioms 1 and 2 above plus idempotency imply $\exists_x \delta_{x,y} = \mathbf{0}$, which implies (by axiom 2 and idempotency of $\exists$) that $\text{supp}(\delta_{x,y}) = \{x, y\}$ for $x \neq y$. We lastly fix a diagonal operator δ for $\exists$.

Definition 5 (Substitution). *Let $x, y \in V$ and $a \in A$. The substitution $a[^y/_x]$ is defined as a if $x = y$ and as $\exists_x (\delta_{x,y} \oplus a)$ otherwise.*

Substitution behaves correctly with respect to $\exists$.

Lemma 1. *For all $x, y, w \in V$ and $a \in A$, we have*

- $(\exists_x a)[^y/_x] = \exists_x a$;
- $\exists_x a = \exists_y (a[^y/_x])$, *if* $y \notin \text{supp}(a)$;
- $(\exists_w a)[^y/_x] = \exists_w (a[^y/_x])$, *if* $w \notin \{x, y\}$.

Proof. The proofs are immediate. Consider for instance the most difficult item 3. If $x = y$ the proof is over. Now, since $w \notin \{x, y\}$ we have that $w \notin \text{supp}(\delta_{x,y})$, so that $\exists_w (a[^y/_x]) = \exists_w \exists_x (\delta_{x,y} \oplus a) = \exists_x \exists_w (\delta_{x,y} \oplus a) = \exists_x (\delta_{x,y} \oplus \exists_w a) = (\exists_w a)[^y/_x]$.

Finally, we can now rephrase some additional laws that hold for the crisp case (see e.g. [7, p. 140]).

Lemma 2. *For all $x, y \in V$ and $a \in A$, we have*

1. $(a[^y/_x])[^x/_y] = a$, *if* $y \notin \mathrm{supp}(a)$*;*
2. $a[^y/_x] \oplus b[^y/_x] = (a \oplus b)[^y/_x]$*;*
3. $x \notin \mathrm{supp}(a[^y/_x])$*, if* $x \neq y$*.*

Proof. Consider the most difficult item 2. By definition $a[^y/_x] \oplus b[^y/_x] = \exists_x(\delta_{x,y} \oplus a) \oplus \exists_x(\delta_{x,y} \oplus b)$, which in turn coincides with $\exists_x(\delta_{x,y} \oplus a \oplus \exists_x (\delta_{x,y} \oplus b))$ by axiom 2 of $\exists$, and by axiom 3 of δ and its idempotency we have that $\exists_x(\delta_{x,y} \oplus a \oplus \exists_x(\delta_{x,y} \oplus b)) \leq \exists_x(\delta_{x,y} \oplus a \oplus b) = (a \oplus b)[^y/_x]$. The vice versa holds by the monotonicity of $\exists$, so that $\exists_x(\delta_{x,y} \oplus b) \geq \delta_{x,y} \oplus b$. Item 1 has a similar proof, while 3 is immediate.

3 A Key Example: Soft Constraints

Previously, we mentioned as typical examples of distributive CLMs the Fuzzy semiring $\langle [0,1], \leq, \min, 1\rangle$ of the $[0,1]$ interval of real numbers with the usual order and multiplication as the monoidal operator, and the Tropical semiring $\langle \mathbb{R}^+ \cup \{+\infty\}, \geq, +, 0\rangle$ of non-negative reals plus ∞ with the inverse order and addition. In this section, we give a pivotal example of a CLM that is a cylindric algebra, introducing the notion of soft constraint (following, yet generalising [12]).

Definition 6 (Soft constraints). *Let V be a set of variables, $\mathcal{D}$ a data domain and $\mathcal{M} = \langle A, \leq, \oplus, \mathbf{0}\rangle$ a CLM. A* (soft) constraint $c : (V \to \mathcal{D}) \to A$ *is a function associating a value in A to every variable assignment $\eta : V \to \mathcal{D}$.*

We define C as the set of constraints that can be built starting from chosen $\mathcal{M}$, V, and $\mathcal{D}$. The application of a constraint function $c : (V \to \mathcal{D}) \to A$ to a variable assignment $\eta : V \to \mathcal{D}$ is denoted $c\eta$.

Although a constraint involves all the variables in V, it may depend on the assignment of a finite subset of them, called its support (cf. Definition 3). For instance, a binary constraint c with $\mathrm{supp}(c) = \{x, y\}$ is a function $c : (V \to \mathcal{D}) \to A$ which depends only on the assignment of variables $\{x, y\} \subseteq V$, meaning that two assignments $\eta_1, \eta_2 : V \to \mathcal{D}$ differing only for the image of variables $z \notin \{x, y\}$ coincide (i.e., $c\eta_1 = c\eta_2$). The support generalises the classical notion of scope of a constraint.[2] We often refer to a constraint with support X as c_X.

The set of constraints forms a CLM, with the structure lifted from $\mathcal{M}$.

Lemma 3 (CLM of constraints). *The set of constraints C (over $\mathcal{M}$, V, and $\mathcal{D}$) is endowed with a relation $\leq$, operation $\oplus$, and constant $\mathbf{0}$, such that*

- $c_1 \leq c_2$ *if* $c_1\eta \leq c_2\eta$ *for all* $\eta : V \to \mathcal{D}$
- $(c_1 \oplus c_2)\eta = c_1\eta \oplus c_2\eta$
- $\mathbf{0}\eta = \mathbf{0}$

is a complete lattice monoid.

[2] For a first-order constraint ϕ, the support $\mathrm{supp}(\phi)$ is contained in the set of free variables $\mathrm{free}(\phi)$. For example, $\mathrm{supp}(x = x) = \emptyset \subseteq \{x\} = \mathrm{free}(x = x)$.

We denote the CLM $\langle C, \leq, \oplus, \mathbf{0} \rangle$ of constraints by $\mathcal{C}$. Combining two constraints by the $\oplus$ operator means building a new constraint whose support contains at most the variables of the original ones. Such constraint associates with each tuple of domain values for such variables the value obtained by multiplying those associated by the original constraints to the appropriate sub-tuples. The identity is the constant function mapping all η to $\mathbf{0}$.

Example 1 (A simple CLM). Let us consider $\mathcal{S}$ as the CLM of non-negative reals, and as $\mathcal{D}$ any subset of such reals. A linear polynomial with variables in V and non-negative reals as coefficients such as $ux + vy + z$ can be interpreted as the soft constraint associating with a function $\eta : V \rightarrow \mathcal{D}$ the real obtained as $(u \times \eta(x)) + (v \times \eta(y)) + z$. Clearly, the composition of such constraints is precisely the addition of polynomials. Instead, the ordering might not be the one induced by the coefficients, due to the presence of constants. For example, let us consider the polynomials $2x + 1$ and $x + 5$ and let us assume $\mathcal{D} = \{1, 2, 3\}$: it holds that $(2x + 1) \oplus (x + 5) = (3x + 6)$ and $2x + 1 \leq x + 5$.

Similarly for residuation, which is just bounded subtraction of coefficients. Since $2x + 1 \leq x + 5$, by construction $(2x + 1) \ominus (x + 5)$ is the bottom constraint, which can be represented by the polynomial 0. Instead, $(x + 5) \ominus (2x + 1)$ could be synthetically described as $-x + 4$, even if the latter falls outside of the polynomials we considered since it has a negative coefficient. In general terms, also such polynomials might be allowed: it would suffice to assume that if the result of the evaluation of the polynomial is a negative real, then it is put to 0.

If $\mathcal{D}$ is not the singleton, then the support of a polynomial is precisely the set of variables occurring in it.

The CLM of constraints enjoys the cylindric properties, as shown by the result below (for cylindric operators and diagonals in the idempotent case, see [12]).

Lemma 4 (Cylindric algebra of constraints). *The CLM of constraints $\mathcal{C}$ endowed with cylindric operators $\exists_x$ and diagonal elements $\delta_{x,y}$, such that*

- $(\exists_x c)\eta = \bigvee_{d \in \mathcal{D}} c\eta[x := d]$*, for all* $c \in C, x \in V$
- $\delta_{x,y}\eta = \begin{cases} \mathbf{0} & \textit{if } \eta(x) = \eta(y) \\ \bot & \textit{otherwise} \end{cases}$ *, for all* $x, y \in V$

is a cylindric algebra.

Differently from the tradition in soft constraint literature, we allow the data domain $\mathcal{D}$ to be infinite. Hence, $\exists_x$ may need to compute the least upper bound of an infinite set of soft constraints. However, Lemma 3 shows that $\mathcal{C}$ is a CLM, which guarantees the existence of such an upper bound. These observations motivate why we view the set of constraints as a CLM rather than just a POM.

Hiding means removing variables from supports: $\text{supp}(\exists_x c) \subseteq \text{supp}(c) \setminus \{x\}$.[3] Finally, the diagonal element $\delta_{x,y}$ has support $\{x, y\}$ for $x \neq y$, and $\emptyset$ for $\delta_{x,x}$.

[3] The operator is called *projection* in the soft framework, and $\exists_x c$ is denoted $c \Downarrow_{V - \{x\}}$.

Example 2 (Continued...). Let us consider again the situation of Example 1. Polynomials can also be equipped with a cylindric operator, so that e.g. $\exists_x(2x + 1) = \bigvee_{d\in\mathcal{D}} 2d + 1 = 3$, i.e., the supremum obtained for the evaluation of the polynomial with respect to the elements in $\mathcal{D}$. The diagonal operator $\delta_{x,y}$ is a kind of matching $[x = y]$, since $[x = y]\eta$ is either 0 or ∞ depending if $\eta(x) = \eta(y)$ or $\eta(x) \neq \eta(y)$, respectively. A proper treatment would anyhow require to extend the syntax of polynomials by including suitable constants.

4 Soft Constraint Automata

Constraint Automata (CA) have bee introduced in [8] as a formalism to describe the behaviour and possible data flow in coordination models (such as Reo language [8]); they can be considered as acceptors of *Timed Data Streams* [1,8]. The proposal has been recently enriched by adding an explicit notion of memory [18]. In this section we rephrase most definitions presented in the weighted extension of CAs, namely *Soft Constraint Automata* (SCA) [6], and we further extend the framework by relying on CLMs instead of c-semirings as in [6] and by including memory, thus obtaining Soft Constraint Automata *with Memory* (SCAM).

4.1 Weighted Data Streams

As a first step, we recall and rephrase the definition of *Weighted Data Streams* (WDS), which is also given in [6].[4]

In this section we fix a data domain $\mathcal{D}$ as well as a CLM $\mathcal{M} = \langle A, \leq, \oplus, \mathbf{0}\rangle$.

Definition 7 (Weighted data streams). *A weighted data stream (WDS) over $\mathcal{D}$ and $\mathcal{M}$ is a partial function $\phi : \mathbb{R}_+ \rightharpoonup (A \setminus \{\bot\}) \times \mathcal{D}$ whose domain* $\mathrm{dom}(\phi)$ *is closed and discrete. We write WDS for the set of all weighted data streams.*

Intuitively, a WDS $\phi : \mathbb{R}_+ \rightharpoonup (A\setminus\{\bot\})\times\mathcal{D}$ records the time stamps $\mathrm{dom}(\phi) = \{\phi_0, \phi_1, \ldots\}$, such that $i < j$ implies $\phi_i < \phi_j$, at which data is exchanged. Indeed, discreteness (each element is isolated) ensures that $\mathrm{dom}(\phi)$ is countable (hence, it has the cardinality of a subset of natural numbers). Since it is also closed, the domain contains no unrealistic situations like $\{1/n \mid n \in \mathbb{N}\}$, since its limit would not be discrete. The values $\phi(\phi_i) = (a_i, d_i)$, for $i \geq 0$ and $\phi_i \in \mathrm{dom}(\phi)$, consists of the observed data d_i and its preference $a_i \neq \bot$.

Our current definition of WDSs slightly differs from the original definition in [6]. The main distinction is that we allow finite WDSs, i.e., the domain $\mathrm{dom}(\phi)$ may be finite. In case of an empty domain $\mathrm{dom}(\phi) = \emptyset$, WDS ϕ admits no observable behaviour at all. This generalisation is especially useful for the hiding operator on SCAM, as proposed in Sect. 4.6.

The second difference with the original definition in [6] is that our WDSs are partial functions, rather than triples of streams. However, our assumptions on the domain imply that these representations are similar.

[4] As noted in [6], these streams do not imply time constraints, and thus our (soft) CA are not "timed" [8], so that we dropped the adjective altogether.

The behaviour of a system is often described in terms of tuples of WDSs, each one corresponding to a data passing through a given port; put simply, given a finite set of port names $\mathcal{N}$, the behaviour is described by a function $\mathcal{N} \rightarrow \mathit{WDS}$.

4.2 Soft Constraint Automata with Memory

In a CA, a transition label is a pair $\langle N, g \rangle$ consisting of a synchronisation constraint N and a data constraint g. The synchronisation constraint is a finite set of names that consists exactly of those input/output ports through which data is exchanged during the current transition. The data constraint is a boolean formula that guards the data exchanged at the ports in N. In the soft framework, the overall structure is similar, even if the guard is now a soft constraint that evaluates to an element in $\mathcal{M}$, instead of a boolean value. Furthermore, we extend our data constraints with memory variables. Using these variables, a guard can require a property about the current and next state of the data in memory.

Besides a data domain $\mathcal{D}$ and a finite set of port names $\mathcal{N}$, we fix a finite set $\mathcal{X}$ of memory cells and a CLM $\mathcal{M} = \langle A, \leq, \oplus, \mathbf{0} \rangle$. Furthermore, we consider the set $\mathcal{X}^v = {}^\bullet\mathcal{X} \cup \mathcal{X}^\bullet$ of memory variables: they are obtained by tagging the memory cells. To avoid conflicting names, we assume that $\mathcal{N} \cap \mathcal{X}^v = \emptyset$.

Definition 8 (Soft constraints with memory). *A soft constraint with memory over $\mathcal{N}$, $\mathcal{X}$, $\mathcal{D}$, and $\mathcal{M}$ is a soft constraint $c : (\mathcal{N} \cup \mathcal{X}^v \rightarrow \mathcal{D}) \rightarrow A$.*

We denote by $\mathcal{C}_\mathcal{X}$ the CML of soft constraints with memory. Informally, a soft constraint with memory is a function that returns a preference value $a \in A$ given an assignment for a subset N of names in $\mathcal{N}$ and a subset X of variables in $\mathcal{X}^v$ that occur in its support.

Note that, by using the *Boolean* semiring we model the "crisp" data-constraints presented in the definition of CA [8]. Therefore, CA are subsumed by Definition 9. Note also that weighted automata have already been defined in the literature [13]; in SCA, weights are determined by a constraint function instead.

Definition 9 (Soft constraint automata with memory). *A Soft Constraint Automaton with Memory (SCAM) over $\mathcal{D}$ and $\mathcal{M}$ is a tuple $\langle \mathcal{Q}, \mathcal{X}, \mathcal{N}, \longrightarrow, \mathcal{Q}_0 \rangle$ such that $\mathcal{Q}$ is a finite set of states, $\mathcal{X}$ a finite set of memory cells, $\mathcal{N}$ a finite set of port names, $\longrightarrow \subseteq \mathcal{Q} \times 2^\mathcal{N} \times \mathcal{C}_\mathcal{X} \times \mathcal{Q}$ a finite set of transitions, and $\mathcal{Q}_0 \subseteq \mathcal{Q}$ a set of initial states, such that $(q, N, c, p) \in \longrightarrow$ implies* $\mathrm{supp}(c) \subseteq N \cup \mathcal{X}^v$.

We usually write $q \xrightarrow{N,c} p$ instead of $(q, N, c, p) \in \longrightarrow$ and we call N the synchronisation constraint and c the guard of the transition, respectively. We say a transition is *invisible* whenever $N = \emptyset$.

The intuitive meaning of a SCAM $\mathcal{T}$ as an operational model for service queries is similar to the interpretation of labelled transition systems as models for reactive systems. The states represent the configurations of a service. The transitions represent the possible one-step behaviour, where the meaning of $q \xrightarrow{N,c} p$ is that, in configuration q, the ports in $n \in N$ have the possibility of performing I/O operations that satisfy the soft guard c, which now also includes requirements on memory cells, and that leads from configuration q to p, while the other ports in $\mathcal{N}\backslash N$ perform no I/O operation. Each assignment to port names in N represents the data exchanged by the I/O operations through these ports, while the assignments to variables in ${}^\bullet X$ and $X^\bullet$ represent the data in memory cells before and after the transition.

Example 3 (Buying and selling). In Fig. 1, we show a (deterministic) SCAM, where the set of names $\mathcal{N}$ is $\{b, s\}$, the set of variables $\mathcal{X}^v$ is $\{{}^\bullet a, a^\bullet, {}^\bullet l, l^\bullet\}$, and the data domain consists of all integers. The constraints c_b and c_s describe the preferences on the operation of buying and selling: if the cost b of buying an item is below a threshold with respect to the value in the account ${}^\bullet a$ (for the sake of simplicity, the account remains positive, i.e., ${}^\bullet a - b > 0$), then the item is bought and the account is decreased ($a^\bullet = {}^\bullet a - b$ and $l^\bullet = b$). Likewise, if the price s received in selling an item is above the paid price (${}^\bullet l < s$), then it is accepted and the account is incremented ($a^\bullet = {}^\bullet a + s$).

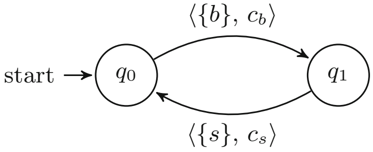


Fig. 1. An automaton with memory.

The formal definition of constraints c_b and c_s is given in Eqs. 1 and 2.

$$c_{\{b\}}\eta = \begin{cases} -\eta(b) \ \textit{if} \ 0 < \eta(l^\bullet) = \eta(b) = \eta({}^\bullet a) - \eta(a^\bullet) < \eta({}^\bullet a) \\ \bot \qquad\qquad\qquad\qquad\qquad\qquad \textit{otherwise} \end{cases} \tag{1}$$

$$c_{\{s\}}\eta = \begin{cases} \eta(s) \ \textit{if} \ \eta({}^\bullet l) < \eta(s) = \eta(a^\bullet) - \eta({}^\bullet a) \\ \bot \qquad\qquad\qquad\qquad\qquad \textit{otherwise} \end{cases} \tag{2}$$

For example, let us assume that for the first two transitions (from q_0 to q_1 and from q_1 to q_0, respectively) we have $\eta(b) = 2$ and then $\eta(s) = 3$. Also, let us assume that we have as initial account $a = 6$ (i.e., ${}^\bullet a = 6$). Now, the constraint $c_{\{b\}}\eta$ associated to the first transition has value -2, since the money is spent, and the values associated to the memory cells l and a after the transition (i.e., to the variables $l^\bullet$ and $a^\bullet$) are 2 and 4, respectively. The constraint $c_{\{s\}}\eta$ associated to the second transition has value 3, since the money is earned (and it is more than what the item was paid), and the value associated to the memory cell a after the transition (i.e., to the variable $a^\bullet$) is 7, while the value associated to the memory cell l (i.e., to the variable $l^\bullet$) is irrelevant, hence it can be any value.

4.3 The Language of SCAM

Let $\mathcal{T}$ be a SCAM and $\Omega_{\mathcal{X}} : \mathcal{X} \to \mathcal{D}$ the set of assignments over its memory cells. The configurations of $\mathcal{T}$ are pairs $\langle q, \omega \rangle \in \mathcal{Q} \times \Omega_{\mathcal{X}}$, which are initial whenever $q \in \mathcal{Q}_0$. In other words, we initialise each memory cell to a random datum.

The accepted language of a SCAM $\mathcal{T}$ at a given configuration s is a relation $\mathcal{L}(\mathcal{T}, s) \subseteq \mathit{WDS}^{\mathcal{N}}$ over weighted data streams on ports from $\mathcal{N}$. An accepted word consists of a $\mathcal{N}$-tuple of weighted data streams (i.e., a map $\mathcal{N} \to \mathit{WDS}$). As usual, the language of an automaton is given by the union of all the languages accepted by its initial states, i.e., $\mathcal{L}(\mathcal{T}) = \bigcup_{s \in \mathcal{Q}_0 \times \Omega_{\mathcal{X}}} \mathcal{L}(\mathcal{T}, s)$.

Consider the automaton $\mathcal{T}$ from Example 3, with ports $\mathcal{N} = \{b, s\}$ and starting state q_0. The accepted language of $\mathcal{T}$ is formed by the runs of alternate assignments for $\{b\}$ and $\{s\}$, which guarantees that a balance a is always positive.

Recall that $\phi_0 \in \mathbb{R}_+$ is the minimum of the domain $\mathrm{dom}(\phi)$, for every WDS $\phi : \mathbb{R}_+ \rightharpoonup A \times \mathcal{D}$ with non-empty domain.

Definition 10 (Accepted runs). *Let $\mathcal{T} = \langle \mathcal{Q}, \mathcal{X}, \mathcal{N}, \longrightarrow, \mathcal{Q}_0 \rangle$ be a SCAM. The accepted language of $\mathcal{T}$ is the largest map $\mathcal{L}(\mathcal{T}, -) : \mathcal{Q} \times \Omega_{\mathcal{X}} \to 2^{(\mathit{WDS})^{\mathcal{N}}}$, such that if $(\phi^x)_{x \in \mathcal{N}} \in \mathcal{L}(\mathcal{T}, \langle q, \omega_0 \rangle)$ and $t_0 = \min\{\phi_0^x \mid x \in \mathcal{N}, \mathrm{dom}(\phi^x) \neq \emptyset\}$ exist, there exist a transition $q \xrightarrow{N,c} p$ and an assignment $\eta : \mathcal{N} \cup \mathcal{X}^v \to \mathcal{D}$ with*

- *$c\eta \neq \bot$,*
- *$N = \emptyset$ or $N = \{x \in \mathcal{N} \mid \phi_0^x = t_0\}$,*
- *$\phi^x(t_0) = \langle c\eta, \eta(x) \rangle$, for all $x \in N$,*
- *$\eta({}^\bullet x) = \omega_0(x)$, for all $x \in \mathcal{X}$,*
- *$(\phi^x|_{\mathbb{R}_+ \setminus \{t_0\}})_{x \in \mathcal{N}} \in \mathcal{L}(\mathcal{T}, \langle p, \omega_1 \rangle)$, with $\omega_1(x) = \eta(x^\bullet)$, for all $x \in \mathcal{X}$,*

where $\phi^x|_{\mathbb{R}_+ \setminus \{t_0\}}$ is the restriction of ϕ^x to $\mathbb{R}_+$ without t_0. The set $\mathcal{L}(\mathcal{T})$ of accepted runs is the union of the acceptable runs from $\mathcal{Q}_0 \times \Omega_{\mathcal{X}}$.

Observe that the expression $\phi^x(t_0)$ is well-defined, because $x \in N$ implies $N \neq \emptyset$ and $t_0 = \phi_0^x \in \mathrm{dom}(\phi^x)$. Also note that, since $\mathrm{supp}(c) \subseteq N \times \mathcal{X}^v$, the value of $\eta(x)$, for $x \in \mathcal{N} \setminus N$, is irrelevant.

If we never observe any data at any port (i.e., $\mathrm{dom}(\phi^x) = \emptyset$, for all $x \in \mathcal{N}$), then the minimum $t_0 = \min\{\phi_0^x \mid x \in \mathcal{N}, \mathrm{dom}(\phi^x) \neq \emptyset\}$ does not exist, and the condition in Definition 10 is vacuously true.

Example 4 (The language of business). Going back to Example 3, the language recognised by state q_0 with respect to the memory cell assignment $[a = p, l = 0]$ is the possibly infinite sequence of weights and port name assignments $\langle -b_1, b = b_1 \rangle; \langle s_1 - b_1, s = s_1 \rangle; \ldots$ such that $0 < b_i \leq s_i$ and additionally in any (even) prefix of such a sequence the sum of all the costs and all the sales is always positive. Indeed, it is given by the sum of all constraints! The timing of these operations is instead irrelevant and it is omitted.

4.4 Stateless SCAM

States and memory cells of SCAM essentially serve the same purpose. The following theorem shows that we can eliminate all states from any SCAM, at the cost of new memory cells.

Theorem 1. *Let $\mathcal{T}$ be a SCAM over a domain $\mathcal{D}$ that is not a singleton. Then, it is language equivalent to a SCAM with only a single state.*

Proof. Let $\mathcal{T} = (\mathcal{Q}, \mathcal{X}, \mathcal{N}, \longrightarrow, \mathcal{Q}_0)$ be a SCAM over a data domain $\mathcal{D}$ and a CLM $\mathcal{M}$ such that $\mathcal{D}$ is not a singleton. We find an injective encoding $e : Q \to \mathcal{D}^n$, for some integer $n \geq \log_{|\mathcal{D}|}(|Q|)$. Write $e(q) = (e_i(q))_{i=1}^n$, for all $q \in Q$. For every variable $x \in \mathcal{X}^v$ and datum $d \in \mathcal{D}$, define the (soft) constraint $x = d$ by $(x = d)\eta = \mathbf{0}$, if $\eta(x) = e_i(q)$, and $(x = d)\eta = \bot$, otherwise. Let $z_1, \ldots, z_n \notin \mathcal{N} \cup \mathcal{X}$ be fresh names. For every $\tau = (q, N, c, p) \in \longrightarrow$, define $N_\tau = N$ and

$$c_\tau = c \oplus \bigoplus_{i=1}^{n} {}^\bullet z_i = e_i(q) \oplus z_i^\bullet = e_i(p).$$

Consider $\mathcal{T}' = (\{q_0\}, \mathcal{X} \cup \{z_1, \ldots, z_n\}, \mathcal{N}, \{(q_0, N_\tau, c_\tau, q_0) \mid \tau \in \longrightarrow\}, \{q_0\})$. We show that $\mathcal{T}'$ is language equivalent to $\mathcal{T}$. Suppose that $(\phi^x)_{x \in \mathcal{N}} \in \mathcal{L}(\mathcal{T})$. Then, we find some transition $(q, N, c, p) \in \longrightarrow$ and assignment $\eta : \mathcal{N} \cup \mathcal{X}^v \to \mathcal{D}$ that satisfy the conditions in Definition 10. Since the z_i's are fresh, we find an extension η' of η to $\bigcup_{i=1}^n \{{}^\bullet z_i, z_i^\bullet\}$ that satisfies $c_\tau \eta' \neq \bot$. Hence, transition $(q_0, N_\tau, c_\tau, q_0)$ and assignment η' witness that $(\phi^x)_{x \in \mathcal{N}} \in \mathcal{L}(\mathcal{T}')$.

On the other hand, suppose that $(\phi^x)_{x \in \mathcal{N}} \in \mathcal{L}(\mathcal{T}')$. Then, we find some transition $(q_0, N_\tau, c_\tau, q_0)$, with $\tau \in \longrightarrow$, and an assignment $\eta' : \mathcal{N} \cup (\mathcal{X} \cup \{z\})^v \to \mathcal{D}$ that satisfy the conditions in Definition 10. Then, τ and the restriction η of η' to $\mathcal{N} \cup \mathcal{X}^v$ witness that $(\phi^x)_{x \in \mathcal{N}} \in \mathcal{L}(\mathcal{T})$. We conclude that $\mathcal{L}(\mathcal{T}) = \mathcal{L}(\mathcal{T}')$. □

The result of Theorem 1 can be used as a first step towards an equivalence between SCAM and soft constraints with memory (over a data domain that includes a special datum $*$ that denotes absence of data). Moreover, by using $*$, it seems possible to encode the synchronisation constraint N of each transition in the soft constraint by adding $x \neq *$, for $x \in N$, and $x = *$, otherwise. In combination with Theorem 1, such construction would show a correspondence between SCAM and soft constraints with memory. Since the representation of SCAM as soft constraints with memory is much more flexible, this alternative representation can help us to prevent relentless state space explosions. We leave the details of such correspondence as future work.

4.5 SCAM Composition

We now introduce the product of automata, extending [6, Definition 5].

Definition 11 (Soft join). *Let $\mathcal{T}_i = (\mathcal{Q}_i, \mathcal{X}_i, \mathcal{N}_i, \to_i, \mathcal{Q}_{0i})$, for $i \in \{0, 1\}$, be two SCAM over $\mathcal{D}$ and $\mathcal{M}$, with $(\mathcal{N}_0 \cup \mathcal{N}_1) \cap (\mathcal{X}_0^v \cup \mathcal{X}_1^v) = \emptyset$. Then, their soft* *product* $\mathcal{T}_0 \bowtie \mathcal{T}_1$ *is the tuple* $\langle \mathcal{Q}_0 \times \mathcal{Q}_1, \mathcal{X}_0 \cup \mathcal{X}_1, \mathcal{N}_0 \cup \mathcal{N}_1, \longrightarrow, \mathcal{Q}_{00} \times \mathcal{Q}_{01} \rangle$ *where* $\longrightarrow$ *is the smallest relation that satisfies the rule*

$$\frac{q_0 \xrightarrow{N_0, c_0}_0 p_0, \quad q_1 \xrightarrow{N_1, c_1}_1 p_1, \quad N_0 \cap \mathcal{N}_1 = N_1 \cap \mathcal{N}_0}{\langle q_0, q_1 \rangle \xrightarrow{N_1 \cup N_2, c_1 \oplus c_2} \langle p_0, p_1 \rangle}$$

The rule applies when there is a transition in each automaton such that they can fire together. This happens only if the two local transitions agree on the subset of shared ports that *fire*. The transition in the resulting automaton is labelled with the union of the name sets on both transitions, and the constraint is the conjunction of the constraints of the two transitions.

Note that the product automaton can include asynchronous executions: it suffices that the SCAM are *reflexive*, i.e., for any state q there is a transition $q \xrightarrow{\emptyset, \mathbf{0}} q$.

We can express the composition of SCAM in Definition 11 in terms of a simple composition operator on languages. Let $\mathcal{L}_0$ and $\mathcal{L}_1$ be two languages over the sets of ports $\mathcal{N}_0$ and $\mathcal{N}_1$, respectively. The product language $\mathcal{L}_0 \bowtie \mathcal{L}_1$ consists of those tuples $(\phi^x)_{x \in \mathcal{N}_0 \cup \mathcal{N}_1}$ of WDSs, such that $(\phi^x)_{x \in \mathcal{N}_i} \in \mathcal{L}_i$, for all $i \in \{0, 1\}$. In particular, $\mathcal{N}_0 = \mathcal{N}_1$ implies $\mathcal{L}_0 \bowtie \mathcal{L}_1 = \mathcal{L}_0 \cap \mathcal{L}_1$.

Lemma 5 (Correctness of soft join). *Let* $\mathcal{T}_0$ *and* $\mathcal{T}_1$ *be two SCAM that do not share memory cells. Then,* $\mathcal{L}(\mathcal{T}_0 \bowtie \mathcal{T}_1) = \mathcal{L}(\mathcal{T}_0) \bowtie \mathcal{L}(\mathcal{T}_1)$*.*

Proof. Suppose that $(\phi^x)_{x \in \mathcal{N}_0 \cup \mathcal{N}_1} \in \mathcal{L}(\mathcal{T}_0 \bowtie \mathcal{T}_1)$. We show that $(\phi^x)_{x \in \mathcal{N}_i} \in \mathcal{L}(\mathcal{T}_i)$, for all $i \in \{0, 1\}$. Let $i \in \{0, 1\}$ be arbitrary. By Definition 10, we find some transition $q \xrightarrow{N,c} p$ and an assignment $\eta : \mathcal{N}_0 \cup \mathcal{N}_1 \cup \mathcal{X}_0^v \cup \mathcal{X}_1^v \to A$ that satisfy the conditions in Definition 10. By Definition 11, we find a local transition $q_i \xrightarrow{N_i, c_i} p_i$ in $\mathcal{T}_i$, and by restriction of η, we find a local assignment $\eta_i : \mathcal{N}_i \cup \mathcal{X}_i^v \to A$. By construction, this transition and assignment witness the inclusion $(\phi^x)_{x \in \mathcal{N}_i} \in \mathcal{L}(\mathcal{T}_i)$ according to the conditions in Definition 10. By definition of join, we have $\mathcal{L}(\mathcal{T}_0 \bowtie \mathcal{T}_1) \subseteq \mathcal{L}(\mathcal{T}_0) \bowtie \mathcal{L}(\mathcal{T}_1)$.

Suppose that $(\phi^x)_{x \in \mathcal{N}_0 \cup \mathcal{N}_1} \in \mathcal{L}(\mathcal{T}_0) \bowtie \mathcal{L}(\mathcal{T}_1)$. We show that $(\phi^x)_{x \in \mathcal{N}_0 \cup \mathcal{N}_1} \in \mathcal{L}(\mathcal{T}_0 \bowtie \mathcal{T}_1)$. By definition, we have $(\phi^x)_{x \in \mathcal{N}_i} \in \mathcal{L}(\mathcal{T}_i)$, for all $i \in \{0, 1\}$. By Definition 11, we find, for $i \in \{0, 1\}$, a transition $q_i \xrightarrow{N_i, c_i} p_i$ and an assignment $\eta_i : \mathcal{N}_i \cup \mathcal{X}_i^v \to A$ that satisfy the conditions in Definition 10. By construction, we have $N_0 \cap \mathcal{N}_1 = N_1 \cap \mathcal{N}_0$, and we find with Definition 11 a global transition $q \xrightarrow{N,c} p$ in $\mathcal{T}_0 \bowtie \mathcal{T}_1$. Since $\mathcal{T}_0$ and $\mathcal{T}_1$ do not share memory, we have $\mathcal{X}_0^v \cap \mathcal{X}_1^v = \emptyset$. Therefore, $\eta_0 \cup \eta_1$ is a well-defined assignment that, together with $q \xrightarrow{N,c} p$, witnesses that $(\phi^x)_{x \in \mathcal{N}_0 \cup \mathcal{N}_1} \in \mathcal{L}(\mathcal{T}_0 \bowtie \mathcal{T}_1)$. We conclude that $\mathcal{L}(\mathcal{T}_0 \bowtie \mathcal{T}_1) = \mathcal{L}(\mathcal{T}_0) \bowtie \mathcal{L}(\mathcal{T}_1)$. □

4.6 SCAM Hiding

The hiding operator [8] abstracts the details of the internal communication in a CA. In SCA [6, Definition 6], the hiding operator $\exists_O \mathcal{T}$ removes from the transitions all the information about the names in $O \subseteq \mathcal{N}$, including those in the (support of the) constraints. The definition smoothly extends over SCAM: in fact, since we allow invisible transitions, our definition is much more compact.

Definition 12 (Soft hiding). *Let* $\mathcal{T} = \langle \mathcal{Q}, \mathcal{X}, \mathcal{N}, \longrightarrow, \mathcal{Q}_0 \rangle$ *be a SCAM and* $O \subseteq \mathcal{N}$ *a set of port names. Then,* $\exists_O \mathcal{T}$ *is the SCAM* $\langle \mathcal{Q}, \mathcal{X}, \mathcal{N} \setminus O \longrightarrow_*, \mathcal{Q}_0 \rangle$ *where* $\longrightarrow_*$ *is defined by* $q \xrightarrow{N \setminus O, \exists_O c}_* p$ *if* $q \xrightarrow{N,c} p$.

Similarly to the correctness of join, we express the hiding operator for SCAM in terms of a simple operation on languages. Let $\mathcal{L}$ be a language over a set of ports $\mathcal{N}$, and let $O \subseteq \mathcal{N}$ be a set of ports. Then, $\exists_O \mathcal{L}$ consists exactly of the restriction $(\phi^x)_{x \in \mathcal{N} \setminus O}$ of a given tuple $(\phi^x)_{x \in \mathcal{N}} \in \mathcal{L}$.

If we require that the domain $\mathrm{dom}(\phi)$ of a WDS ϕ is infinite, then we can only hide ports that necessarily fire infinitely often in every (infinite) run of the SCAM. This condition is part of the correctness of hiding for SCA [6, Definition 6]. Since we allow finite WDS, correctness of SCAM amounts to the following result.

Lemma 6 (Correctness of soft hiding). *Let* $\mathcal{T}$ *be a SCAM and* O *a set of its ports. Then,* $\mathcal{L}(\exists_O \mathcal{T}) = \exists_O \mathcal{L}(\mathcal{T})$.

Proof. Suppose that $(\phi^x)_{x \in \mathcal{N} \setminus O} \in \mathcal{L}(\exists_O \mathcal{T})$. We will show by coinduction that $(\phi^x)_{x \in \mathcal{N} \setminus O} \in \exists_O \mathcal{L}(\mathcal{T})$. By Definition 10, we find some transition $q \xrightarrow{N',c'} p$ in $\exists_O \mathcal{T}$. By Definition 12, we find some transition $q \xrightarrow{N,c} p$, with $N' = N \setminus O$ and $c' = \exists_O c$. We construct a WDS ϕ^x, for $x \in O$, such that $(\phi^x)_{x \in \mathcal{N}}$ satisfies the conditions of Definition 10, for $\mathcal{T}$. If $x \in N \cap O$, we define $\phi^x = [t_0 \mapsto \langle c\eta, \eta(d) \rangle]$, with $t_0 := \min\{\phi_0^p \mid p \in \mathcal{N} \setminus O, \mathrm{dom}(\phi^p) \neq \emptyset\}$ and $\eta : \mathcal{N} \cup \mathcal{X}^v \to \mathcal{D}$ is any assignment that satisfies c. If $x \in (\mathcal{N} \setminus N) \cap O$ and we define $\phi^x = [\,]$ as the empty map. By the coinduction hypothesis, $(\phi^x)_{x \in \mathcal{N}} \in \mathcal{L}(\mathcal{T})$, and $(\phi^x)_{x \in \mathcal{N} \setminus O} \in \exists_O \mathcal{L}(\mathcal{T})$.

On the other hand, suppose that $(\phi^x)_{x \in \mathcal{N} \setminus O} \in \exists_O \mathcal{L}(\mathcal{T})$. By definition, we find some extension $(\phi^x)_{x \in \mathcal{N}} \in \mathcal{L}(\mathcal{T})$. By coinduction hypothesis and Definition 10 we find $(\phi^x)_{x \in \mathcal{N} \setminus O} \in \mathcal{L}(\exists_O \mathcal{T})$. We conclude that $\mathcal{L}(\exists_O \mathcal{T}) = \exists_O \mathcal{L}(\mathcal{T})$. □

5 Related Works on Constraint Automata

The closest related work to what discussed in this paper concerns other extensions of CA, previously advanced in the ample and mature literature about Reo.

In [5,21], Arbab et al. introduce *Quantitative* CA (QCA) with the aim of describing the behaviour of connectors tied to their *Quality of Service* (QoS), e.g., a reliability measure or the shortest transmission time. Similarly to CA, the states of a QCA correspond to the internal states of the connector it models. The label on a transition consists instead of a firing set, a data constraint, and a cost that represents a QoS metric. Hence, QCA differ from *Timed* [4] and *Probabilistic* [9] CA, because these latter two classes of models describe functional aspects of connectors, while QoS represents non-functional properties.

As applications, SCA have been already used in [6,23] and [19,20,24]. Differently from previous related work, the main motivation behind SCA is to associate an action with a preference. In [6,23] the authors present a formal framework that is able to discover stateful Web Services, and to rank the results according to a similarity score expressing the affinities between the query, asked by a user, and the services in a database. Preference for the similarity between the query and each service is modelled through SCA. In the second bunch of works instead, the authors advance a framework that facilitates the construction of autonomous agents in a compositional fashion; these agents are "soft", in that their actions are associated with a preference value, and agents may or may not execute an action depending on a threshold preference. Hence, at design-time SCA can be used to reason about the behaviour of the components in an uncertain physical world, i.e., to model and verify the behaviour of cyber-physical systems.

6 Conclusions

We have reworked Soft Constraint Automata as originally proposed in [6,19], with the dual purpose of *(i)* extending the underlying algebraic structure in order to model both positive and negative preferences, and *(ii)* adding memory cells as originally provided for "standard" CA [18]. Therefore, the main objective has been to further generalise the notion of SCA, which could already accommodate different preference systems parametrically.

As future work, we have many interesting directions in mind. As a start, we would like to exploit the properties of soft constraints to give additional operators on SCAM, first of all an operator for port *renaming*, or for considering *deterministic* accepted runs, i.e., where memory cells that are not in the support of a constraint labelling a transition are not modified by that transition. We will consider a more flexible notion of accepted run by taking into account *weak* transitions, i.e., by considering the relation $q \overset{\emptyset,c}{\Longrightarrow} p$ obtained as the sequence $q \xrightarrow{\emptyset,c_1} q_1 \dots q_{n-1} \xrightarrow{\emptyset,c_n} p$ such that $c = c_1 \oplus \dots \oplus c_n$ and the relation $q \overset{N,c}{\Longrightarrow} p$ obtained as $q \overset{\emptyset,c_1}{\Longrightarrow} q_1 \xrightarrow{N,c_2} p_1 \overset{\emptyset,c_3}{\Longrightarrow} p$ such that $c = c_1 \oplus c_2 \oplus c_3$. This would be pivotal in defining a proper notion of *weak bisimulation* for automata.

Finally, thinking about our result on single state automata, we would like to encode the behaviour of SCA into a *concurrent constraint programming* language [15]. Such languages provide agents with actions to *tell* (i.e., add) and *ask* (i.e., query) constraints to a centralised store of information; this store represents a *Constraint Satisfaction Problem*, and standard heuristics-based technique might be applied to find a solution to complex conditions on *filter channels* [3].

References

1. Arbab, F., Rutten, J.J.M.M.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2002. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40020-2_2
2. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
3. Arbab, F.: Puff, the magic protocol. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 169–206. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24933-4_9
4. Arbab, F., Baier, C., de Boer, F.S., Rutten, J.J.M.M.: Models and temporal logical specifications for timed component connectors. Softw. Syst. Model. **6**(1), 59–82 (2007)
5. Arbab, F., Chothia, T., Meng, S., Moon, Y.-J.: Component connectors with QoS guarantees. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 286–304. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72794-1_16
6. Arbab, F., Santini, F.: Preference and similarity-based behavioral discovery of services. In: ter Beek, M.H., Lohmann, N. (eds.) WS-FM 2012. LNCS, vol. 7843, pp. 118–133. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38230-7_8
7. Aristizábal, A., Bonchi, F., Palamidessi, C., Pino, L., Valencia, F.: Deriving labels and bisimilarity for concurrent constraint programming. In: Hofmann, M. (ed.) FoSSaCS 2011. LNCS, vol. 6604, pp. 138–152. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19805-2_10
8. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. **61**(2), 75–113 (2006)
9. Baier, C.: Probabilistic models for Reo connector circuits. Univers. Comput. Sci. **11**(10), 1718–1748 (2005)
10. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. J. ACM **44**(2), 201–236 (1997)
11. Bistarelli, S., Gadducci, F.: Enhancing constraints manipulation in semiring-based formalisms. In: Brewka, G., Coradeschi, S., Perini, A., Traverso, P. (eds.) ECAI 2006. FAIA, vol. 141, pp. 63–67. IOS Press (2006)
12. Bistarelli, S., Montanari, U., Rossi, F.: Soft concurrent constraint programming. ACM Trans. Comput. Logic **7**(3), 563–589 (2006)
13. Droste, M., Kuich, W., Vogler, H. (eds.): Handbook of Weighted Automata. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01492-5
14. Gadducci, F., Santini, F.: Residuation for bipolar preferences in soft constraints. Inf. Process. Lett. **118**, 69–74 (2017)
15. Gadducci, F., Santini, F., Pino, L.F., Valencia, F.D.: Observational and behavioural equivalences for soft concurrent constraint programming. Log. Algebr. Methods Program. **92**, 45–63 (2017)
16. Golan, J.: Semirings and Affine Equations over Them: Theory and Applications. Kluwer, Norwell (2003)
17. Jongmans, S.T.Q., Arbab, F.: Overview of thirty semantic formalisms for Reo. Sci. Ann. Comput. Sci. **22**(1), 201–251 (2012)

18. Jongmans, S.T.Q., Kappé, T., Arbab, F.: Constraint automata with memory cells and their composition. Sci. Comput. Program. **146**, 50–86 (2017)
19. Kappé, T., Arbab, F., Talcott, C.L.: A compositional framework for preference-aware agents. In: Kargahi, M., Trivedi, A. (eds.) V2CPS@IFM 2016. EPTCS, vol. 232, pp. 21–35 (2016)
20. Kappé, T., Arbab, F., Talcott, C.: A component-oriented framework for autonomous agents. In: Proença, J., Lumpe, M. (eds.) FACS 2017. LNCS, vol. 10487, pp. 20–38. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68034-7_2
21. Meng, S., Arbab, F.: QoS-driven service selection and composition using quantitative constraint automata. Fundamenta Informaticae **95**(1), 103–128 (2009)
22. Saraswat, V.A., Rinard, M.C., Panangaden, P.: Semantic foundations of concurrent constraint programming. In: Wise, D.S. (ed.) POPL 1991, pp. 333–352. ACM Press (1991)
23. Sargolzaei, M., Santini, F., Arbab, F., Afsarmanesh, H.: A tool for behaviour-based discovery of approximately matching web services. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM 2013. LNCS, vol. 8137, pp. 152–166. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40561-7_11
24. Talcott, C., Nigam, V., Arbab, F., Kappé, T.: Formal specification and analysis of robust adaptive distributed cyber-physical systems. In: Bernardo, M., De Nicola, R., Hillston, J. (eds.) SFM 2016. LNCS, vol. 9700, pp. 1–35. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-34096-8_1

On the Relation Between Control-Based and Data-Based Coordination Languages

Jean-Marie Jacquet[1(✉)], Isabelle Linden[2], and Denis Darquennes[1]

[1] Faculty of Computer Science, University of Namur,
Rue Grandgagnage 21, 5000 Namur, Belgium
{jean-marie.jacquet,denis.darquennes}@unamur.be

[2] Business Administration Department, University of Namur,
Rempart de la Vierge 8, 5000 Namur, Belgium
isabelle.linden@unamur.be

Abstract. Coordination languages have classically been divided into control-based coordination languages, on the one hand, and data-based coordination languages, on the other hand. The great majority of work on coordination addresses the one family or the another but rarely connects the two. In the honor of the retirement of a leading expert of control-based coordination languages, the authors, who devoted many research efforts on data-based coordination languages, aim at addressing the connection between the two families of coordination languages. To that end, a Reo-like dialect, named ReoD, is first presented. Variants of a Linda-like language, named BachT, VBachT and MRT, are then described and subsequently used to translate and simulate the ReoD language.

Keywords: Coordination languages · Control · Data · Reo · Linda
Bach

1 Introduction

Since its very beginning Computer Science has improved in many fields, such as the reduction of the size of the hardware components, the increase in their speed of execution, but also the degree of complexity of applications. This is, among others, due to the development of the internet which has lead to conceive applications in a distributed and heterogeneous way.

Gelernter and Carriero already recognized that fact in 1992 and postulate in [23] that programming would gain by separating two main concerns: real computations and communication between components. That has lead to their seminal equation *Programming* = *Coordination* + *Computation*. Slightly before, they had proposed a novel paradigm, named Linda [22], based on the deposit and retrieval of pieces of information on a shared space. Since then many languages extending this idea have been proposed. The authors have themselves contributed to that trend of research, as exemplified for instance in [8,11,16–18,24,26,32–34]. Research in the domain is yearly reported in the International

F. de Boer et al. (Eds.): Arbab Festschrift, LNCS 10865, pp. 86–106, 2018.
https://doi.org/10.1007/978-3-319-90089-6_7

Conference on Coordination Languages and Models as well as in several workshops.

Farhad Arbab had a similar diagnostic since the early 90's but proposed another manner of organising coordination. Taking inspiration from Unix shell scripts, he developed a language named Manifold to "describe and manage complex interactions among independent and concurrent processes" [41]. Similarly to pipes in Unix, this model is based on linking the input and output ports of processes. The model evolved along the years in what we now know as Reo [1]. It created a new trend of coordination models, not relying on the availability of data but more on the control or coordination patterns to which concurrent processes should agree. Farhad introduced this classification in his paper [35], but also made the point clear at his talk in the first International Conference on Coordination Languages and Models. There, by similarity with the classical disclaimer of films, according to which any resemblance to real persons, living or dead, is to be considered as purely coincidental, he declared that any resemblance to the work on Linda in his talk was also to be considered as coincidental.

This paper aims at linking the two worlds of coordination languages just introduced. More specifically, after having identified in Sect. 2 a Reo-like language, named ReoD, and after having described in Sect. 3 different variants of Bach, a Linda-like language developed at the University of Namur, we shall explore the translation of ReoD in these variants in Sect. 4. To that end, we shall first show that the external behavior of ReoD connectors can be described in a BachT variant based on multiset rewriting, inspired by the chemical metaphore. Moreover, we shall establish that the internal behavior of the ReoD connectors can be described by a vector-based variant of BachT. The relations of these two descriptions will also be addressed. These two forms of translation suggest a methodology of development of coordination programs by a high-level specification inspired from ReoD and translated into Bach-like languages. Finally, Sect. 5 compares our work with related work, draws our conclusions and presents expectations for future work.

2 The Reo Language

2.1 Linguistic Description

Reo [1] is a channel-based coordination language wherein complex coordinators are composed from primitive connectors, called channels, by means of nodes.

A channel is a medium of communication that consists of two ends and a protocol that describes how data flows at those ends. There are two types of ends: source and sink. A *source end* accepts data into the channel and a *sink end* outputs data out of the channel. For instance, the Sync channel depicted as $a \longrightarrow b$ has a source end at a and a sink end at b. It specifies that data can be accepted at point a if and only if it can be output at point b. More specifically, this means that, for the communication to proceed, a process connected at a must be ready to output the data at a while another process connected at b must be ready to input the data.

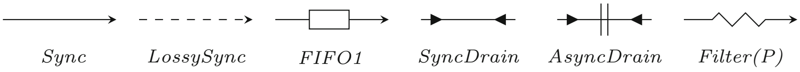


Fig. 1. Channels in Reo

Other channels are depicted in Fig. 1. Their specification is as follows:

- A LossySync channel has a similar protocol to that of a Sync channel, except that it always accepts all data items through its source end. It transfers a data item if it can be accepted by the sink end. Otherwise, the data item is consumed at the source end but lost by the channel.
- A FIFO_1 channel acts as a one-place buffer. It thus has two states: empty or full. When empty, the buffer receives a data item from its source end and changes its state to full. In that state, no more data can be received from its source end but the channel can transfer the previously received data item to its sink end, resetting thereby its state to empty.
- A SyncDrain channel is composed of two source ends. It synchronizes them: data flows if and only if the ends are ready to input it. As a result, since there is no sink end, the data is lost.
- An AsyncDrain channel is an asynchronous version that accepts data items through its source ends and loses them exclusively one at a time but never simultaneously.
- A Filter channel Filter(P) acts as a Sync channel for data items belonging to P but as the lossy part of a LossySync channel for data items not belonging to P. In other terms, it accepts a data item of P through its source end iff it can simultaneously be output at its sink end. Moreover, it always accepts all data items not in P through its source end but loses them immediately.

Complex connectors are built from the elementary channels by plugging their ends into nodes. Reo specifies three kinds of nodes, according to the channel ends that coincide on the nodes. A node is either a *source node* if all the channel ends are source ends, a *sink node* if all the channel ends are sink ends or a *mixed node* if the channel ends are a combination of source and sink ends. The three kinds of nodes are depicted in Fig. 2.

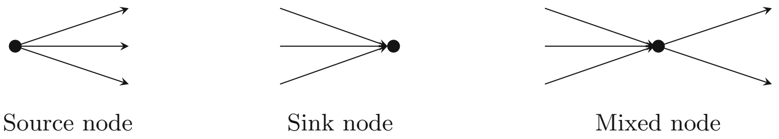


Fig. 2. Reo nodes

The semantics attached to the nodes is as follows. A source node acts as a synchronous replicator. Hence a data item put in it is duplicated to all the source ends of the channels. A sink node acts as a non-deterministic merger: a data item is output when one of the channel ends offers it. A mixed node nondeterministically selects and takes a data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends.

Mixed nodes play the role of hidden nodes necessary to make the complex connector work. In constrast, source nodes and sink nodes are interfaces with the environment of a complex connector. They are also called *boundary nodes*. Processes (or components, actors, agents) connect to them. In Reo philosophy, this amounts to attach ports of processes to them, one port being only able to connect to one node at a time. At ports linked to source nodes, processes perform blocking write operations, writing a data item to the port when the source node is ready to accept it. At ports linked to sink nodes, processes perform blocking read operations, taking a data item when the node is ready to output it.

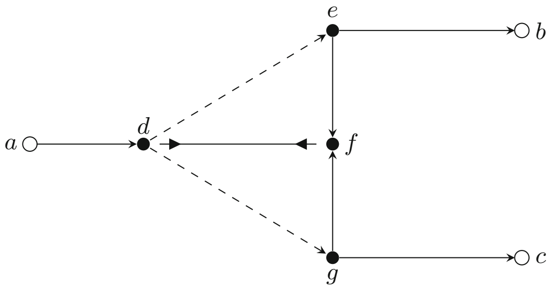


Fig. 3. An exclusive router in Reo

Following [2], Fig. 3 describes a complex connector representing an exclusive router, which transfers data items from a to either b or c but not to both. There a is a source node and b and c are sink nodes. They constitute the boundary nodes of the router. Nodes d, e, f and g are mixed nodes used to compose different primitive channels.

The semantics of the connector can in principle be deduced from the semantics attached to the primitive channels and to the nodes. Several semantics have actually been proposed to that end (see [12–14,27,28,39]). The first one and perhaps the most intuitive one is based on a coinductive calculus developed by Arbab and Rutten [4]. It is briefly described subsequently.

2.2 Timed Data Stream Semantics

The timed data stream semantics for Reo relies on the idea of describing scenarios of data being input or output over time at each connector end. Concretely, for a data end, this amounts to considering pairs of the form $<\alpha, a>$ where α is a sequence of data items and a is a sequence of positive real numbers. As observed by Arbab and Rutten in [4], a slightly less general model can be built by preserving most of the results by using natural numbers (or equivalently discrete time) instead of real numbers (typically corresponding to continuous time). For the purposes of this paper, we shall use this slightly more restricted version of using integers, which allows to perceive computations in the two-phase approach used by languages such as Esterel [7]. According to this view, computations proceed through steps of time units in which all possible actions are first computed before time passes one unit.

As a result, *timed data streams* are defined as elements of $TDS = D^\omega \times TS$ where D is the set of data items to be communicated and TS is the set of sequences s of positive integers which are strictly increasing (ie $s(i) < s(i+1)$, for all i) and progressive in the sense that, for all $N > 0$ there exists $n > 0$ such that $a(n) > N$. A timed data stream $<\!\alpha, a\!>$ for a channel end specifies that the data item $\alpha(i)$ is input or output, according to the nature of the end, at time $a(i)$. Elementary channels then appear as binary relations between TDS. To explain them, we shall use the following notation: for any sequence s, the construct s' denotes that sequence without its first element. Moreover, $s < t$ denotes the pointwise extension of the order between the elements of the sequences s and t, namely $s < t$ iff $s(n) < t(n)$ for any n.

- The Sync channel is defined, for all timed data streams $<\!\alpha, a\!>$ and $<\!\beta, b\!>$ by $<\!\alpha, a\!>\, \mathsf{Sync} <\!\beta, b\!> \equiv \alpha = \beta$ and $a = b$
- The LossySync channel is defined, for all timed data streams $<\!\alpha, a\!>$ and $<\!\beta, b\!>$ by

$$\begin{aligned} &<\!\alpha, a\!>\, \mathsf{LossySync} <\!\beta, b\!> \\ &\quad \equiv \begin{cases} \alpha(0) = \beta(0) \text{ and } <\!\alpha', a'\!>\, \mathsf{LossySync} <\!\beta', b'\!> & \text{if } a(0) = b(0) \\ <\!\alpha', a'\!>\, \mathsf{LossySync} <\!\beta, b\!> & \text{if } a(0) < b(0) \end{cases} \end{aligned}$$

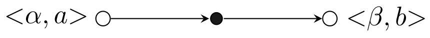
- The FIFO_1 channel is defined, for all timed data streams $<\!\alpha, a\!>$ and $<\!\beta, b\!>$ by $<\!\alpha, a\!>\, \mathsf{FIFO}_1 <\!\beta, b\!> \equiv \alpha = \beta$ and $a < b < a'$
- The SyncDrain channel is defined, for all timed data streams $<\!\alpha, a\!>$ and $<\!\beta, b\!>$ by $<\!\alpha, a\!>\, \mathsf{SyncDrain} <\!\beta, b\!> \equiv a = b$
- The ASyncDrain channel is defined, for all timed data streams $<\!\alpha, a\!>$ and $<\!\beta, b\!>$ by $<\!\alpha, a\!>\, \mathsf{ASyncDrain} <\!\beta, b\!> \equiv a \bowtie b$ where $a \bowtie b$ is defined as

$$a \bowtie b \equiv a(0) \neq b(0) \text{ and } \begin{cases} a' \bowtie b \text{ if } a(0) < b(0) \\ a \bowtie b' \text{ if } b(0) < a(0) \end{cases}$$

- The Filter(P) channel is defined, for all timed data streams $<\!\alpha, a\!>$ and $<\!\beta, b\!>$ by

$$\begin{aligned} &<\!\alpha, a\!>\, \mathsf{Filter(P)} <\!\beta, b\!> \\ &\quad \equiv \begin{cases} \alpha(0) = \beta(0) \text{ and } a(0) = b(0) \text{ and } <\!\alpha', a'\!>\, \mathsf{Filter(P)} <\!\beta', b'\!> & \text{if } \alpha(0) \in P \\ a(0) < b(0) \text{ and } <\!\alpha', a'\!>\, \mathsf{LossySync} <\!\beta, b\!> & \text{otherwise} \end{cases} \end{aligned}$$

Complex connectors are built from more elementary ones by means of mixed nodes, which thus appear as the composition of relations. For instance, the composition of two copies of the synchronous channels



yields the following binary relation R:

$$<\alpha, a>R<\beta, b> \equiv \exists <\gamma, c> : <\alpha, a>R<\gamma, c> \wedge <\gamma, c>R<\beta, b>$$

More complex mixed nodes involve m "input" channels and n "output" channels, the data items of the first one being merged and the data items of the last ones being replicated. As one may expect, the semantics of the merge of m data flows amounts to $m-1$ compositions of the merge of two data flows, while the semantics of the replication of n data streams amounts to the composition of $n-1$ replications of two data flows. These binary merge and replication operators are themselves defined as follows:

- The (binary) Merge of two input ends into one output end is defined for all timed data streams $<\alpha, a>$, $<\beta, b>$ and $<\gamma, c>$ by

$$\begin{aligned} &\mathsf{Merge}(<\alpha, a>, <\beta, b>, <\gamma, c>) \\ &\equiv a(0) \neq b(0) \wedge \begin{cases} \gamma(0) = \alpha(0) \wedge a(0) = c(0) \\ \qquad \wedge\ \mathsf{Merge}(<\alpha', a'>, <\beta, b>, <\gamma', c'>) \text{ if } a(0) < b(0) \\ \gamma(0) = \beta(0) \wedge b(0) = c(0) \\ \qquad \wedge\ \mathsf{Merge}(<\alpha, a>, <\beta', b(>, <\gamma', c'>) \text{ if } b(0) < a(0) \end{cases} \end{aligned}$$

- The (binary) Replicate of one input end into two output ends is defined for all timed data streams $<\alpha, a>$, $<\beta, b>$ and $<\gamma, c>$ by

$$\mathsf{Replicate}(<\alpha, a>, <\beta, b>, <\gamma, c>) \equiv \beta = \alpha \wedge \gamma = \alpha \wedge b = a \wedge c = a$$

As an example of the composition of the elementary channels with respect to the mixed nodes, the semantics of the exclusive router presented in Fig. 3 is defined by the following relation:

$$\begin{aligned} &XRout(<\alpha, a>, <\beta, b>, <\gamma, c>) \\ &\equiv \begin{cases} \beta(0) = \alpha(0) \wedge b(0) = a(0) \wedge XRout(<\alpha', a'>, <\beta', b'>, <\gamma, c>) \text{ if } b(0) < c(0) \\ \gamma(0) = \alpha(0) \wedge c(0) = a(0) \wedge XRout(<\alpha', a'>, <\beta, b>, <\gamma', c'>) \text{ if } c(0) < b(0) \end{cases} \end{aligned}$$

2.3 The ReoD Language

It will be helpful later to describe Reo connectors by means of a language. Taking inspiration from the Latex package `reotex` and other textual representation of Reo such as RSL, we define the ReoD calculus as follows.

Definition 1. *Let Data, SN and SBP be three denumerable disjoint sets, the elements of which are respectively called data items, node names and channel names.*

- *Define the node instructions as the statements* `ionode(n)` *and* `mixednode(n)`*, where* `n` *denotes a node name.*

- *Define the channel instructions as the statements of the form `Sync(c,n1,n2)`, `LossySync(c,n1,n2)`, `Fifo1(c,n1,n2)`, `SyncDrain(c,n1,n2)`, `AsyncDrain (c,n1,n2)`, `Filter(c,n1,n2,P)`, where* `c` *is a channel name,* `n1`*,* `n2` *are node names and* `P` *is a list of data elements.*
- *Define* ReoD *as the set of agents formed from a list of node instructions, each one describing a different node name, followed by a list of channel instructions, the node names of which being all described in the list of node instructions.*

As an example, the exclusive router discussed above is described by the following ReoD agent:

```
ionode(a), mixednode(d), mixednode(e), mixednode(g), mixednode(f),
ionode(b), ionode(c), sync(ad,a,d), lossysync(dg,d,g),
lossysync(de,d,e), syncdrain(df,d,f), sync(gf,g,f), sync(ef,e,f),
sync(gc,g,c), sync(eb,e,b)
```

3 A Family of Data-Based Coordination Languages

The authors have developed extensions of the data-driven language Linda. Some of them are reviewed in this section to provide the background needed to translate ReoD. Guided by this purpose, we shall use tokens instead of more general structured tuples.

3.1 The Language BachT

We first consider a simplified version of a dialect of Linda, developed at the University of Namur, named Bach (see [25]), and more precisely its restriction to tokens, which we shall name BachT. This restricted language is based on four primitives for accessing a shared space, called *the store*, here seen as a multiset of tokens. Formally, the language is defined as follows.

Definition 2. *Let* Stoken *be an enumerable set, the elements of which are subsequently called tokens and are typically represented by the letters t and u. Define the set of stores* Sstore *as the set of finite multisets with elements from* Stoken.

Definition 3. *Define the set $\mathcal{T}$ of the token-based primitives as the set of primitives T_b generated by the following grammar:*

$$T_b ::= tell(t) \mid ask(t) \mid get(t) \mid nask(t)$$

where t represents a token.

Definition 4. *Let Spvar be a denumerable set disjoint from the set of tokens and let us name their elements procedure variables. Define the* BachT *language as the set of agents A generated by the following grammar:*

$$A ::= T_b \mid X \mid A \; ; \; A \mid A \; || \; A \mid A \; + \; A$$

where T_b represents a token-based primitive, where X is a procedure variable and where " ; ", " || " and " + " denote the sequential, parallel and choice compositional operators. Define the set of guarded agents of BachT *as the agents generated by the following grammar:*

$$G ::= T_b \mid G \,;\, A \mid G \,||\, G \mid G \,+\, G$$

where A denotes a BachT *agent. As usual in concurrency theory, we shall assume that each procedure variable X is associated with a guarded agent by a set of declarations of the form $X = G$.*

3.2 The Language MRT

The second language we shall consider is a Gamma-like language (see [5,6]), based on the chemical reaction metaphor. It considers communication primitives as the rewriting of pre-condition multi-sets into post-condition multi-sets. Intuitively, the operational effect of a multi-set rewriting (*pre*, *post*) consists in inserting all the positive post-conditions, and in deleting all the negative post-conditions from the current store σ, provided that σ contains all positive pre-conditions and does not meet any of the negative pre-conditions. Formally, these rewritings are specified as follows.

Definition 5. *Define the set of multi-set rewriting primitives $\mathcal{T}_{MR}$ as the set of primitives T_{MR} generated by the following grammar:*

$$\begin{aligned} T_{MR} &::= (\{M\}, \{M\}) \\ M &::= \lambda \mid \; +t \mid \; -t \mid M, M \end{aligned}$$

where λ indicates an empty multi-set and where t denotes a token.

It is worth observing that not all pairs of preconditions and postconditions correspond to reasonable computations. Indeed, as stated above, it is possible to require in a precondition that the same token is present and absent or to require in the postcondition the removal of a token which has not been tested for presence in the precondition. We subsequently define such reasonable pairs of pre- and post-conditions as those being respectively consistent and valid. To that end, we first introduce some notations.

Definition 6. *Given a multi-set rewriting pair (Pre, Post), denote by Pre^+ the multi-set $\{t \mid +t \in Pre\}$ of tokens positively appearing in the precondition and by Pre^- the multi-set $\{t| -t \in Pre\}$ negatively appearing in it. Similarly, we shall denote by $Post^+$ and $Post^-$ the multiset of tokens appearing positively and negatively in the postcondition.*

A multi-set rewriting pair (Pre, Post) is said to be consistent if $Pre^+ \cap Pre^- = \emptyset$. It is said to be valid if $Post^- \subseteq Pre^+$.

Definition 7. *In this context, the language* MRT *is defined as* BachT *in Definition 4 by taking multi-set primitives instead of token-based primitives.*

3.3 The Language VBachT

In an attempt to provide BachT with the same property of MRT of handling many tokens at once, a natural extension consists in replacing in the primitives of BachT a token by a list of tokens. For instance, the primitive $ask(t, u, v)$ would succeed on a store containing one occurrence of t, one of u and one of v. Dually, the computation of $tell(t, u, v)$ would result in adding on the store one occurrence of t, one of u and one of v. This is formalized by the following definitions.

Definition 8. *Define a vector of tokens as a list* $t_1, \cdots, t_n$ *of tokens. Such a vector is subsequently denoted as* $\overrightarrow{t}$. *Define SVtoken as the set of vectors of tokens.*

With this definition in mind, a natural extension of BachT consists in lifting the arguments of the primitives to vectors of tokens.

Definition 9. *Define the set of vectorized token-based primitives* $\mathcal{T}_v$ *as the set of primitives* T_v *generated by the following grammar:*

$$T_v ::= tell(\overrightarrow{t}) \mid ask(\overrightarrow{t}) \mid get(\overrightarrow{t}) \mid nask(\overrightarrow{t})$$

where $\overrightarrow{t}$ *represents a vector of tokens.*

Definition 10. *Define the Vectorized Bach language* VBachT *similarly to Definition 4 of the BachT language but by taking vector of token-based primitives* T_v.

3.4 Transition System

To study possible links between ReoD and the three languages we just introduced, their semantics needs to be defined. To that end, we shall use an operational one in the style of Plotkin [36], based on a transition system. The configurations to be considered consist of an agent, summarizing the current state of the agents running on the store, and a multi-set of tokens, denoting the current state of the store. In order to express the termination of the computation of an agent, we extend the set of agents by adding a special terminating symbol E that can be seen as a completely computed agent. For uniformity purpose, we abuse language by qualifying E as an agent. To meet the intuition of this terminating agent E, we shall always rewrite agents of the form $(E; A)$, $(E \mid\mid A)$ and $(A \mid\mid E)$ as A.

Figure 4 specifies the transition rules for the primitives of the BachT language. The first rule (T) expresses that an atomic agent *tell(t)* can be executed in any store σ, and that its action has the effect of adding the token t to the same store. The second rule (A) states that an atomic agent *ask(t)* can be executed in any store σ containing the token t, however leaving the store σ unaltered after its execution. The third rule (G) works similarly to the previous rule (A), but with the difference of retrieving the token t initially present on the store σ after the execution of the agent *get(t)*. Finally, the fourth rule (N) establishes that an atomic agent *nask(t)* can be executed in any store σ not containing the token t, leaving the store σ unaltered after its execution. Note that, as no rules are provided in these cases, the *ask(t)*, *get(t)* and *nask(t)* primitives are assumed to suspend as long as the corresponding presence/absence of token is not met.

$$(\mathbf{T}) \qquad \langle\ tell(t) \mid \sigma\ \rangle \longrightarrow \langle\ E \mid \sigma \cup \{t\}\ \rangle$$

$$(\mathbf{A}) \quad \langle\ ask(t) \mid \sigma \cup \{t\}\ \rangle \longrightarrow \langle\ E \mid \sigma \cup \{t\}\ \rangle$$

$$(\mathbf{G}) \qquad \langle\ get(t) \mid \sigma \cup \{t\}\ \rangle \longrightarrow \langle\ E \mid \sigma\ \rangle$$

$$(\mathbf{N}) \qquad \frac{t \not\in \sigma}{\langle\ nask(t) \mid \sigma\ \rangle \longrightarrow \langle\ E \mid \sigma\ \rangle}$$

Fig. 4. Transition rules for token-based primitives (BachT)

$$(CM) \qquad \frac{\begin{array}{c} pre^{+} \subseteq \sigma,\ pre^{\,-} \perp \sigma, \\ \sigma' = (\ \sigma \setminus\ post^{-}) \cup post^{+} \end{array}}{\langle (pre,\ post) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma' \rangle}$$

Fig. 5. Transition rules for multi-set rewriting-based primitives (MR)

For MRT, it turns that it is possible to define it by one rule. To express it, an auxiliary notion is however needed. It extends the notation of Definition 6 to capture the fact that, for each token, the tokens mentioned negatively in the definition are not with their multiplicity on the current store σ.

Definition 11. *For any token t, define $Pre^{-}[t]$ as the multiset of negatively marked tokens t in the precondition Pre: $[Pre^{-}[t] = \{t : -t \in Pre^{-}\}$. Given a precondition Pre and a store σ, we then define the non element-wise inclusion operator $\perp$ as follows: $Pre^{-} \perp \sigma$ iff $Pre^{-}[t] \not\subseteq \sigma$, for any token t.*

With this notation, rule (CM) of Fig. 5 states that a multi-set rewriting $(Pre, Post)$ can be executed in a store σ if the multi-set Pre^{+} is included in σ and if no negative pre-condition occurs with the required multiplicity in σ. Under these conditions, the effect of the rewriting is to delete from σ all the negative post-conditions and to add to σ all the positive post-conditions.

$$(\mathbf{T_v}) \qquad \langle\ tell(t_1, \ldots, t_n) \mid \sigma\ \rangle \longrightarrow \langle\ E \mid \sigma \cup \{t_1, \ldots, t_n\}\ \rangle$$

$$(\mathbf{A_v}) \quad \langle\ ask(t_1, \ldots, t_n) \mid \sigma \cup \{t_1, \ldots, t_n\}\ \rangle \longrightarrow \langle\ E \mid \sigma \cup \{t_1, \ldots, t_n\}\ \rangle$$

$$(\mathbf{G_v}) \qquad \langle\ get(t_1, \ldots, t_n) \mid \sigma \cup \{t_1, \ldots, t_n\}\ \rangle \longrightarrow \langle\ E \mid \sigma\ \rangle$$

$$(\mathbf{N_v}) \qquad \frac{t_1 \not\in \sigma, \ldots, t_n \not\in \sigma}{\langle\ nask(t_1, \ldots, t_n) \mid \sigma\ \rangle \longrightarrow \langle\ E \mid \sigma\ \rangle}$$

Fig. 6. Transition rules for vectorized token-based primitives (VBachT)

Figure 6 provides the transitions for the vectorized token-based primitives. Rule (T_v), (A_v) and (G_v) generalize the corresponding rules (T), (A) and (G) from tokens to vectors of tokens. As a result, rule (T_v) asserts that telling a vector of tokens amounts to adding each of them. Similarly, rule (A_v) requires for an ask primitive to succeed the presence of each token t_i. According to rule (G_v) the behavior of a get primitive performs such a test for presence but also removes one occurrence of each t_i on the store. Finally, rule (N_v) requires, for each token t_i, its absence on the store.

$$(\mathbf{S}) \quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\langle A\ ;\ B \mid \sigma\rangle \longrightarrow \langle A'\ ;\ B \mid \sigma'\rangle}$$

$$(\mathbf{P}) \quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle A \mid\mid B \mid \sigma\rangle \longrightarrow \langle A' \mid\mid B \mid \sigma'\rangle \\ \langle B \mid\mid A \mid \sigma\rangle \longrightarrow \langle B \mid\mid A' \mid \sigma'\rangle\end{array}}$$

$$(\mathbf{C}) \quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle A\ +\ B \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle \\ \langle B\ +\ A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle\end{array}}$$

$$(\mathbf{X}) \quad \frac{X = G,\ \ \langle G \mid \sigma\rangle \longrightarrow \langle G' \mid \sigma'\rangle}{\langle X \mid \sigma\rangle \longrightarrow \langle G' \mid \sigma'\rangle}$$

Fig. 7. Transition rules for the operators

Figure 7 details the usual rules for sequential composition, parallel composition, interpreted in an interleaved fashion, non-deterministic choice, and procedure variables.

3.5 Expressiveness

Following previous work by the authors (see [9–11,32–34]) and based on the modular embedding introduced by de Boer and Palamidessi in [19], Darquennes' PhD thesis [15] has studied the expressiveness of the languages BachT, MRT and a slightly more general version of VBachT. In a snapshot, BachT has been proved stricly less expressive than VBachT which is itself strictly less expressive than MRT. However, the increase of expressiveness comes with a price of implementation: BachT allows for more efficient implementations than VBachT, which itself is more efficiently implementable than MRT.

4 Translating ReoD in Bach

4.1 First Observations

The BachT, VBachT and MRT languages provide means to describe ReoD channels at two levels. Before going to them, it is first worth observing that, apart from the filter channel, the behavior of all the other channels does not depend upon the data item which is exchanged. In other terms, if the data streams $<\alpha, a>$ and $<\beta, b>$ are in the relation R associated with a connector and if γ and δ are the same renaming of data items in α and β, then $<\gamma, a>$ and $<\delta, b>$ are also in R. Coming back to the intuition behind Reo, this is not very surprising. Indeed, in constrast to BachT, VBachT and MRT which rely on the availability of information, Reo is essentially concerned with the control of the synchronization of channels.

As our purpose is to relate ReoD and the BachT family of languages, this remark allows us to restrict subsequently the set of data items $Data$ to one element, say t.

4.2 Specification

It turns out that the MRT language allows for a process algebra specification of ReoD from an external point of view, namely from the point of view of the environment which does not care for the internal details of the connectors.

To expose it, let us introduce an auxiliary notation. Consider a timed data stream $<\gamma, c>$ at node n. Then for any $i \in N$, we denote by $MS_i(<\gamma, c>, n)$ the following set:

$$MS_i(<\gamma, c>, n) = \begin{cases} \{\gamma(k)@n\} & \text{if there is k such that } c(k) = i \\ \emptyset & \text{otherwise} \end{cases}$$

where $\gamma(k)@n$ is to be considered as a token (named after the data $\gamma(k)$ and the node n). Note that by the definition of data streams, if it exists, for any positive integer, there is only one k such that $c(k) = i$.

Let us now consider a (possibly complex) connector composed of $in_1, \ldots, in_m$ as input nodes and $on_1, \ldots, on_n$ as output nodes. Assume that $<\alpha_1, a_1>, \ldots,$

$<\alpha_m, a_m>$, $<\beta_1, b_1>$, …, $<\beta_m, a_m>$ are data streams of R associated respectively with in_1, …, in_m and on_1, …, on_n. Then let us define for any $i \in N$,

$$Pre_i = MS_i(<\alpha_1, a_1>, in_1) \cup \cdots \cup MS_i(<\alpha_m, a_m>, in_m)$$
$$Post_i = MS_i(<\beta_1, b_1>, on_1) \cup \cdots \cup MS_i(<\beta_n, b_n>, on_n)$$

It is easy to observe that the MRT agent $(Pre_t, Post_t)$ computes what can be observed from the data stream at time t. Namely, assuming the presence of tokens in the input nodes, as specified by Pre_t (these tokens being provided by the environment), the computation produces the tokens in the output nodes, as specified by $Post_t$ (these tokens being consumed by the environment).

As a result, the MRT sequential composition

$$(Pre_0, Post_0); (Pre_1, Post_1); \cdots ; (Pre_i, Post_i); \cdots$$

computes what can be observed by the data streams. As an alternative operational characterization of the connector, using the reactive semantics proposed by [20], one may claim that

$$(Pre_0, Post_0).(Pre_1, Post_1). \cdots .(Pre_i, Post_i). \cdots$$

is a history computed by the connector, the hole between each step $(Pre_i, Post_i)$ being made by the environment.

It is also worth observing that as any connector in ReoD is formed from a finite number of elementary channels and as we reduce $Data$ to one data item, the above construction can only produce a finite number of Pre and $Post$ sets. As a result, following well-known results in process algebra (see for instance [21]), connectors can be described by linear recursive equations involving MRT primitives as actions. For instance, the exclusive router of Fig. 3 can be described by the following recursive equation:

$$XRout = (\{t@a\}, \{t@b\}); XRout + (\{t@a\}, \{t@c\}); XRout$$

4.3 Implementation

The specification is useful to describe the behaviour of a connector from the point of view of the environment but does not give much insight into the internal computations. To that end, we shall employ BachT and VBachT.

Two notations are required to that end. First, following the definition of the ReoD-calculus (see Sect. 2.3), each elementary channel end may be associated with the name of the channel and the name of the node to which the end is connected. In view of that, we shall associate with the data item t a token named as $t@c{:}n$ where c is the name of the channel and n the name of the node. Intuitively, the presence of the token in the store represents the presence of the data item at the end referenced by the node n and the channel name c. Second, in a similar way and as in the previous subsection, we shall employ $t@n$ to denote the presence of the data item t at node n.

As suggested in Sect. 2, looking internally into the computation of the connector is achieved in the two-phase functioning approach, according to which data flow is first performed inside the connector in the first phase and then time progresses in the second phase. To grasp this in the BachT and VBachT languages, we enrich them with a new primitive named next. Its operational semantics is provided by adding a new transition relation $\rightsquigarrow$ specified in Fig. 8.

$$(\mathbf{N_1}) \qquad \langle\ next \mid \sigma\ \rangle \rightsquigarrow \langle\ E \mid \sigma\rangle$$

$$(\mathbf{N_2}) \quad \frac{\langle A \mid \sigma\rangle \rightsquigarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c} \langle A\ ;\ B \mid \sigma\rangle \rightsquigarrow \langle A'\ ;\ B \mid \sigma'\rangle \\ \langle A\ +\ B \mid \sigma\rangle \rightsquigarrow \langle A'\ +\ B \mid \sigma'\rangle \\ \langle B\ +\ A \mid \sigma\rangle \rightsquigarrow \langle B\ +\ A' \mid \sigma'\rangle \\ \langle A \mid\mid B \mid \sigma\rangle \rightsquigarrow \langle A' \mid\mid B \mid \sigma'\rangle \\ \langle B \mid\mid A \mid \sigma\rangle \rightsquigarrow \langle B \mid\mid A' \mid \sigma'\rangle \end{array}}$$

Fig. 8. Transition rules for the next operator

The operational semantics of an agent of BachT and VBachT is then obtained by alternating the → transitions, performed as much as possible and representing the first phase, and the $\rightsquigarrow$-transitions, also performed as much as possible and representing the second phase.

We are now in a position to translate the connectors described in ReoD. This is done by associating an agent with each (elementary) channel as well as with each node. We start with the translation of the channels. Consider a channel of name c associated with the nodes named n_1 and n_2. Depending upon its form, it is associated with the following agent:

- the `Sync(c,n1,n2)` channel is associated with the procedure variable `sync_c` defined by `sync_c = get(t@c:n1); tell(t@c:n2); next; sync_c`
- the `LossySync(c,n1,n2)` channel is associated with the procedure variable `lossy_sync_c` defined by

```
lossy_sync_c = (get(t@c:n1); tell(t@c:n2); next; lossy_sync_c
               + (get(t@c:n1); next; lossy_sync_c)
```

- the `Fifo1(c,n1,n2)` channel is associated with the procedure variable `Fifo1_c_e` defined by

```
Fifo1_c_e = get(t@c:n1); next; Fifo1_c_f
Fifo1_c_f = tell(t@c:n2); next; Fifo1_c_e
```

- the `SyncDrain(c,n1,n2)` channel is associated with the procedure variable `sync_drain_c` defined by `sync_drain_c = get(t@c:n1, t@c:n2); next; sync_drain_c`

- the AsyncDrain(c,n1,n2) channel is associated with the procedure variable async_drain_c defined by

  ```
  async_drain_c = (get(t@c:n1); next; async_drain_c)
                + (get(t@c:n2); next; async_drain_c)
  ```

- the Filter(c,n1,n2,P) channel is associated with the procedure variable filter_c defined by one of the following equations, according to the fact that $t \in P$ or not

  ```
  filter_c = get(t@c:n1); tell(t@c:n2); next; filter_c
  filter_c = get(t@c:n1); next; filter_c
  ```

The behavior of nodes is defined similarly. We shall assume that the environment is in charge of providing the tokens in the sink nodes and that the computation of the connector is in charge of producing the tokens in the source nodes. This given, the behavior of a mixed node n is to take one token from a coincident sink channel end and to duplicate it to all its coincident source channel ends. Assume that the coincident sink channel ends of node n are $i_1, \ldots, i_p$ for channel named $c_1, \ldots c_p$, respectively, and that the coincident source channel ends of node n are $o_1, \ldots, o_q$ for channel named $d_1, \ldots d_q$, respectively, then the mixed node can be coded by the procedure variable mixed_node_n defined as

```
mixed_node_n = ( get(t@c1:i1) + ... + get(t@cp:ip) );
                 tell(t@d1:o1,...,t@dq:oq); next; mixed_node_n
```

The behavior of a sink node is similar, except that it takes the token placed by the environment: assuming the coincident source channel ends of the sink node n are $o_1, \ldots, o_q$ for channel named $d_1, \ldots d_q$, respectively, then its behavior is coded by the procedure variable sink_node_n defined as

```
sink_node_n = get(t@n); tell(t@d1:o1,...,t@dq); next; sink_node_n
```

Finally, source nodes act dually by taking one of the token present at the end of a coincident sink channel end: assuming the coincident sink channel ends of source node n are $i_1, \ldots, i_p$ for channel named $c_1, \ldots c_p$, then its behavior is coded by the procedure variable sink_node_n defined as

```
source_node_n = ( get(t@c1:i1) + ... + get(t@cp:ip) );
                  tell(t@n); next; source_node_n
```

Summing up, the VBachT agent associated with a complex connector is the agent formed by the parallel composition of the agent associated (as described above) to the channels and nodes. If the connector is named C, we shall denote by *vbacht*(C) the VBachT agent associated by the reasoning of this section.

4.4 Correctness

A natural question to ask is the correctness of $\mathit{vbacht}(C)$ with respect to the specification of C. To discuss this point, we introduce the following definition.

Definition 12. *1. For any connector C whose sink nodes are i_1, ..., i_p, we define the set of initial configurations of C as the set of subsets formed from $\{t@i_1, \cdots, t@i_p\}$.*

2. For any connector C, the store σ is said to be node-free if contains no tokens $t@n$ corresponding to a source node and no token $t@c : n$ corresponding to the end of a channel.

3. For any connector C and any VBachT *agent A, a history*

$$h = (Pre_0, Post_0).(Pre_1, Post_1). \cdots .(Pre_i, Post_i). \cdots$$

of C is said to be computable by A if there is a series of transitions such that

$$\begin{array}{l} \langle A \mid Pre_0 \rangle \rightarrow^* \langle B_0 \mid Post_0 \rangle \rightsquigarrow^* \langle A_1 \mid Post_0 \rangle \\ \cdots \\ \langle A_i \mid Pre_i \rangle \rightarrow^* \langle B_i \mid Post_i \rangle \rightsquigarrow^* \langle A_{i+1} \mid Post_i \rangle \\ \cdots \end{array}$$

with $\langle B_i \mid Post_i \rangle \not\rightarrow$ for any i and $\langle A_{i+1} \mid Post_i \rangle \not\rightsquigarrow$ for any i.

4. For any connector C and any VBachT *agent A, A is said to be coherent with C if any history of C is computable by A.*

It is easy to observe that, for any connector C, the agent $\mathit{vbacht}(C)$ is coherent with C. Restated in other terms, for any history observed as an external behavior of C, there is a computation of $\mathit{vbacht}(C)$ which computes it. However, $\mathit{vbacht}(C)$ exhibits non-desirable computations. This is in particular due to the coding of the LossySync channel which is not context-dependent. Indeed, in our coding, LossySync channel is allowed to deliver data items at a channel end even if this end is not ready to accept it and thereby to create a store which is not node-free.

This is actually a well-known problem in the Reo community. Fortunately, Proença et al. have shown in [14,37] that, by using constraint satisfaction techniques, it is possible to pre-compute the paths of suitable executions of any Reo connector. For instance, for the exclusive router, provided a data item is present in node a, two paths may be executed: ad-de-df-eb with dg acting lossy or ad-df-dg-gf-gc with de acting lossy. This given, real-life executions may be obtained by, in view of the data items put in source nodes, telling tokens for the execution of the channels corresponding to the corresponding path and guarding the execution of the channels by the suitable tokens. Along these lines, let us code these tokens from the names of the channels suffixed with `act`, for an active channel, `nact` for a non active channel, `lost`, for an active LossySync channel loosing the data item and, `nlost` for an active LossySync channel not loosing the data item. Then to get an execution of the first path of the exclusive router, the following tokens should be put on the store by a multiple tell at the beginning of the first phase:

```
tell(ad_act, de_nlost, df_act, eb_act, dg_lost, gf_nact, gc_nact)
```

This given, the behavior of the LossySync channel *de*, for instance, can be coded as

```
lossy_sync_de =
   get(de_nlost); get(t@de:n1); tell(t@de:n2); next; lossy_sync_de
 + get(de_lost); get(t@de:n1); next; lossy_sync_de
 + get(de_nact); next; lossy_sync_de
```

while the behavior of the Sync channel *gc* can be coded as

```
sync_gc =
   get(gc_act); get(t@gc:n1); tell(t@gc:n2); next; sync_gc
 + get(gc_nact); next; sync_gc
```

It is also worth observing that the `next` statement can be coded through the store by having each process first telling a token indicating its intention to proceed to the next step and then getting a token provided by a coordinator to allow it to actually do so. Dually, the coordinator first collects the process intentions to go to the next step by a multiple get and then tells them to do so by a multiple tell putting the permissions to do so on the store.

4.5 Restriction on *Data*

The observation that the behavior of channels does not depend upon the actual data items being exchanged has been made in Sect. 4.1 and has resulted in considering *Data* to be composed of just one data item t. In the presence of various `Filter(P)` channels, it is interesting to lift this hypothesis to tackle a finite number of tokens, for instance to provide, for each predicate, a token that satisfies it and another one that does not. This is actually quite easy to achieve by duplicating through choices what we have produced for the token t. For instance, for $Data = \{t_1, \cdots, t_m\}$, the translation of the Sync(c,n1,n2) becomes

```
sync_c = get(c_act);
             ( get(t1@c:n1); tell(t1@c:n2); next; sync_c
               + ... +
               get(tm@c:n1); tell(tm@c:n2); next; sync_c )
         + get(c_nact); next; sync_c
```

5 Conclusion

The paper has aimed at providing links between two coordination worlds, which are generally considered in isolation: the control-driven and data-driven families of languages. To that end, we have proposed a translation of the key features of the Reo model in two data-driven languages MRT and VBachT studied by the authors. This has been done at two levels. First, from the point of view of the specification of ReoD descriptions, we have shown how MRT constructs can be used to define the timed data stream semantics of connectors. Second, from a more computational point of view, we have shown how connectors and nodes of ReoD can be encoded in VBachT.

The correctness of the translation has also been addressed. We have shown that the translation of a connector in VBachT is coherent with the external perception given by the MRT description of the considered connector. However, we have also shown that the translation in VBachT produces non desirable computations. In order to avoid them, we have then shown how the results of [14,37] can be used to adapt our translation.

To the best of our knowledge, although many pieces of work have addressed tool support for Reo (see for instance [3,29,31,37]) and [38] has presented an automaton model that capture the semantics of Reo and Linda, our work is the first to address the translation of Reo in a data-driven language. It shares similarities with [30,31] where a translation of Reo in mCRL2 is provided. For instance, in [30,31], processes running in parallel are also associated with channels and nodes. However, mCRL2 being synchronous, the synchronization of data is addressed differently through actions sharing arguments, hiding and blocking operators. This allows the authors to capture a correct semantics of the LossySync channel by reusing the colouring framework of [12]. In contrast, our work relies on asynchronous communication through information being shared by the deposit of tokens. Moreover a correct behavior of the LossySync channel is obtained through the constraint satisfaction framework of [14,37]. Finally, the external behavior of connectors is not addressed [30,31].

Our work suggests directions for future work such as an extension to tackle other connectors, involving time and quantitative models. Moreover it would be interesting to characterize how so-called synchronous regions [40] can be detected in our translation scheme. Finally the translation has been provided from Reo to BachT-like languages. Exploring the converse translation would be interesting as well.

Acknowledgment. We thank the anonymous reviewers for their comments and suggestions. We also wish to Farhad a wonderful retirement and thank him for what he has done, directly or indirectly, to promote coordination as a research field.

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
2. Arbab, F.: Puff, the magic protocol. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 169–206. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24933-4_9
3. Arbab, F., Baier, C., de Boer, F.S., Rutten, J.J.M.M.: Models and temporal logical specifications for timed component connectors. Softw. Syst. Model. **6**(1), 59–82 (2007)

4. Arbab, F., Rutten, J.J.M.M.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2002. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40020-2_2
5. Banâtre, J.-P., Le Métayer, D.: Programming by multiset transformation. Commun. ACM **36**(1), 98–111 (1993)
6. Banâtre, J.-P., Métayer, D.L.: Gamma and the chemical reaction model: ten years after. In: Coordination Programming, pp. 3–41. Imperial College Press, London (1996)
7. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**, 87–152 (1992)
8. Brogi, A., Jacquet, J.-M.: On the expressiveness of linda-like concurrent languages. Electron. Notes Theor. Comput. Sci. **16**(2), 61–82 (1998)
9. Brogi, A., Jacquet, J.-M.: On the expressiveness of coordination models. In: Ciancarini, P., Wolf, A.L. (eds.) COORDINATION 1999. LNCS, vol. 1594, pp. 134–149. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48919-3_11
10. Brogi, A., Jacquet, J.-M. (eds.): Foclasa 2002, Foundations of Coordination Languages and Software Architectures (Satellite Workshop of CONCUR 2002), vol. 68, no. 3 (2003)
11. Brogi, A., Jacquet, J.-M.: On the expressiveness of coordination via shared dataspaces. Sci. Comput. Program. **46**(1–2), 71–98 (2003)
12. Clarke, D., Costa, D., Arbab, F.: Connector colouring I: synchronisation and context dependency. Sci. Comput. Program. **66**(3), 205–225 (2007)
13. Clarke, D., Proença, J., Lazovik, A., Arbab, F.: Deconstructing Reo. Electron. Notes Theor. Comput. Sci. **229**(2), 43–58 (2009)
14. Clarke, D., Proença, J., Lazovik, A., Arbab, F.: Channel-based coordination via constraint satisfaction. Sci. Comput. Program. **76**(8), 681–710 (2011)
15. Darquennes, D.: On multiplicities in coordination languages. Ph.D. thesis, Faculty of Computer Science, University of Namur, Namur, Belgium (2017)
16. Jacquet, J.-M., Linden, I., Darquennes, D.: On density in coordination languages. In: Canal, C., Villari, M. (eds.) ESOCC 2013. CCIS, vol. 393, pp. 189–203. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45364-9_16
17. Jacquet, J.-M., Linden, I., Darquennes, D.: On the introduction of density in tuple-space coordination languages. Sci. Comput. Program. **115–116**, 149–176 (2016)
18. Darquennes, D., Jacquet, J.-M., Linden, I.: On distributed density in tuple-based coordination languages. In: Cámara, J., Proença, J. (eds.) Proceedings 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems, Rome, Italy. EPTCS, vol. 175, pp. 36–53 (2015)
19. de Boer, F., Palamidessi, C.: Embedding as a tool for language comparison. Inf. Comput. **108**(1), 128–157 (1994)
20. de Boer, F.S., Palamidessi, C.: A fully abstract model for concurrent constraint programming. In: Abramsky, S., Maibaum, T.S.E. (eds.) CAAP 1991. LNCS, vol. 493, pp. 296–319. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-53982-4_17
21. Fokkink, W.: Introduction to Process Algebra. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-662-04293-9
22. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7**(1), 80–112 (1985)
23. Gelernter, D., Carriero, N.: Coordination languages and their significance. Commun. ACM **35**(2), 97–107 (1992)

24. Jacquet, J.-M., De Bosschere, K., Brogi, A.: On timed coordination languages. In: Porto, A., Roman, G.-C. (eds.) COORDINATION 2000. LNCS, vol. 1906, pp. 81–98. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45263-X_6
25. Jacquet, J.-M., Linden, I.: Coordinating context-aware applications in mobile adhoc networks. In: Braun, T., Konstantas, D., Mascolo, S., Wulff, M. (eds.) Proceedings of the First ERCIM Workshop on eMobility, pp. 107–118. The University of Bern (2007)
26. Jacquet, J.-M., Linden, I.: Fully abstract models and refinements as tools to compare agents in timed coordination languages. Theoret. Comput. Sci. **410**(2–3), 221–253 (2009)
27. Jongmans, S.T.Q., Arbab, F.: Correlating formal semantic models of reo connectors: connector coloring and constraint automata. In: Silva, A., Bliudze, S., Bruni, R., Carbone, M. (eds.) Proceedings Fourth Interaction and Concurrency Experience, ICE 2011, Reykjavik, Iceland, 9 June 2011. EPTCS, vol. 59, pp. 84–103 (2011)
28. Jongmans, S.-S.T.Q., Krause, C., Arbab, F.: Encoding context-sensitivity in Reo into non-context-sensitive semantic models. In: De Meuter, W., Roman, G.-C. (eds.) COORDINATION 2011. LNCS, vol. 6721, pp. 31–48. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21464-6_3
29. Kemper, S.: SAT-based verification for timed component connectors. Electron. Notes Theor. Comput. Sci. **255**, 103–118 (2009)
30. Kokash, N., Krause, C., de Vink, E.P.: Data-aware design and verification of service compositions with Reo and mCRL2. In: Shin, S.Y., Ossowski, S., Schumacher, M., Palakal, M.J., Hung, C. (eds.) Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, 22–26 March 2010, pp. 2406–2413. ACM (2010)
31. Kokash, N., Krause, C., de Vink, E.P.: Verification of context-dependent channel-based service models. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 21–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17071-3_2
32. Linden, I., Jacquet, J.-M.: On the expressiveness of absolute-time coordination languages. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 232–247. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24634-3_18
33. Linden, I., Jacquet, J.-M.: On the expressiveness of timed coordination via shared dataspaces. Electron. Notes Theor. Comput. Sci. **180**(2), 71–89 (2007)
34. Linden, I., Jacquet, J.-M., Bosschere, K.D., Brogi, A.: On the expressiveness of relative-timed coordination models. Electron. Notes Theor. Comput. Sci. **97**, 125–153 (2004)
35. Papadopoulos, G., Arbab, F.: Coordination models and languages. Technical report SEN-R9834. Centrum voor Wiskunde en Informatica (CWI) (1998). ISSN 1386–369X
36. Plotkin, G.: A structured approach to operational semantics. Computer Science Department, Aarhus University, DAIMI FN-19 (1981)
37. Proença, J.: Coordination of distributed components. Ph.D. thesis, Leiden University, Leiden, Netherland (2011)
38. Proença, J.: Synchronous coordination of distributed components. Ph.D. thesis, Leiden University, May 2011

39. Proenca, J., Clarke, D., de Vink, E., Arbab, F.: Decoupled execution of synchronous coordination models via behavioural automata. In: Mousavi, M., Ravara, A. (eds.) Proceedings 10th International Workshop on the Foundations of Coordination Languages and Software Architectures. EPTCS, vol. 58, pp. 65–79 (2011)
40. Proença, J., Clarke, D., de Vink, E., Arbab, F.: Dreams: a framework for distributed synchronous coordination. In: Ossowski, S., Lecca, P. (eds.) Proceedings of the ACM Symposium on Applied Computing, pp. 1510–1515. ACM (2012)
41. Soede, D., Arbab, F., Herman, I., ten Hagen, P.J.W.: The GKS input model in MANIFOLD. Comput. Graph. Forum **10**(3), 209–224 (1991)

Release the Beasts: When Formal Methods Meet Real World Data

Rudolf Schlatte, Einar Broch Johnsen(✉), Jacopo Mauro,
S. Lizeth Tapia Tarifa, and Ingrid Chieh Yu

Department of Informatics, University of Oslo, Oslo, Norway
{rudi,einarj,jacopom,sltarifa,ingridcy}@ifi.uio.no

Abstract. It is well-known that the difference between theory and practice seems smaller in theory than in practice. From the perspective of the coordinator, the coordinated components play the role of wild beasts, fortunately imprisoned in boxes. From the perspective of the care-free semanticist, the development of tools is merely a minor step away (possibly hidden in promises of future work). This paper draws parallels between beasts and tool building by describing challenges we have encountered and sharing experiences and lesson learned when going from a compositional semantics to a well-functioning tool interacting with industrial use cases. Concretely, we discuss the development of the simulation backend for Real-Time ABS.

In addition to his scientific contributions, Farhad Arbab has always been an outstanding speaker with a flair for inspiring talks and memorable punchlines. This paper is written for a highly appreciated colleague.

1 Introduction

Inside every box, there is a beast[1] just waiting to be discovered. Look closely and you will find it, lurking in the shadow of some interface. Even if you decide not to look, the beasts will still be there; their behavior an unsolvable mystery to the exogenous spectator. Each beast has its own particularities, its own irregularities, and its own side effects. Every beast is potentially a new friend, some of them can be worth knowing.

Many researchers rely on abstraction for their formalizations and reasoning systems. It is our secret weapon; apply it and a lot of problems simply vanish in a “puff” [1]. Programming languages are also getting increasingly more abstract, allowing us to express programs in more generic ways, relying on some low-level machinery to ensure that they are well-behaved. High-level languages should make it easier to express and prove the correctness of the behavior we want in

This research was supported by the SIRIUS Centre for Scalable Data Access (237898) and by the EU project HyVar: Scalable Hybrid Variability for Distributed Evolving Software Systems (H2020-644298).

[1] The metaphor of the ‘beast in the box’ was invented by Farhad Arbab around 2005.

F. de Boer et al. (Eds.): Arbab Festschrift, LNCS 10865, pp. 107–121, 2018.
https://doi.org/10.1007/978-3-319-90089-6_8

our systems. We continue to strive for more abstraction [2], for more semantics [3], for more compositionality [4]. As we climb the ladder of abstraction, we leave the operational behind in favor of the denotational, we ignore the "how" in favor of the "what". Let us consider Reo [5] as a case in point; happily unconcerned with the behavior it coordinates, it is pure compositionality with an endless flow of semantics [6]. Reo's different semantics describe the flow of data through connectors coinductively [6], using constraint automata [7] abstracting from the distinction between input and output, or as an artist's palette of colors [8]. Each semantics highlights a particular aspect of Reo's exogenously coordinated flow. The different semantics also enable the implementation of tools (e.g., [8–10]). The construction of tools reveals another kind of beasts: the implementation details and the interface to the real world. In fact, it is on the path from semantics to tools that we encounter the beasts that are the focus of this paper, where the ideal compositional world of semantics comes with many afterthoughts.

In the rest of this paper, we discuss these afterthoughts and open the boxes to get a closer look inside. Section 2 gives a brief overview of ABS and its semantics, the language that we are using to illustrate our findings when looking into the boxes. Section 3 proposes a starting point when venturing towards implementation and tools. Section 4 discusses implementation issues, Sect. 5 describes interfacing with models, and Sect. 6 describes issues of community and development. Sections 7 and 8 describe two case studies which illustrate the interest of venturing down this road.

2 A Short Overview of Real-Time ABS

Real-Time ABS is a formally defined, actor-based, object-oriented modeling language targeting distributed systems with early deployment decisions and timing requirements. Real-Time ABS extends ABS, a language with a formal syntax and semantics defined in operational semantics (SOS) [13] as well as trace semantics [12]. Compared to other actor or active object languages [26], two distinguishing features of Real-Time ABS are its support for *cooperative concurrency* and the explicit modeling of *deployment decisions* in a real-time setting.

The language is layered and combines a simple, functional language to express local computation; an object-oriented, imperative language for asynchronous communication and synchronization; and real-time and deployment layers which allows object requiring resources for their computations to be placed at locations with restricted resource capacity, and to model the time-sensitive behavior of these objects. The combination of functional and imperative layers makes it easy to model an object-oriented *design*, yet retain a high level of abstraction for internal computations and data modeling. The real-time and deployment layers make it possible to express timing properties and compare deployment decisions early in the software development process.

Real-Time ABS includes a Cloud API, used to model how software applications interact with a cloud provider [27]. The model offer services to client applications to dynamically acquire and release virtual machines on demand.

$$\begin{array}{c}\text{(S{\scriptsize USPEND})}\\ o(a, \{l \mid \textsf{\textbf{suspend}}; s\}, q)\\ \to o(a, \mathit{idle}, \{l \mid s\} \circ q)\end{array} \qquad \begin{array}{c}\text{(R{\scriptsize ELEASE}-C{\scriptsize OG})}\\ o(a, \mathit{idle}, q)\ c(o)\\ \to o(a, \mathit{idle}, q)\ c(\epsilon)\end{array} \qquad \begin{array}{c}\text{(A{\scriptsize CTIVATE})}\\ \dfrac{p = \mathit{select}(q, a, cn)}{\begin{array}{c}\{o(a, \mathit{idle}, q)\ c(\epsilon)\ cn\ cl(t)\}\\ \to \{o(a, p, (q \setminus p))\ c(o)\ cn\ cl(t)\}\end{array}}\end{array}$$

$$\begin{array}{c}\text{(R{\scriptsize UN}-I{\scriptsize NSIDE}-I{\scriptsize NTERVAL})}\\ \dfrac{\begin{array}{c}cn\ cl(t) \xrightarrow{!} cn'\ cl(t)\\ 0 < d \leq \mathit{mte}(cn', t) \quad \lfloor t \rfloor = \lfloor t + d \rfloor\end{array}}{\begin{array}{c}\{cn\ cl(t)\}\\ \to_t \{\mathit{timeAdv}(cn', d)\ cl(t + d)\}\end{array}}\end{array} \qquad \begin{array}{c}\text{(R{\scriptsize UN}-T{\scriptsize O}-N{\scriptsize EW}-I{\scriptsize NTERVAL})}\\ \dfrac{\begin{array}{c}cn\ cl(t) \xrightarrow{!} cn'\ cl(t)\\ 0 < d \leq \mathit{mte}(cn', t) \quad \lceil t \rceil = t + d\end{array}}{\begin{array}{c}\{cn\ cl(t)\}\\ \to_t \{\mathit{timeAdv}(\mathit{rscRefill}(cn'), d)\ cl(t + d)\}\end{array}}\end{array}$$

Fig. 1. Some rules from the operational semantics of Real-Time ABS.

The model of the cloud provider is based on deployment components, which are computation locations with limited resource capacities and which are used to represent created virtual machines of given processing capacities. The communication interface of the cloud provider allows a model of a client application to create machines with a desired execution capacity, acquire machines to start task executions, release machines, and finally get the accumulated usage cost. This API extension has been used in several case studies. In particular, Sect. 7 reports on experiences with Real-Time ABS using this Cloud API.

Figure 1 illustrates the SOS semantics of Real-Time ABS (for details of the full semantics, see [15,19]). In these rules, a *configuration* cn is a multiset of terms, including objects, concurrent object groups (cogs), which share a thread of execution, and execution locations with restricted amount of resources. The timed configuration includes a global clock $cl(t)$. The use of brackets encapsulating timed configurations allows the left hand sides of rules to match the *whole* configuration and not just some of its terms. An *object* is a term $o(\sigma, p, q)$ where o is the object's identifier, σ is a substitution representing the binding of the object's fields, p is an (active) process, and q a *pool of processes*. For the process pools q, concatenation is denoted by $q_1 \circ q_2$. A *process* $\{\sigma|s\}$ consists of a substitution σ of local variable bindings (including the variable *deadline* which denotes the remaining execution time of the process until a soft deadline is passed) and a list s of statements, or it is *idle*. A cog $c(\mathit{act})$ contains an identifier c and the currently active object act or ϵ if no object of the cog is currently active.

Rule Suspend enables cooperative scheduling and suspends the active process to the process pool, leaving the active process *idle*, and Release-Cog makes the cog idle if the object holding its lock is idle. Given an idle object with an idle cog, rule Activate schedules a process from the process queue and grabs the lock of the cog. Here, the function *select* chooses a process which is ready to execute from the process queue. If there is no such process, the premise is false.

Time advance in the semantics is specified by a transition relation $\to_t$ and the rules Run-Inside-Interval and Run-To-New-Interval. The model of time is based on maximal progress, so time will only advance when execution is otherwise blocked (i.e., $\xrightarrow{!}$ denotes the reduction to normal form in the premises of the rules). The rule Run-Inside-Interval captures time advance which does not influence the resource availability in the execution locations of the system, and the rule Run-To-New-Interval captures the case when the resources in the execution locations should be "refilled" for the next time interval. The function *mte* calculates the maximal time advance, which is the largest amount by which time can advance such that no "interesting" occurrence will be missed in any object or execution location. The function *timeAdv* updates the active and suspended processes of all objects, decrementing the values of all deadlines and duration statements. The function *rscRefill* captures the effect of time advance on the execution locations, causing the refilling of resources in each of them.

3 Leaving the World of Semantics and Compositionality

Compositionality is often regarded as the key to address real systems using formal methods. In semantics, compositionality gives us maintainability by minimizing the interference between different mathematical objects such that new objects will not violate existing semantic rules and such that different objects can be composed or coordinated via their interfaces. In reasoning, compositionality allows us to reason about each of these objects separately, and later put together the derived local behaviors by means of composition rules. In an ideal, mathematical setting, composition rules such as logical conjunction come naturally [11]. In practice, they are often complex and need to, e.g., resolve interference between processes [4] or match shared events in local traces [12]. Remark that compositionality also often leads to incompleteness in analysis by introducing *abstractions* in local reasoning in terms of interfaces, communication traces, scheduling traces, and other assumptions which generalize the environment.

A major challenge in formal methods is what we may call *"leaking abstractions"*. Leaking abstractions typically occur when our reasoning about a high-level model or a program requires more low-level information than we have available at the surface level: We have lost too much information in our abstractions. For example, the abstractions are leaking when knowledge about the runtime system's locks, its partitioning of data into blocks, or its (often unspecified or non-deterministic) scheduling decisions are required to reason about the behavior of programs which do not mention any locks, memory blocks, or schedulers in the surface language.

An interesting example of leaking abstractions is deployment. In high-level languages we want to abstract from knowledge of, e.g., memory layout, which processor gets to run a task, or how tasks distribute over nodes in a grid. With virtualization, hardware becomes data in our software programs. In our work on virtualized services for the cloud in the ABS modeling language, described in Sect. 2, we were confronted with how to give a high-level representation of low-level deployment details; we needed to explicitly represent time as well as dynamic deployment decisions allowing the program to change its own deployment. Our solutions were also confronted with real industrial case studies [14].

To capture uniform time advance and their effect on computing resources operationally, we suddenly needed global rules, as the one shown in Fig. 1. To apply the modeling language to industrial case studies, we needed efficient tools which integrated our models with real world operational data. To be useful to practitioners, these tools should not derive theorems from our models, but rather produce easily accessible information; exit Greek variables, enter the world of visual analytics. As all things flow [5], we have focussed on timed data streams depicting the runtime behavior of models.

4 From Operational Semantics to Simulation

If denotational semantics captures the "what" and operational semantics the "how", a simulator captures the "really how". This section discusses some details from the experiences gained in moving the perspective from the operational semantics of Real-Time ABS to the realm of execution (see Sect. 2 for a brief introduction to the language). The process of implementing a language's operational semantics into the tool domain includes many conventional software development tasks: fixing a concrete syntax that is expressible in ASCII, choosing an implementation platform, implementing a parser and type-checker, code generation, etc. In the case of Real-Time ABS, the tool chain runs on top of the Java Virtual Machine, translating ABS models into runnable code using one of several "backends". The first backend, initially developed for a precursor language called Creol [16], was implemented on top of the rewriting logic system Maude [17]. Later this backend was joined by a language implementation in Java and one in Erlang [18].

In addition to standard software engineering issues, translating a formal semantics such as Real-Time ABS into code presents some unique challenges. In this particular case even though the starting point of this translation was an operational semantics [15,19] detailing the "how", some of its rules have a denotational flavour hiding the "really how". Rules that are straightforward to understand in terms of "what" they are doing, devolve into convoluted code; e.g., the humble negation operator morphs into a global actor resulting in performance bottlenecks, etc. In the remainder of this section, we present a selection of interesting implementation challenges, encountered during the implementation of the Real-Time ABS simulator. The chosen challenges relate to the rules of the operational semantics shown in Fig. 1.

4.1 Clock Advance

A straightforward expression of a logical clock rule is: *If no process can execute, advance the clock by the maximum amount that makes no process miss a deadline.* This can be expressed in a rule such as Run-Inside-Interval of Sect. 2, Fig. 1, where the symbol $\xrightarrow{!}$ denotes the maximum application of other rules. In the more concrete world of Maude's rewriting logic, expressing that no process can execute entails checking the status of each process, slowing down simulation.

Lesson 1. *Semantic rules which contain global (whole-program) state are expensive and easily lead to problems of scaling during implementation.*

Rules with global state are not compositional by nature. This makes a direct implementation of semantic rules with global state badly suited for simulating systems with large state. This problem can be circumvented by introducing a centralized coordinator or a distributed protocol.

Note that the property "cannot execute" is non-monotonous since a process waiting for a computation result can become runnable again as a consequence of a process in another cog terminating. On the even less abstract distributed Erlang platform using explicit actors, this can easily lead to temporary "glitches" as a completion message travels from source cog to target cog. As a consequence, it was necessary to implement a dedicated singleton actor tracking the status of each cog. Entertaining months were spent chasing ever more improbable protocol errors that resulted in spurious clock advances.

Lesson 2. *Negations ("it is not the case that…") translate into universal quantifiers whose implementation requires knowledge of global state.*

4.2 Scheduling Processes

In contrast to the semantics of a logical clock representing dense time, which is specific to ABS, the semantics of scheduling of cooperative processes is well-understood and standard. In ABS, scheduling entails an idle cog picking an enabled process and executing it, thereby becoming busy. The semantics of scheduling is shown in Suspend, Release-Cog and Activate of Sect. 2, Fig. 1; the *select* function here returns a runnable process p from the process pool q.

This behavior was implemented in Erlang by choosing one element out of a set of ready process identifiers, sending it an activation signal and removing it from the ready process set. We were satisfied that this simple implementation was trivially correct and according to the desired semantics, until we received a bug report about a deadlock in the simulation engine.

The user, as it turns out, was running two processes communicating via a shared object field: process A was spinning on a field (`while(!field) suspend;`), while process B's task was to set the field (`field = True;`). Note that both of these processes are enabled and ready to run. Due to the implementation of Erlang's standard set datatype `gb_sets`, process A happened to be chosen every time, thereby starving process B that would, in turn, have enabled process A to make progress. The solution was to *(i)* gently mock the offending user's program, which should have used the ABS construct `await field;` instead of busy-looping, and *(ii)* implement a randomizing scheduler to cater for a potentially infinite sequence of naïve models in the future.

Lesson 3. *The simplest possible, obviously correct implementation of a semantic rule might not be suitable in practice.*

5 Getting the Real World into the Models

An implementation of a language semantics gives us a "compute kernel" of sorts that can be used to execute programs written in the language. However, pure computation, even when correctly implemented, is not always useful by itself. End users tend to expect facilities for input and output, which are often abstracted away in language semantics.

This section describes some useful patterns and extensions of Real-Time ABS in the areas of input, output, and visualization. Most of these are implemented in terms of a "Model API". The very first implementations of ABS (and Creol, its precursor) consisted of equations and rewrite rules over terms representing objects, processes and other runtime semantics entities that executed on the rewriting logic system Maude [17]. The result of a computation was represented as a dump[2] of the final state of the global configuration, rendered as ASCII. The results of simulation (both computation results and model state) were accessed by one-off scripts, often using regular expressions to extract relevant parts of this dumped output.

While working on various case studies with industrial partners, the need to both access model state and influence the model from the outside became apparent. The creation of other backends (and later versions of Maude) enabled the addition of printed character output to the language, but accessing richer, structured data remained elusive.

Lesson 4. *Simulation engines need visualization and text output as a minimum, ideally also a way to access structured state.*

In response to these end user needs, a *Model API* was established for the Erlang backend. The API is based on web technologies: communication via the HTTP protocol, with data returned in JSON format. This choice was made to maximize the ease of implementation of tools to interoperate with a running model; most programming languages come with standard libraries to emit HTTP requests and to parse JSON.

Figure 2 illustrates interaction with a running model from the command line: the user first obtains a list of entry points (entry points are added to the model by the modeler), then a list of callable methods and finally the result of a method call. A representation of an ABS object's internal state can be obtained in a similar way. This API was used in an industrial case study [20] to drive an ABS model according to traces obtained from the system logs of a real system.

Lesson 5. *Any aspect of a tool that is not core to its functionality (e.g., communication protocols, structured data storage) should be implemented using established industrial standards and existing libraries. This makes it easier for both implementor and end user.*

[2] The reviewers of the EU project Credo (https://projects.cwi.nl/credo/), tasked with implementing Creol [16], correctly pointed out that screenfuls of text (or, for larger model states, hundreds of kilobytes) were not an effective way of communicating and understanding model behavior.

```
~$ curl localhost:8080/call
["helloobj"]
~$ curl localhost:8080/call/helloobj
[{"name":"greeting",
  "parameters":[{"name":"name",
                 "type":"ABS.StdLib.String"}],
  "return":"ABS.StdLib.String"}]
~$ curl localhost:8080/call/helloobj/greeting?name=Joe
{"result":"Hello␣Joe!"}
```

Fig. 2. Interacting with the Model API.

One consequence of adding a Model API, i.e., a way of communicating with a model from outside, is that we move from a closed to an open world where the full behavior of the model can no longer be analyzed statically. This can impact proof theories and other analysis approaches, especially when relying on the whole-program analysis. Modular, compositional analysis methods are less affected as only a few selected modules are opened to the outside world.

Lesson 6. *Tools have different, sometimes conflicting requirements. Making a language implementation more useful for simulation ("programming") can result in proofs of correctness becoming more difficult, and vice versa.*

6 Getting the Models into the Real World

In the early days of ABS and its extensions, knowledge about the language was transmitted orally. All users were part of the same institution, or at least of the same project, so education and discovery of best practices happened face-to-face. Similarly, bugs and problems were discovered, reported, discussed, and fixed via personal interaction. However, this does not scale for a language with users that are not personally known to the language implementors and designers.

The aim for a widely-used tool must be to make it "self-supporting"; i.e., the users should be able to find the answers to common problems by themselves. Updating the documentation in response to user questions must be an ongoing process.

Lesson 7. *When a user asks a question that is covered by the documentation, ask where they looked for the answer, then update the documentation to put it there.*

Additionally, a lot of programs are developed in a process of "coding-by-imitation". Good examples and tutorials help in the process of picking up an unfamiliar language.

Lesson 8. *Provide both small and large examples that show best practices and "proper" ways to use a language to its fullest potential.*

Another aspect of language uptake is visibility of ongoing development. For users, access to the source code provides a measure of safety—but maybe more important is a visible and accessible development process. Multi-year commit activity and prompt responses to bug reports assure prospective customers that any problems they might uncover will likely be solved as well.

Lesson 9. *Make development activity visible to interested end users.*

On the other hand, development can lead to "churn" in that introducing and adapting features can break old code. Once a language is used more broadly, care must be taken not to invalidate the users' work. This can be done in multiple ways: by keeping deprecated features around if they do not conflict with new features; by documenting changes and update paths for outdated code; and by providing means of identifying the tool version used for a specific model and obtaining that version later, should the need arise.

Lesson 10. *Do not break user code unnecessarily, and provide ways forward (adapting code for new tool versions) and backward (obtaining previous tool versions) in case of necessary changes.*

7 Use Case: Scaling with Traffic Data

This section describes a use case modeling a microservice architecture for dispatching car software updates [21]. The use case describes an innovative business model which combines cloud computing and microservices to allow on-demand delivery of scalable and modular applications with pay-as-you-go pricing.

The starting point for the model creation was the existing microservice architecture. Part of the challenge was to create an appropriate abstraction, i.e., a simple executable model which exposes scaling decisions as configurable parameters. ABS helped to cope with this challenge because (i) it natively supports CPU, memory resources and the notion of deployment components [15], (ii) being a full-fledged language it is more flexible than ad-hoc cloud simulators, (iii) it has parallel run-time support in Erlang, (iv) tools for worst-case performance analysis [22] and visualization are available.

Figure 3 shows the chosen methodology. First, we used worst case analysis (e.g., queuing theory) and profiling techniques to understand which parts of the system could be simplified and abstracted. This allowed in a second step to create a simple model with fewer parameters to tune. Finally, to reduce the cost of the cloud resources used by the microservice system and find good scaling parameters, we used automatic parameter configurators [23,24], i.e., tools that rely on machine learning techniques to explore the possible configurations in a smarter and more systematic way and come up with good parameter settings.

The creation of the model[3] was quite straightforward. A microservice instance and the load balancer for the redirection of requests was represented with objects.

[3] https://github.com/HyVar/abs_optimizer.

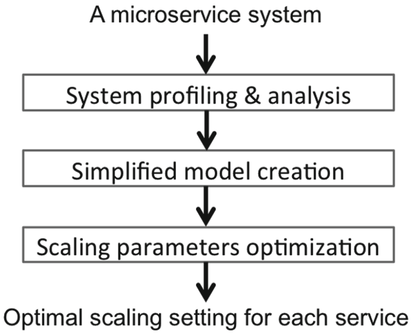

A microservice system
System profiling & analysis
Simplified model creation
Scaling parameters optimization
Optimal scaling setting for each service

Fig. 3. The scaling optimization methodology.

The internal computation performed by a microservice was abstracted to a skip statement taking a given computation cost `c` as follows.

```
[Cost: c] skip;
```

The objects were instantiated on deployment components, a native construct of ABS used to represent the virtual machines on which the real microservice instances are deployed. In this way, the acquisition/dismissal of a virtual machine for scaling up/down was modeled by the creation/removal of a deployment component exploiting the ABS native Cloud API (for details, see Sect. 2).

Based on this model, we can now search for good scaling settings using the Sequential Model-Based Optimization for General Algorithm Configuration (SMAC) tool [24], an automatic parameter configurator, to explore possible configurations. This computationally heavy task was done using 64 nodes in a Numascale cluster, a scalable cluster with shared memory[4]. We run 64 instances of SMAC in parallel for 12 h. Every execution of SMAC was performing in sequence the simulations running the generated Erlang processes on 6 dedicated cores. The input request pattern used 24-h of car traffic based on the number of cars registered on the A414 highway, UK, on Monday, March 2, 2015.

Calibrating the microservice system with good scaling settings was in this case vital. In theory a microservice system is a simple thing, in practice it was not. In the beginning, our abstraction was completely disconnected from the actual performances of the system. For instance, instead of having a uniform latency distribution (predicted by our model), we obtained a distribution of latencies like the one presented in Fig. 4. We could not understand why this was happening since, based on our simulations, this was not explainable by considering network problems or the variability of the performance of the cloud. At the end, we discovered that the Amazon default "round-robin" load balancers used in the real system implementation were not adopting a strictly round robin policy. This official response on the AWS blog[5] highlights the issue:

[4] https://www.numascale.com/.

[5] https://forums.aws.amazon.com/message.jspa?messageID=316829.

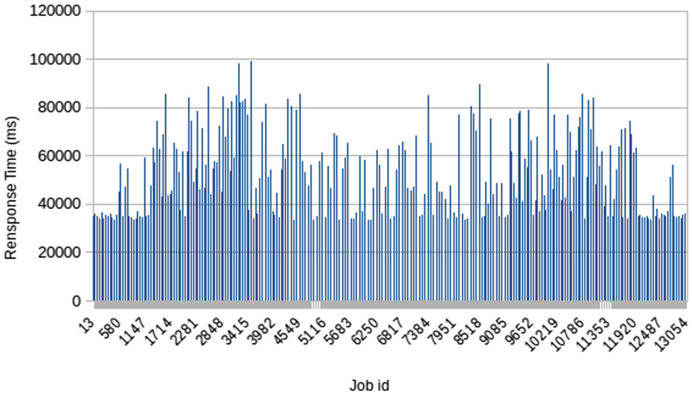


Fig. 4. Uneven request processing times.

> "Round-robin does come into play but the client sessions do not always honour TTL's or DNS caches so you can get skewed results and uneven distributions of requests. The ELB does not take into effect what traffic/requests instances have received to-date in there traffic routing decisions."

When using Amazon's load balancers, this problem rendered our abstraction useless. We tried multiple approaches to mitigate this while still running the default Amazon load balancers without achieving a satisfactory level of precision, due to the unknown real policy of Amazon load balancers.

To improve predictability, the original microservice system was changed by replacing the Amazon load balancers with HAProxy[6], an open source and more controllable solution. This way, the original system was improved and the abstraction was able to predict its behavior, thus allowing good scaling strategies to be found [21]. Figure 5 shows that the simulation was robust enough to mimic the real system and offer a performance estimation usable to set good scaling parameters, even considering the random performance fluctuation of the cloud instances. We used the Model API described in Sect. 5 to visualize the simulations. This visualization greatly simplified the discovery of discrepancies and, later, the gain of confidence in the robustness of the model. In this particular case, it was possible to change the original application to make the model accurate. However, this is not always the case. Developing accurate models which faithfully represent commercial black box components still remains a challenge.

Lesson 11. *Without having a faithful representation of the behavior of the system, an optimization step in the model is useless.*

[6] https://www.haproxy.org.

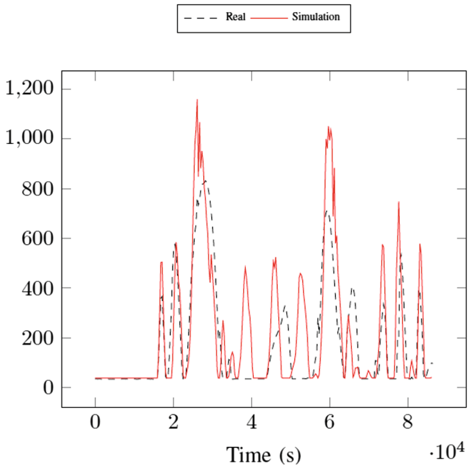


Fig. 5. Comparison latency as predicted by ABS to the latency of the real system.

8 Use Case: Vessel Planning

Whereas Sect. 7 showcased a case study of a distributed software system with virtualized deployment, ABS has been increasingly used to model other kinds of systems (e.g., railways [25]). In this section, we consider an industrial case study from the domain of operational planning. This case study addresses vessel movements and cargo transport in the North Sea. The stakeholders want to improve their workflow to have a better overview of the (potential) bottlenecks delaying overall progress, the general load on different vessels, and the quality of their logistics operation both in terms of the exploitation of vessel capacity and on the timely delivery of material. We use Real-Time ABS as a modeling language to simulate and visualize the actual logistics operations. Compared to the tools currently used, Real-Time ABS simulations provide a different level of overview which helps to gain precision in the decision making phase.

The case study illustrates the usefulness of ABS modeling beyond the realm of computing systems, and makes use of both the input and output-facilities of the Model API to drive the simulation of the model and for visualization of output. Currently Real-Time ABS is here used for simulations. In a longer term perspective, we intend to combine these simulations with stronger analyses to generate solutions and verify their correctness with respect to requirements such as resource restrictions, safety regulations, and space limitations.

We are working with industrial data from different parts of a complex supply chain, and integrate these into a uniform ABS model. The data covers transport plans for a large number of vessels moving between processing plants, with logs for bulk and cargo delivery covering a twelve month period. In this use case, ABS is used to define a general framework for modeling transport plans by means of abstractions for, e.g., vessels, containers, bulk cargo, route segments, and delivery deadlines in a generic way.

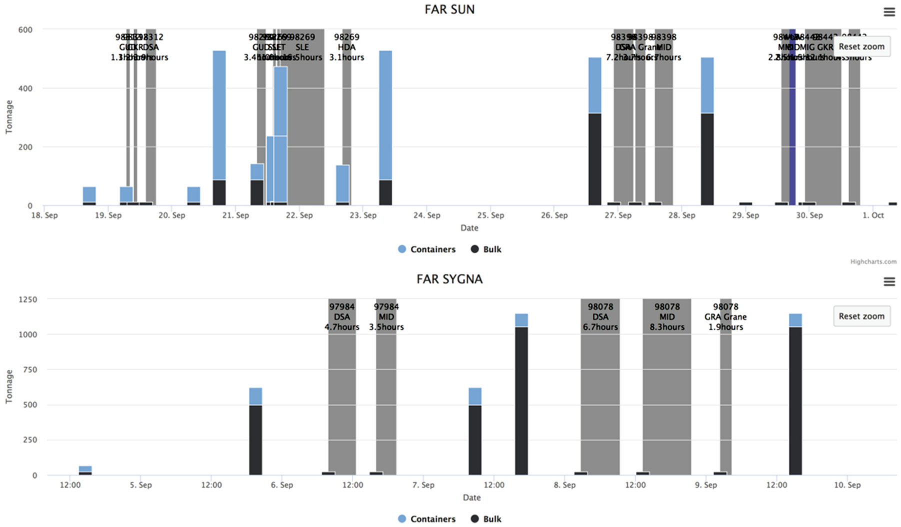


Fig. 6. Visualization of time series data depicting vessel movements.

The model is populated by specific data representing a concrete plan. This is currently done by moving the data from Excel into a SQL database, then generating ABS data structures corresponding to the industrial data set. Thus, the industrial data set acts as the driver for the ABS model. The modeler specifies a time window, and data for this time interval is converted from the SQL database and turned into the model of a concrete plan. This allows the ABS model of the concrete plan for the given time window to be simulated. The planner is presented with a graphical view of the simulated plan, see Fig. 6. This graphical view is dynamically generated in-browser from JSON data fetched via the Model API (described in Sect. 5) and can be easily adapted by a frontend developer; no knowledge of ABS is needed to create different views over the simulation data.

A practical challenge with this case study, in addition to the data cleaning required to convert operational data to fit with the modeling framework in Real-Time ABS and the interaction of the simulation backend and the SQL database, was the conversion of calendar data to model time. Real-Time ABS represents time using rational numbers. We calibrated the model with time `0` representing midnight on the first day simulated. Subsequent dates were numbered `1`, `2` ..., with the fractional part representing time of day. This approach gave us sufficient resolution to model real time using abstract time units.

9 Conclusion

This paper has discussed challenges in moving from formal, compositional language semantics to industrially applicable tools. These challenges span from pattern matching in reduction rules necessitating protocols in a distributed implementation to documentation and input/output interfaces for real world data. We have compared these challenges to the beasts hidden in the boxes of the exogenous coordinator.

References

1. Arbab, F.: Puff, the magic protocol. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 169–206. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24933-4_9
2. Kramer, J.: Is abstraction the key to computing? Commun. ACM **50**(4), 36–42 (2007)
3. Jongmans, S.T.Q., Arbab, F.: Overview of thirty semantic formalisms for Reo. Sci. Ann. Comput. Sci. **22**(1), 201–251 (2012)
4. de Roever, W.P., de Boer, F.S., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press, Cambridge (2001)
5. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
6. Arbab, F., Rutten, J.J.M.M.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2002. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40020-2_2
7. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. **61**(2), 75–113 (2006)
8. Clarke, D., Costa, D., Arbab, F.: Connector colouring I: synchronisation and context dependency. Sci. Comput. Program. **66**(3), 205–225 (2007)
9. Arbab, F., Meng, S., Moon, Y., Kwiatkowska, M.Z., Qu, H.: Reo2MC: a tool chain for performance analysis of coordination models. In: van Vliet, H., Issarny, V. (eds.) Proceedings of 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 287–288. ACM (2009)
10. Clarke, D., Proença, J., Lazovik, A., Arbab, F.: Channel-based coordination via constraint satisfaction. Sci. Comput. Program. **76**(8), 681–710 (2011)
11. Abadi, M., Lamport, L.: Composing specifications. ACM Trans. Program. Lang. Syst. **15**(1), 73–132 (1993)
12. Din, C.C., Hähnle, R., Johnsen, E.B., Pun, K.I., Tapia Tarifa, S.L.: Locally abstract, globally concrete semantics of concurrent programming languages. In: Schmidt, R.A., Nalon, C. (eds.) TABLEAUX 2017. LNCS (LNAI), vol. 10501, pp. 22–43. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66902-1_2
13. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
14. Albert, E., de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L., Wong, P.Y.H.: Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time ABS. J. Serv.-Oriented Comput. Appl. **8**(4), 323–339 (2014)

15. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. J. Logical Algebraic Methods Program. **84**(1), 67–91 (2015)
16. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: a type-safe object-oriented model for distributed concurrent systems. Theor. Comput. Sci. **365**(1–2), 23–66 (2006)
17. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
18. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf, Dallas (2007)
19. Bjørk, J., de Boer, F.S., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: User-defined schedulers for real-time concurrent objects. ISSE **9**(1), 29–43 (2013)
20. Bezirgiannis, N., de Boer, F., de Gouw, S.: Human-in-the-loop simulation of cloud services. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) ESOCC 2017. LNCS, vol. 10465, pp. 143–158. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67262-5_11
21. Lin, J.C., Mauro, J., Røst, T.B., Yu, I.C.: A model-based scalability optimization methodology for cloud applications. In: Proceedings of 7th IEEE International Symposium on Cloud and Service Computing (IEEE SC2). IEEE CS Press (2017)
22. Albert, E., Flores-Montoya, A., Genaim, S., Martin-Martin, E.: May-happen-in-parallel analysis for actor-based concurrency. ACM Trans. Comput. Log. **17**(2), 11:1–11:39 (2016)
23. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: ISAC - instance-specific algorithm configuration. In: Proceedings of 19th European Conference on Artificial Intelligence (ECAI 2010). Frontiers in Artificial Intelligence and Applications, vol. 215, pp. 751–756. IOS Press (2010)
24. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Coello, C.A.C. (ed.) LION 2011. LNCS, vol. 6683, pp. 507–523. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25566-3_40
25. Kamburjan, E., Hähnle, R.: Uniform modeling of railway operations. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2016. CCIS, vol. 694, pp. 55–71. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-53946-1_4
26. Boer, F.D., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., Yang, A.M.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (2017)
27. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Modeling resource-aware virtualized applications for the cloud in real-time ABS. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 71–86. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34281-3_8

Formalizing Propagation of Priorities in Reo, Using Eight Colors

Sung-Shik Jongmans[1,2(✉)]

[1] Department of Computer Science, Open University, Heerlen, The Netherlands
ssj@ou.nl

[2] Department of Computing, Imperial College London, London, UK

Abstract. Reo is a language for programming of coordination protocols among concurrent processes. Central to Reo are connectors: programmable synchronization/communication mediums used by processes to exchange data. Every connector runs at a clock; at every tick, it enacts an enabled synchronization/communication among processes.

Connectors may prioritize certain synchronizations/communications over others. "Passive" connectors use their priorities only at clock ticks, to decide which enabled synchronization/communication to enact. "Active" connectors, in contrast, use their priorities also between clock ticks, to influence which synchronizations/communications become enabled; they are said to "propagate their priorities".

This paper addresses the problem of formalizing propagation of priorities in Reo. Specifically, this paper presents a new instantiation of the connector coloring framework, using eight colors. The resulting formalization of propagation of priorities is evaluated by proving several desirable behavioral equalities.

Foreword

This paper addresses, perhaps, the oldest open problem in the Reo community.

The problem came to my attention for the first time in May 2011, six months into my PhD project. Perhaps—nay, surely!—I should have walked away from it; oh, the time *that* would have saved me... But, the problem was too tempting to resist. Farhad, Kasper, and I worked on solutions intermittently over the past years. Many times, I thought we had solved it; equally many times, we had not.

I promised Farhad more than once to end our suffering (my choice of words), by formalizing propagation of priorities in the connector coloring framework, using $k > 3$ colors. I never quite succeeded. This seems the perfect occasion to finally, half a decade down the road, fulfill that promise. Well, to some extent.

1 Introduction

Context. Reo is a language for programming of coordination protocols among concurrent processes. Central to Reo are *connectors*: programmable synchronization/communication mediums used by processes to exchange data, by invoking

F. de Boer et al. (Eds.): Arbab Festschrift, LNCS 10865, pp. 122–138, 2018.
https://doi.org/10.1007/978-3-319-90089-6_9

write and take operations. Every connector runs at a clock; at every tick, it enacts an enabled synchronization/communication among processes, based on the operations those processes have performed.

To send data, a process can invoke a write operation on the interface of a connector; to receive, it can invoke a take operation. Both writes and takes are *blocking*: after a process has invoked write or take, it immediately *suspends*, its operation becomes *pending*, and it *resumes* only after its operation has been *resolved* by the connector. To resolve a pending write, a connector performs a reciprocal take; to resolve a pending take, it performs a reciprocal write.

As connectors fully control resolution of pending operations, only connectors decide *when* (synchronization) and *whereto/wherefrom* (communication) data *flow*. In this way, connectors coordinate the synchronization/communication among processes.

Problem. Connectors may *prioritize* certain synchronizations/communications over others. "Passive" connectors use their priorities only *at* clock ticks, to decide which enabled synchronization/communication to enact. "Active" connectors, in contrast, use their priorities also *between* clock ticks, to influence which synchronizations/communications become enabled; they are said to "propagate their priorities".

Imagine, for instance, a connector C among processes P_1, P_2, and P_3. Imagine, moreover, that at every clock tick, C can enact either a data-flow from P_1 to P_3 with high priority (enabled only if P_1 and P_3 invoked write and take), or a data-flow from P_2 to P_3 with low priority (enabled only if P_2 and P_3 invoked write and take). If C is passive, it quietly awaits the next clock tick, checks which operations are pending to determine which data-flows are enabled (if any), chooses and enacts the one with the highest priority, and quietly awaits the next clock tick. If C is active, in contrast, it requests P_1 to invoke write (and P_3 to invoke take) *before* the next clock tick, thereby enabling C to choose and enact the high priority data-flow from P_1 to P_3 *at* the next clock tick.

Contribution. Existing formalizations of Reo do not support modeling of connectors that propagate priorities. This paper presents such a formalization.

Section 2 establishes terminology and definitions. The section is terse; more gentle introductions to Reo [Arb04, Arb11] and the connector coloring framework [CCA07, Cos10] appear elsewhere. Section 3 details the problem of formalizing propagation of priorities. Section 4 presents a solution in the connector coloring framework, using eight colors. Section 5 contains an evaluation of this solution, in terms of behavioral equalities. Section 6 concludes this paper with a discussion. Appendix A contains definitions. Proofs appear in a technical report [Jon18].

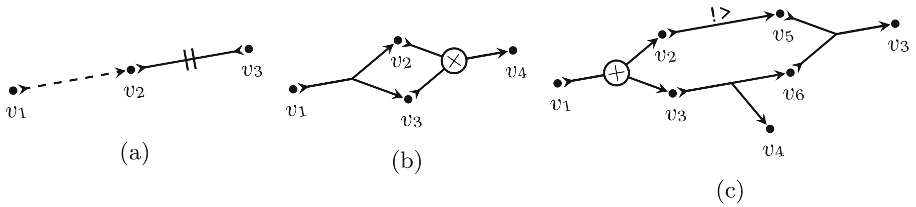


Fig. 1. Examples of connector syntax

2 Preliminaries

Syntax. Structurally, a connector in Reo is a (directed hyper)*graph* of *vertices* and (nonempty, directed hyper)*edges*.[1] Every edge is labeled with a *type*, shortly used to define the semantics of a connector. Figure 1 shows examples.

A vertex of a connector is *external* if it is the source of exactly one edge, or the target of exactly one edge; otherwise, it is *internal*. Processes perform `write` and `take` operations on external vertices, which thus consistute the interface.

A connector is *primitive*, if it has exactly one edge; otherwise, it is *compound*. Figure 2, first column, shows the name and syntax of common primitives.

A connector is *well-formed*, if (i) it has at least one edge, and (ii) if each of its vertices is the *source* of at most one edge, the *target* of at most one edge, and the source or target of at least one edge.

The structural composition of two connectors, denoted by operator $\bowtie$, is the graph consisting of the union of the sets of vertices, and the union of the sets of edges; it is a partial operation, to preserve well-formedness. Moreover, structural composition is associative and commutative.

A vertex is *shared* between two connectors, if it is an external vertex of both.

Informal Semantics. Behaviorally (informal), a connector in Reo is a set of data-flows between vertices, along edges, endowed with a partial order of priorities.[2]

A vertex is *active* in a data-flow, if data passes through it; otherwise, it is *passive*. Every vertex participates either actively or passively in each of its connector's data-flows. *Idling* is the degenerate data-flow in which every vertex participates passively. A data-flow of a connector is *enabled*, if every external vertex that actively participates in the data-flow has a pending `write` or `take`; idling is always enabled, vacuously.

A connector runs on a clock; at every tick, it enacts one of its enabled data-flows. If multiple data-flows are enabled, it nondeterministically selects an order-theoretically maximal one among them. Figure 2, second column, shows the informal semantics of common primitives; "prioritizes (n) over (m)" means "(n) is greater than (m)".

[1] Binary edges are usually called *channels*; maximal sets of adjacent ternary edges are usually called *nodes* [Arb04, Arb11].

[2] For simplicity, and because it is a concern orthogonal to formalizing priorities, I consider only stateless connectors in this paper.

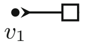
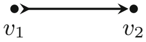
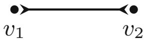
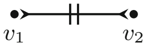
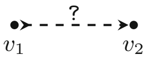
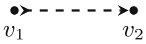
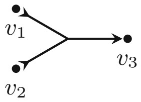
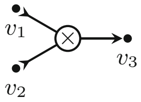
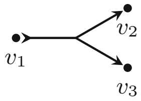
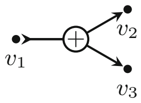
Name & Syntax	*Informal semantics*
Drain 	1. It takes data through v_1, and loses it. 2. Or, it idles. It prioritizes (1) over (2).
Sync 	1. It takes data through v_1, and writes it through v_2. 2. Or, it idles. It prioritizes (1) over (2).
SyncDrain 	1. It takes data through v_1 and v_2, and loses them. 2. Or, it idles. It prioritizes (1) over (2).
ExclDrain 	1. It takes data through v_1, and loses it. 2. Or, it takes data through v_2, and loses it. 3. Or, it idles. It prioritizes (1) over (3), and (2) over (3).
LossySync$_?$ 	1. It takes data through v_1, and writes it through v_2. 2. Or, it takes data through v_1, and loses it. 3. Or, it idles. It prioritizes (1) over (3), and (2) over (3).
LossySync 	1. It takes data through v_1, and writes it through v_2. 2. Or, it takes data through v_1, and loses it. 3. Or, it idles. It prioritizes (1) over (3), and (2) over (3), and (1) over (2).
Merger 	1. It takes data through v_1, and writes it through v_3. 2. Or, it takes data through v_2, and writes it through v_3. 3. Or, it idles. It prioritizes (1) over (3), and (2) over (3).
Join 	1. It takes data through v_1 and v_2, and writes the set containing them through v_3. 2. Or, it idles. It prioritizes (1) over (2).
Replicator 	1. It takes data through v_1, and writes it through v_2 and v_3. 2. Or, it idles. It prioritizes (1) over (2).
ExclRouter 	1. It takes data through v_1, and writes it through v_2. 2. Or, it takes data through v_1, and writes it through v_3. 3. Or, it idles. It prioritizes (1) over (3), and (2) over (3).

Fig. 2. Name, syntax, and informal semantics of common primitives

(1, 1) It takes data through v_1, writes/takes it through v_2, and loses it.
(2, 2) Or, it takes data through v_1 and v_3, and loses it.
(2, 3) Or, it takes data through v_1 and loses it.
(3, 2) Or, it takes data through v_3 and loses it.
(3, 3) Or, it idles.
It prioritizes (1, 1) over (3, 3), and (1, 1) over (2, 3), and (2, 2) over (3, 3), and (2, 3) over (3, 3), and (3, 2) over (3, 3).

Fig. 3. Informal semantics of Fig. 1a.

A data-flow through one connector is *consistent* with a data-flow through another connector, if each of their shared vertices is either active or passive in *both* data-flows. This ensures data can flow between connectors, through their shared vertices. The behavioral composition of two connectors is the set consisting of the pairs of consistent data-flows, endowed with their product order. Every global data-flow through a compound connector, thus, is the concatenation of local data-flows.

For instance, the connector in Fig. 1a is composed of LossySync and ExclDrain in Fig. 2. As these connectors both have three local data-flows, the compound has at most nine global data-flows. Figure 3 shows which of those data-flows are consistent; (n, m) means "the pair consisting of (n) of LossySync and (m) of ExclDrain". As the compound prioritizes (1, 1) over (2, 3), and because (1, 1) and (2, 3) are *always* enabled together, it *never* enacts (2, 3).

Formal Semantics. Behaviorally (formal), in the connector coloring framework, a connector is a set of total functions, called *colorings*, from vertices to natural numbers, called *colors* [CCA07,Cos10,JKA11,CP12]. Every coloring models a data-flow; every color models the activeness/passiveness of a vertex in a data-flow. Depending on the number of colors the framework is instantiated with, different levels of activeness/passiveness can be distinguished, to lesser or greater expressiveness. In particular, colors can be used to model priorities, as an alternative to endowing sets of colorings with partial orders (exemplified shortly).

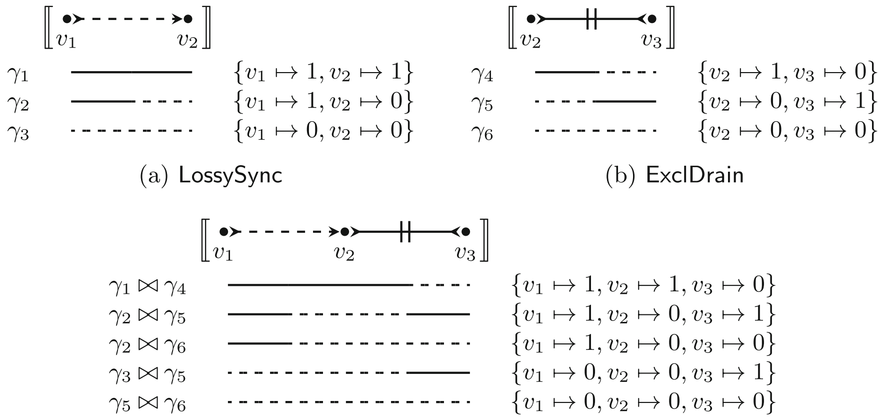
Two colorings are consistent if they map the vertices in the intersection of their domains to the same colors. The behavioral composition of two connectors, denoted by operator $\bowtie$, is the set consisting of the unions of their consistent colorings. As such, behavioral composition in the connector coloring framework straightforwardly models concatenation of consistent data-flows.[3] Behavioral composition is associative and commutative.

The structure and behavior of a connector are related through a *denotation function* $[\![\cdot]\!]$: it consumes as input a connector structure (graph) and produces as output a connector behavior (set of colorings), by decomposing the connector into primitives, looking up the local behavior of every primitive in a predefined type-indexed table, and composing the local behaviors into a global one.

[3] The composition operator can be extended with the *flip-rule* [CCA07,Cos10], to reduce sets of colorings. I do not pursue this in this paper.

#	•≻—— ——➤•	*Meaning*
0	- - - - -	Passive
1	———	Active
2	- -▷- -	Passive, for no `write`
3	- -◁- -	Passive, for no `take`
4	━━➡━━	Active; metadata-flow downstream (to propagate priorities)
5	━━⬅━━	Active; metadata-flow upstream (to propagate priorities)
6	━⬅➡━	Active; metadata-flows downstream + upstream (to propagate priorities)
7	- -▷▷- -	Passive, for no `write`, for conflicting propagated priorities upstream
8	- -◁◁- -	Passive, for no `take`, for conflicting propagated priorities downstream

Fig. 4. Colors


v_1 v_2
γ_1 $\{v_1 \mapsto 1, v_2 \mapsto 1\}$
γ_2 $\{v_1 \mapsto 1, v_2 \mapsto 0\}$
γ_3 $\{v_1 \mapsto 0, v_2 \mapsto 0\}$
v_2 v_3
γ_4 $\{v_2 \mapsto 1, v_3 \mapsto 0\}$
γ_5 $\{v_2 \mapsto 0, v_3 \mapsto 1\}$
γ_6 $\{v_2 \mapsto 0, v_3 \mapsto 0\}$
v_1 v_2 v_3
$\gamma_1 \bowtie \gamma_4$ $\{v_1 \mapsto 1, v_2 \mapsto 1, v_3 \mapsto 0\}$
$\gamma_2 \bowtie \gamma_5$ $\{v_1 \mapsto 1, v_2 \mapsto 0, v_3 \mapsto 1\}$
$\gamma_2 \bowtie \gamma_6$ $\{v_1 \mapsto 1, v_2 \mapsto 0, v_3 \mapsto 0\}$
$\gamma_3 \bowtie \gamma_5$ $\{v_1 \mapsto 0, v_2 \mapsto 0, v_3 \mapsto 1\}$
$\gamma_5 \bowtie \gamma_6$ $\{v_1 \mapsto 0, v_2 \mapsto 0, v_3 \mapsto 0\}$

(a) LossySync

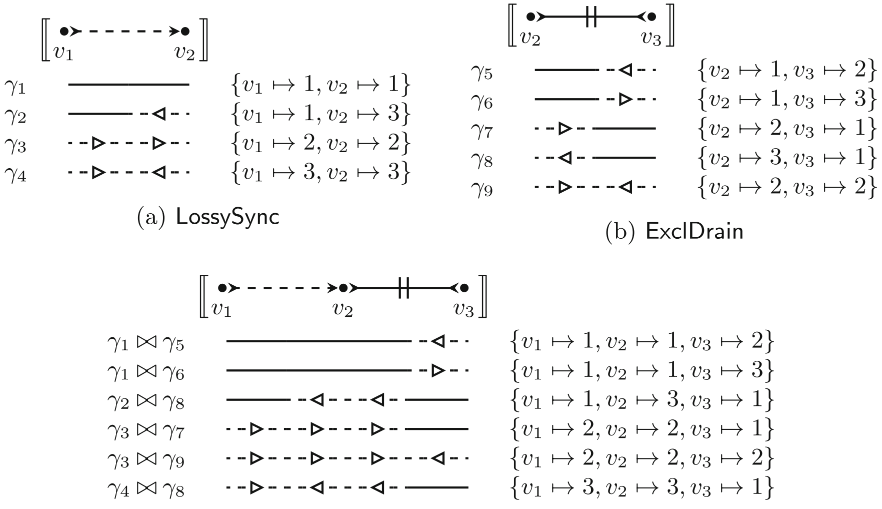
(b) ExclDrain

(c) Figure 1a

Fig. 5. Examples of two-color semantics

To exemplify the connector coloring framework, Fig. 4 shows nine colors. Colors $0, 1, 2, 3$ already exist in the literature; colors $4, 5, 6, 7, 8$ are new. The following lists summarizes three existing instantiations of the framework:

- With two colors [CCA07, Cos10], $\{0, 1\}$, one can model data-flows, but not priorities. Figure 5 shows examples. As the figure shows, colorings can be represented both textually and graphically (using the notation in Fig. 4). Figure 5a shows the behavior of LossySync. Coloring γ_1 models a data-flow from v_1 to v_2 (both vertices are active); coloring γ_2 models the loss of data taken through v_1 (only v_1 is active); coloring γ_3 models idling. Figure 5b and c can be explained similarly. The colorings in Fig. 5 model exactly, one-to-one, the data-flows in Figs. 2 and 3. However, priorities are not modeled.
- With three colors [CCA07, Cos10], $\{1, 2, 3\}$, one can model both data-flows and priorities. Specifically, color 0 is refined into colors 2, 3, to model not only *that* a vertex is passive, but also *why*. Figure 6 shows examples. I write "the


v_1
v_2
γ_1 $\{v_1 \mapsto 1, v_2 \mapsto 1\}$
γ_2 $\{v_1 \mapsto 1, v_2 \mapsto 3\}$
γ_3 $\{v_1 \mapsto 2, v_2 \mapsto 2\}$
γ_4 $\{v_1 \mapsto 3, v_2 \mapsto 3\}$
v_2
v_3
γ_5 $\{v_2 \mapsto 1, v_3 \mapsto 2\}$
γ_6 $\{v_2 \mapsto 1, v_3 \mapsto 3\}$
γ_7 $\{v_2 \mapsto 2, v_3 \mapsto 1\}$
γ_8 $\{v_2 \mapsto 3, v_3 \mapsto 1\}$
γ_9 $\{v_2 \mapsto 2, v_3 \mapsto 2\}$
v_1
v_2
v_3
$\gamma_1 \bowtie \gamma_5$ $\{v_1 \mapsto 1, v_2 \mapsto 1, v_3 \mapsto 2\}$
$\gamma_1 \bowtie \gamma_6$ $\{v_1 \mapsto 1, v_2 \mapsto 1, v_3 \mapsto 3\}$
$\gamma_2 \bowtie \gamma_8$ $\{v_1 \mapsto 1, v_2 \mapsto 3, v_3 \mapsto 1\}$
$\gamma_3 \bowtie \gamma_7$ $\{v_1 \mapsto 2, v_2 \mapsto 2, v_3 \mapsto 1\}$
$\gamma_3 \bowtie \gamma_9$ $\{v_1 \mapsto 2, v_2 \mapsto 2, v_3 \mapsto 2\}$
$\gamma_4 \bowtie \gamma_8$ $\{v_1 \mapsto 3, v_2 \mapsto 3, v_3 \mapsto 1\}$

(a) LossySync

(b) ExclDrain

(c) Figure 1a

Fig. 6. Examples of three-color semantics

environment can `write`/`take` through v" to mean that either a `write`/`take` is pending on v (if the environment at v is a process) or a data-flow can be concatenated at v (if the environment at v is another connector).

The expressive power of the three-color semantics is best exemplified with LossySync, as follows. Coloring γ_2 in Fig. 5a and coloring γ_2 in Fig. 6a both model the loss of data taken through v_1. However, γ_2 in Fig. 6a additionally models that this data-flow can be chosen/enacted only if no `take` can be resolved at v_2. As v_2 is a target vertex of LossySync (i.e., LossySync can only `write` through v_2), this happens only if the environment cannot `take` through v_2. Thus, if the environment can `write` through v_1, but not `take` through v_2, LossySync can lose (γ_2). But, if the environment can both `write` and `take`, LossySync must choose to not-lose (γ_1) instead of to lose (γ_2), just as its informal semantics demands (Fig. 2).

The three-color semantics of LossySync$_?$ is the same as the three-color semantics of LossySync, plus coloring $\gamma_2' = \{v_1 \mapsto 1, v_2 \mapsto 2\}$. This extra coloring models the loss of data taken through v_1, just as γ_2 in Fig. 6a. However, γ_2' additionally models that this data-flow can be chosen/enacted only if no `write` can be resolved at v_2. As v_2 is a target vertex of LossySync$_?$ (i.e., the environment can only `take` through v_2), this happens only if LossySync$_?$ cannot `write` through v_2. This is a condition that LossySync$_?$ *always* can satisfy (independent of the environment). Thus, if the environment can both `write` and `take`, LossySync$_?$ nondeterministically chooses between not-losing (γ_1) and losing (γ_2'); in the former case, it `writes` through v_2, while in the latter case, it does not. Thus, γ_2' is the three-color equivalent of γ_2 in Fig. 5a.

LossySync and LossySync$_?$ illustrate that by carefully modeling *why* vertices are passive, using colors $2, 3$, priorities may emerge. Graphically, the triangle markings always point away from the *root cause* for passiveness. For instance, in coloring $\gamma_3 \bowtie \gamma_7$, vertex v_2 is passive, because there is no `write` on v_2 (cause), because the environment cannot `write` through v_1 (root cause).

- With four colors [CP12], $\{0, 1, 2, 3\}$, one can model data-flows, priorities, and *partiality*. The latter is useful to allow parts of a connector to *skip* clock ticks; this is subtly different from idling, and particularly useful in distributed connector implementations. The details do not matter in this paper.

3 Problem

Informally, propagation of priorities entails the following:

> If a connector propagates the priority of a "superior" data-flow over an "inferior" one into the environment, it enacts the inferior data-flow only if: (i) another connector simultaneously propagates a priority into the environment, and (ii) the environment can facilitate only one of the two priorities—they are *conflicting*—and (iii) the environment chooses the other one. In all other cases, facilitated by the environment, the connector enacts the superior data-flow.

A connector can propagate priorities *downstream* (i.e., in the direction of data-flow), *upstream*, or in both directions.

The problem of formalizing propagation of priorities is perhaps best studied in terms of concrete connectors. To this end, the presentation of Reo so far is extended, as follows. First, Fig. 7 shows four new foundational primitives that *start* ($\mathsf{Sync}_{!>}$ and $\mathsf{Sync}_{<!}$) and *end* ($\mathsf{Sync}_{)}$ and $\mathsf{Sync}_{(}$) propagation of priorities. Second, the informal semantics of every primitive in Fig. 2 is extended with:

> "It always propagates others' priorities downstream and upstream, but never its own."

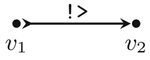
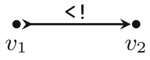
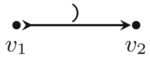
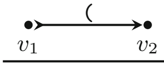
Name & Syntax	*Informal semantics*
$\mathsf{Sync}_{!>}$ 	Same data-flows and priorities as Sync (Fig. 2). It always propagates others' priorities downstream and upstream; it always propagates its own priority downstream.
$\mathsf{Sync}_{<!}$ 	Same data-flows and priorities as Sync (Fig. 2). It always propagates others' priorities downstream and upstream; it always propagates its own priority upstream.
$\mathsf{Sync}_{)}$ 	Same data-flows and priorities as Sync (Fig. 2). It always propagates others' priorities upstream; it never propagates priorities downstream
$\mathsf{Sync}_{(}$ 	Same data-flows and priorities as Sync (Fig. 2). It always propagates others' priorities downstream; it never propagates priorities upstream.

Fig. 7. Name, syntax, and informal semantics of priority primitives

4 Solution

Idea. The idea is to decompose the abstract concept of propagation of priorities into two more concrete auxiliary *metadata-flows*: one from a connector to the environment and one from the environment to the connector. Through the former, called *propagation metadata-flow*, a connector informs its environment on which shared vertices the environment *must* perform reciprocal `write`s and `take`s to facilitate the propagated priority of the connector; through the latter, called *conflict metadata-flow*, the environment informs the connector on which shared vertices it *cannot* perform reciprocal `write`s and `take`s, due to conflicting propagated priorities. The direction of metadata-flows is completely independent of the direction of data-flows: metadata can flow both upstream and downstream, whereas data can flow only downstream.

Now, the plan is to model metadata-flows using colors. The problem is that metadata-flows conceptually *precede* normal data-flows (i.e., they happen between clock ticks), which cannot be directly modeled in the connector coloring framework (i.e., the framework only models what happens at clock ticks). The solution is to conflate metadata-flows and normal data-flows.

To model propagation metadata-flows from a connector to the environment, I introduce three new activeness colors: $4, 5, 6$ (Fig. 4). In a coloring, entry $v \mapsto 4$ ($v \mapsto 5$) models that vertex v *is* active in the current data-flow, and *was* active in the preceding propagation metadata-flow downstream (upstream). To model metadata-flows from the environment to the connector, I introduce two new passiveness colors: $7, 8$ (Fig. 4). In a coloring, entry $v \mapsto 7$ ($v \mapsto 8$) models that vertex v is passive in the current data flow, but *was* active in the preceding conflict metadata-flow downstream (upstream); this means the environment cannot `write` (`take`) through v, because of conflicting priorities upstream (downstream). Thus, the new instantiation of the connector coloring framework has eight colors: $\{1, 2, 3, 4, 5, 6, 7, 8\}$.

Priority Primitives. Figure 8 shows the eight-color semantics of the new, priority primitives. Coloring γ_1 of $\mathsf{Sync}_{!>}$ models a data-flow from v_1 to v_2, preceded by a propagation metadata-flow downstream from v_2 (into the environment). Through this metadata-flow, $\mathsf{Sync}_{!>}$ informs the environment that it must perform a reciprocal `take` on v_2. Coloring γ_2 is similar to γ_1, except that the metadata-flow does not start at v_2, but further upstream; the metadata simply flows from v_1 to v_2. Coloring γ_3 is similar to γ_1, but beside modeling a propagation metadata-flow downstream from v_2 (into the environment), it also models a propagation metadata-flow upstream from v_2 to v_1. Coloring γ_4 combines γ_2 and γ_3. Colorings γ_5–γ_8 all model idling. Specifically, γ_5 and γ_7 permit idling if the environment cannot `write` through v_1, while γ_6 and γ_8 permit idling if the environment cannot `take` through v_2 because of conflicting propagated priorities. Note that there is no coloring that permits idling if the environment cannot `take` through v_2, *not* because of conflicting propagated priorities. The colorings of $\mathsf{Sync}_{<!}$ are symmetric.

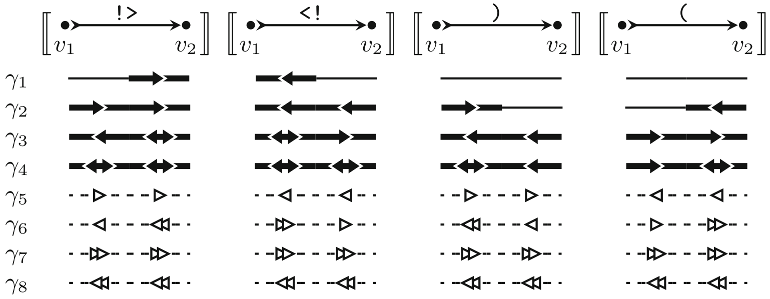


Fig. 8. Eight-color semantics of priority primitives

The key colorings of $\mathsf{Sync}_{)}$ are γ_2, γ_3, and γ_6. Coloring γ_2 models a data-flow from v_1 to v_2, preceded by a propagation metadata-flow downstream to v_1, *but no further*. In this way, $\mathsf{Sync}_{)}$ *blocks* propagation of priorities downstream. Coloring γ_3 models a data-flow from v_1 to v_2, preceded by a propagation metadata-flow upstream from v_2 to v_1. This shows that the blockade works only in one direction. Coloring γ_6 models idling, *supposedly* caused by conflicting propagated priorities. However, such a conflict does not really exist: $\mathsf{Sync}_{)}$ only *pretends* it has a conflict, to enable anyone further downstream to truly ignore priorities propagated through v_1, as part of its blockade. The colorings of $\mathsf{Sync}_{(}$ are symmetric.

Common Primitives. Figure 9 shows the eight-color semantics of the existing, common primitives (unary and binary); a "+M" annotation below a coloring means that the "horizontally mirrored" version of that coloring is part of the semantics as well. I highlight two salient aspects. First, the three-color semantics of every primitive [CCA07, Cos10] is strictly contained in its eight-color semantics (cf. the three-color semantics of ExclDrain and LossySync in Fig. 6). Second, coloring γ_4 of ExclDrain is a premier example of a propagation metadata-flow (from connector to environment) that induces a conflict metadata-flow (from environment to connector).

Figure 9 shows the eight-color semantics of the existing, common primitives (ternary). Again, the eight-color semantics strictly contain the three-color semantics. The interesting colorings are γ_6, γ_{16}, and γ_{12}–γ_{14} of Merger. Coloring γ_6 and γ_{16} are similar to coloring γ_4 of ExclDrain. Colorings γ_{12}–γ_{14} are notable, because they model propagation metadata-flows, but no conflict metadata-flows, in contrast to colorings γ_6 and γ_{16}. This is because propagation metadata-flows upstream have no bearing on the choices made by Merger: regardless of whether Merger chooses one of γ_{12}–γ_{14}, or one of their "vertically mirrored" versions, shared vertex v_3 is *always* active; this is all the propagated priority needs.

Next, to evaluate whether the eight-color semantics of the primitives compose as expected, I state and prove a number of eight-color semantics equalities.

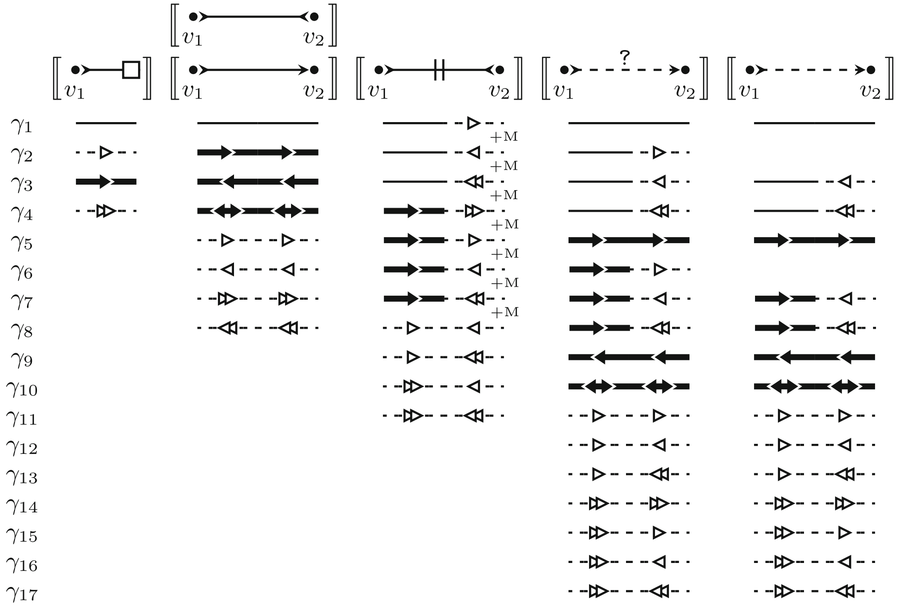
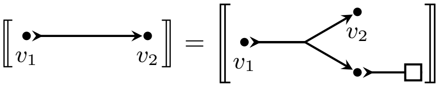


Fig. 9. Eight-color semantics of common unary and binary primitives

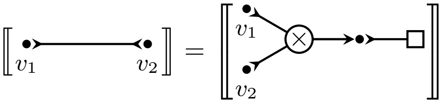
5 Evaluation

Basic Properties of Common Primitives. The following four propositions state that the common binary primitives in Fig. 2 (except LossySync) can be constructed out of unary and ternary primitives.[4]

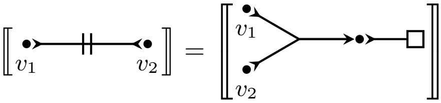
Proposition 1.



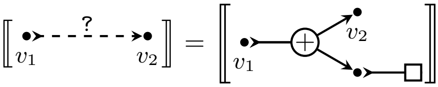
Proposition 2.



Proposition 3.



Proposition 4.



[4] All propositions in this paper should be interpreted modulo application of an *hide operator*, to remove internal vertices from the domains of colorings. This is straightforward to explicitly formalize.

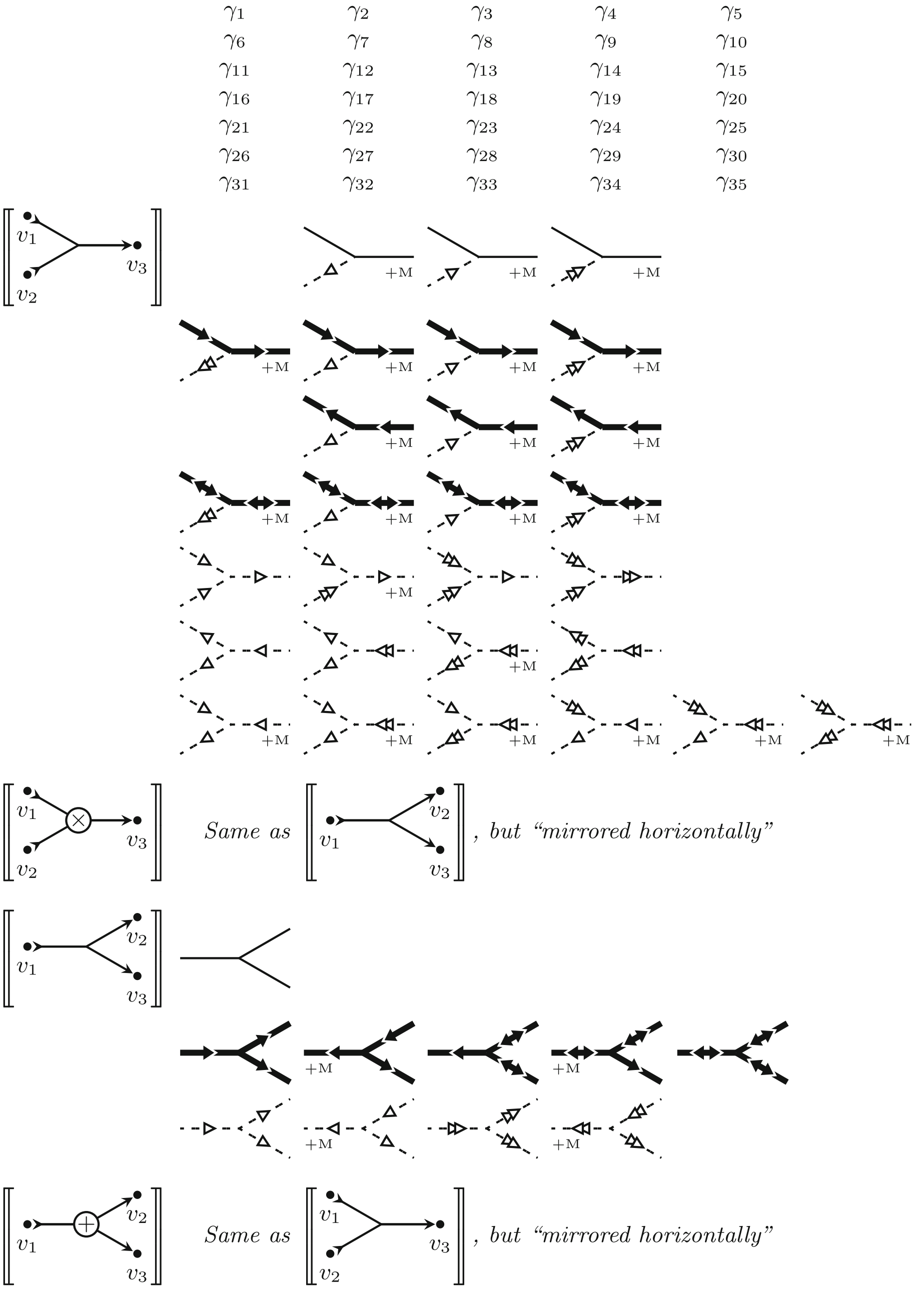
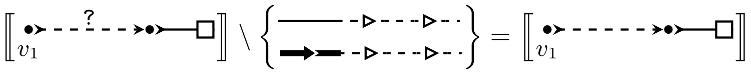

γ_1 γ_2 γ_3 γ_4 γ_5
γ_6 γ_7 γ_8 γ_9 γ_{10}
γ_{11} γ_{12} γ_{13} γ_{14} γ_{15}
γ_{16} γ_{17} γ_{18} γ_{19} γ_{20}
γ_{21} γ_{22} γ_{23} γ_{24} γ_{25}
γ_{26} γ_{27} γ_{28} γ_{29} γ_{30}
γ_{31} γ_{32} γ_{33} γ_{34} γ_{35}
v_1
v_2
v_3
+M
Same as
, but "mirrored horizontally"

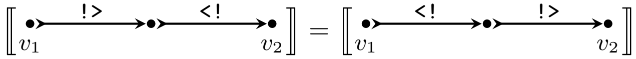
Fig. 10. Eight-color semantics of common ternary primitives

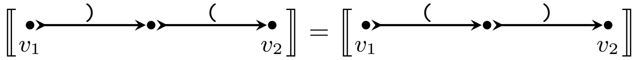
The following proposition states that $\mathsf{LossySync}_{?}$ and $\mathsf{LossySync}$ behave differently when composed with Drain. Specifically, according to its eight-color semantics, $\mathsf{LossySync}_{?}$ can (nondeterministically choose to) lose data before it reaches Drain, which $\mathsf{LossySync}$ cannot. This difference in semantics is intended: $\mathsf{LossySync}$ prioritizes not-losing over losing, whereas $\mathsf{LossySync}_{?}$ does not.

Proposition 5.

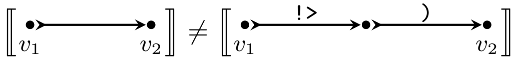


Basic Properties of Priority Primitives. The following two propositions state that $\mathsf{Sync}_{!>}$ and $\mathsf{Sync}_{<!}$, and $\mathsf{Sync}_{)}$ and $\mathsf{Sync}_{(}$, commute. Both compounds have the same data-flows and priorities as Sync in Fig. 2. But, the former compound always propagates priorities downstream and upstream, whereas the latter compound connector, in contrast, never propagates priorities downstream or upstream.

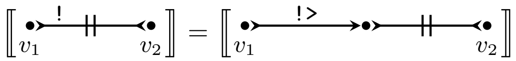
Proposition 6.


Proposition 7.


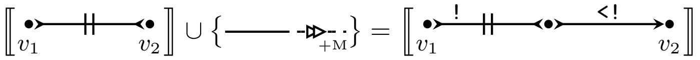
The following proposition states that $\mathsf{Sync}_{)}$ is not the "inverse" of $\mathsf{Sync}_{!>}$: starting and ending propagation of priorities is not "neutral". The reason is that $\mathsf{Sync}_{)}$ ends the downstream propagation of *all* priorities; not just those of $\mathsf{Sync}_{!>}$.

Proposition 8.


Imagine a variant of $\mathsf{ExclDrain}$ that, informally, has the same data-flows and priorities as $\mathsf{ExclDrain}$ in Fig. 2, but additionally prioritizes (1) over (2). The following proposition states that this connector, called $\mathsf{ExclDrain}_{!}$ in Fig. 11, can be constructed out of $\mathsf{Sync}_{!>}$ and $\mathsf{ExclDrain}$.

Proposition 9.


The following proposition states that conflicting propagated priorities "cancel out": the composition of $\mathsf{ExclDrain}_{!}$ and $\mathsf{Sync}_{<!}$ is almost the same as $\mathsf{ExclDrain}$. The only difference is that the compound is *saturated*: the extra coloring (cf. $\mathsf{ExclDrain}$) means that the compound can always ignore propagated priorities, by pretending there is a conflict. As a result, it is actually impossible to (re)construct $\mathsf{ExclDrain}_{!}$ from the compound.

Proposition 10.


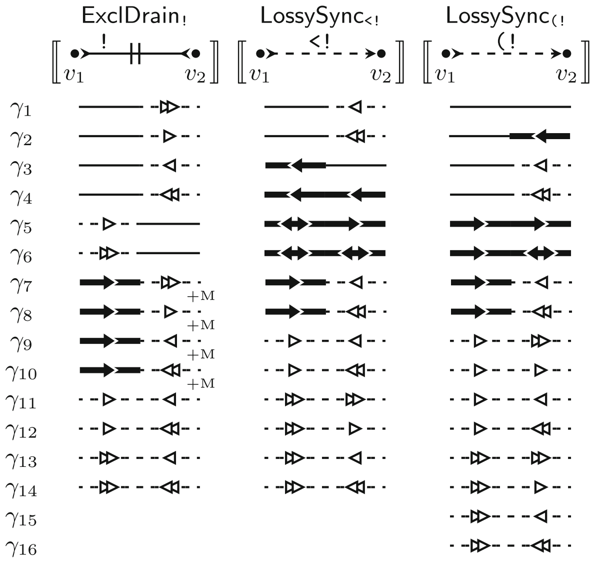


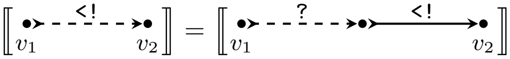
Fig. 11. Eight-color semantics of additional priority primitives

Advanced Properties: Context-Sensitivity. Perhaps *the* litmus test for any formalization of propagation of priority is the construction of the context-sensitive LossySync out of the nondeterministic $\mathsf{LossySync}_?$ and the priority primitives.

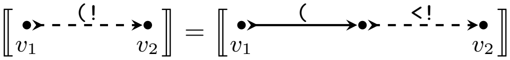
The construction proceeds in three steps. First, compose $\mathsf{LossySync}_?$ and $\mathsf{Sync}_{<!}$. The idea is that, through propagation of its own priorities, the latter forces the former to prioritize not-losing over losing. This works, but there is an undesirable side effect: the compound connector, called $\mathsf{LossySync}_{<!}$ in Fig. 11, propagates its own priorities upstream, which $\mathsf{LossySync}_?$ does not. To solve this, second, compose $\mathsf{Sync}_{(}$ and $\mathsf{LossySync}_{<!}$. The idea is that the former blocks the upstream propagation of $\mathsf{LossySync}_{<!}$'s priorities. This works, but there is again an undesirable side effect: the compound connector, called $\mathsf{LossySync}_{(!}$ in Fig. 11, blocks the upstream propagation of *all* priorities (cf. Proposition 15). To solve this, finally, compose $\mathsf{LossySync}_{(!}$ with ExclRouter and Merger. The idea is that the upstream propagation of others' priorities is not blocked, essentially because the propagation can proceed via a different upstream path through the graph.

The following propositions state that using the eight-color semantics, this construction *roughly* works: the only discrepancy is the presence of two colorings in the eight-color semantics of the final compound—absent in the eight-color semantics of LossySync—that model partial metadata-flows. This is an interesting phenomenon: relative to the informal semantics, the colorings are not wrong. They essentially mean that it is not really necessary to propagate priorities upstream, if a data-flow from vertex v_1 to vertex v_2 is *already* possible without such propagation. Through the construction of LossySync, this property "incidentally" emerges. I conjecture that if this property is consistently included in the eight-color semantics of all primitives, including LossySync, the resulting formalization of propagation of priority fully passes this litmus test.

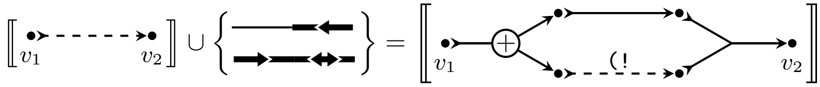
Proposition 11.



Proposition 12.

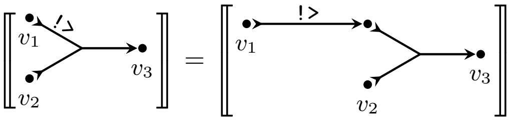


Proposition 13.



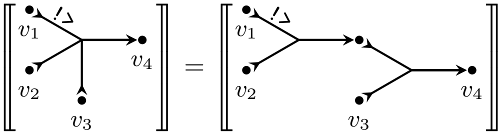
Advanced Properties: Ranks. Imagine a variant of Merger that, informally, has the same data-flows and priorities as Merger in Fig. 2, but additionally prioritizes (1) over (2). The following proposition states that this primitive, called $\mathsf{Merger}_{!>}$, can be composed out of $\mathsf{Sync}_{!>}$ and Merger (cf. Proposition 8).

Proposition 14.



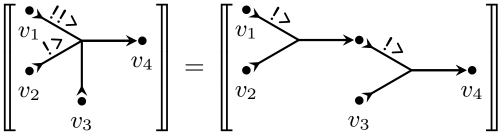
Imagine a variant of Merger with three sources instead of two. Informally, it has a data-flow from each of its sources to its targets, one of which it prioritizes over the other two. The following proposition states that this primitive, called $\mathsf{Merger3}_{!>}$, can be composed out of $\mathsf{Merger}_{!>}$ and Merger.

Proposition 15.



Imagine a variant of $\mathsf{Merger3}_{!>}$ with three sources instead of two. Informally, it has a data-flow from each of its sources to its targets, one of which it prioritizes over the other two (rank #1), and one of those two (rank #2) of which it prioritizes over the other one (rank #3). The following proposition states that this primitive, called $\mathsf{Merger3}_{!!>,!>}$, can be composed out of $\mathsf{Merger}_{!>}$ and $\mathsf{Merger}_{!>}$.

Proposition 16.



6 Discussion

I conclude this paper with some open issues and future work. Section 5 revealed already one open issue, namely the minor discrepancy between $\mathsf{LossySync}$ the primitive and $\mathsf{LossySync}$ the compound.

A second issue with the current formalization is exemplified by the connector in Fig. 1b: the eight-color semantics of this compound contains only one coloring that models idling, and moreover, this coloring has a *causality loop* (i.e., it is non-constructive, in Costa's sense [Cos10]). This problem is surprisingly difficult to solve in a proper way; the obvious solution (adding coloring $\{v_1 \mapsto 3, v_2 \mapsto 3, v_3 \mapsto 3\}$) has quite adverse side effects. Perhaps the problem can be solved by adding one or more colors.

The eight-color semantics of the connector in Fig. 1c allows for a nondeterministic choice between an "upper" data-flow (from v_1 to v_3) and a "lower" data-flow (from v_1 to v_4 and v_3), because $\mathsf{Sync}_{!>}$'s priorities are propagated only downstream, not affecting the nondeterministic choice of $\mathsf{ExclRouter}$, upstream. This is a reasonable interpretation of the informal semantics. An alternative interpretation, and arguably equally reasonable, is that Merger should propagate priorities from v_5 not only to v_3 but also to v_6, reversing the direction of propagation from downstream to upstream. Under this interpretation, the nondeterministic choice of $\mathsf{ExclRouter}$ *is* affected by $\mathsf{Sync}_{!>}$'s priorities, and the lower data-flow should never be chosen. It would be interesting to investigate how to model this alternative interpretation in the connector coloring framework.

Finally, the eight-color semantics of primitives and compounds quickly become prohibitively large. This makes manually reasoning about these semantics quite challenging. The development of software tooling to automate the composition of sets of colorings is imperative to continue this line of research.

A Definitions

Definition 1 (Structure). *$\mathbb{V}$ is the set of all vertices. $\mathbb{T}$ is the set of all types. The structure of a connector is a tuple $g = (V, E)$, where $V \subseteq \mathbb{V}$ and $E \subseteq (2^V \times \mathbb{T} \times 2^V) \setminus \{(\emptyset, t, \emptyset) \mid t \in \mathbb{T}\}$. $\mathbb{G}$ is the set of all structures.*

Definition 2 (Structural composition). $\mathsf{S}, \mathsf{T} : 2^{(2^{\mathbb{V}} \times \mathbb{T} \times 2^{\mathbb{V}})} \to 2^{\mathbb{V}}$ *are the functions defined by the following equations:*

$$\begin{aligned} \mathsf{S}(E) &= \bigcup\{V \mid (V, t, V') \in E\} \setminus \bigcup\{V' \mid (V, t, V') \in E\} \\ \mathsf{T}(E) &= \bigcup\{V' \mid (V, t, V') \in E\} \setminus \bigcup\{V \mid (V, t, V') \in E\} \end{aligned}$$

$\bowtie : \mathbb{G} \times \mathbb{G} \to \mathbb{G}$ *is the partial operation defined by the following equation:*

$$(V_1, E_1) \bowtie (V_2, E_2) = \begin{cases} (V_1 \cup V_2, E_1 \cup E_2) & \text{if: } \mathsf{S}(E_1) \cap \mathsf{T}(E_2) = \mathsf{S}(E_2) \cap \mathsf{T}(E_1) \\ \bot & \text{otherwise} \end{cases}$$

Definition 3 (Behavior). *$\mathbb{C}$ is the set of all colors. A coloring γ over V is a function from V to $\mathbb{C}$. $\mathbb{C}\textsc{ol}(V) = V \to \mathbb{C}$ is the set of all colorings over V. The behavior of a connector (V, E) is a set $\mathit{\Gamma} \subseteq \mathbb{C}\textsc{ol}(V)$ of colorings.*

Definition 4 (Behavioral composition).
$\bowtie \, : (\mathbb{C}\textsc{ol}(V_1) \times \mathbb{C}\textsc{ol}(V_2) \rightharpoonup \mathbb{C}\textsc{ol}(V_1 \cup V_2)) \cup (2^{\mathbb{C}\textsc{ol}(V_1)} \times 2^{\mathbb{C}\textsc{ol}(V_2)} \rightharpoonup 2^{\mathbb{C}\textsc{ol}(V_1 \cup V_2)})$ *is the partial function defined by the following equations:*

$$\gamma_1 \bowtie \gamma_2 = \begin{cases} \gamma_1 \cup \gamma_2 & \text{if: } \gamma_1(p) = \gamma_2(p) \text{ for-all } p \in \mathit{dom}(\gamma_1) \cap \mathit{dom}(\gamma_2) \\ \bot & \text{otherwise} \end{cases}$$

$$\mathit{\Gamma}_1 \bowtie \mathit{\Gamma}_2 = \{\gamma_1 \bowtie \gamma_2 \mid \gamma_1 \in \mathit{\Gamma}_1 \text{ and } \gamma_2 \in \mathit{\Gamma}_2 \text{ and } \gamma_1 \bowtie \gamma_2 \in \mathit{dom}(\bowtie)\}$$

Definition 5 (Denotation). *With $\mathcal{T} : \mathbb{T} \to (2^{\mathbb{V}} \times 2^{\mathbb{V}}) \to \bigcup\{2^{\mathbb{C}\textsc{ol}(V)} \mid V \subseteq \mathbb{V}\}$, $[\![\cdot]\!] : \mathbb{G} \to \bigcup\{\mathbb{C}\textsc{ol}(V) \mid V \subseteq \mathbb{V}\}$ is the function defined by the following equation:*

$$[\![(V, E)]\!] = \mathop{\Large\bowtie}\{\mathcal{T}(t)(V, V') \mid (V, t, V') \in E\}$$

Theorem 1. $[\![g_1 \bowtie g_2]\!] = [\![g_1]\!] \bowtie [\![g_2]\!]$

References

[Arb04] Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)

[Arb11] Arbab, F.: Puff, the magic protocol. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 169–206. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24933-4_9

[CCA07] Clarke, D., Costa, D., Arbab, F.: Connector colouring I: synchronisation and context dependency. Sci. Comput. Program. **66**(3), 205–225 (2007)

[Cos10] Costa, D.: Formal models for component connectors. Ph.D. thesis, Vrije Universiteit (2010)

[CP12] Clarke, D., Proença, J.: Partial connector colouring. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 59–73. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30829-1_5

[JKA11] Jongmans, S.-S.T.Q., Krause, C., Arbab, F.: Encoding context-sensitivity in Reo into non-context-sensitive semantic models. In: De Meuter, W., Roman, G.-C. (eds.) COORDINATION 2011. LNCS, vol. 6721, pp. 31–48. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21464-6_3

[Jon18] Jongmans, S.-S.: Formalizing propagation of priorities in Reo, using eight colors (technical report). Technical report (2018). www.arxiv.org

Learning to Coordinate

Gerco van Heerdt[1](✉), Bart Jacobs[2], Tobias Kappé[1], and Alexandra Silva[1]

[1] Department of Computer Science, University College London, London, UK
gercovanheerdt@gmail.com

[2] Institute for Computing and Information Sciences, Radboud University Nijmegen, Nijmegen, The Netherlands

Coördineren kun je leren — dedicated to Farhad Arbab on the occasion of his retirement

Abstract. Reo is a visual language of connectors that originated in component-based software engineering. It is a flexible and intuitive language, yet powerful and capable of expressing complex patterns of composition. The intricacies of the language resulted in many semantic models proposed for Reo, including several automata-based ones.

In this paper, we show how to generalize a known active automata learning algorithm—Angluin's L*—to Reo automata. We use recent categorical insights on Angluin's original algorithm to devise this generalization, which turns out to require a change of base category.

1 Introduction

In the last two decades, with the widespread use and development of software, there has been a focus on promoting reusability of software code. Component-based software engineering and service-oriented computing are two examples of paradigms that were developed around this idea. Many languages appeared to enable flexible and expressive ways to compose software components. One of those languages is Reo—a language offering a visual approach, where *connectors* are used to compose components into a system. The language is modular, offering ways to *compositionally* build more complex connectors from basic ones, which makes it possible to capture intricate patterns of interaction such as input synchronization, mutual-exclusion, or state-dependent behavior.

Reo serves as a prime example of a language in which interaction is treated as a first-class concept that allows direct specification and manipulation of protocols. The treatment of interaction as a central concept and the development of rigorous mathematical tools and techniques for its study has occupied most of Farhad's career. In this paper we make a modest contribution to Farhad's toolkit—a generalization of Angluin's algorithm to Reo automata, one of many semantic models developed for Reo.

Automata are used in modeling and verification of systems and protocols in Computer Science. Typically, the behavior of the system is modeled by a finite

F. de Boer et al. (Eds.): Arbab Festschrift, LNCS 10865, pp. 139–159, 2018.
https://doi.org/10.1007/978-3-319-90089-6_10

state machine and then desired properties, encoded in an appropriate logic, are checked against the model. Unfortunately, models are not always available and rapid changes in the system require frequent adaptations. This has lead to the development of *automata learning* algorithms, which enable inferring or learning a model from a given system just by observing its behavior or response to certain queries. One of the first algorithms was proposed by Angluin [2] and though it only worked for deterministic finite automata it had an interesting range of applications, such as in verification of software systems and security protocols (a recent survey can be found here [11]).

Category theory provides an abstract framework to study structures in mathematics and computer science. In this paper, we explore the power of abstraction and recast the main ingredients of Angluin's algorithm using basic categorical concepts, from algebra and coalgebra, which open the door to instantiations to other types of automata and in other categories.

Section 2 is a refinement of a section we included in a previous paper [7]. Compared to [7], we added fully categorical characterizations of the data structures and *hypothesis automata* involved in Angluin's algorithm. Using these ingredients, we made our proof of minimality of these automata completely abstract, which also fixes a gap present in the old proof. The application to Reo automata is entirely new and illustrates an important feature of the framework—the ability to have a learning algorithm in a different category. Angluin's algorithm for Reo automata operates in the category **Posets**. This might not be surprising for Farhad and those familiar with the semantics of Reo and the idiosyncrasies of the semantics of interaction and concurrency—the order of actions (and signal flows) is an important part of correctly capturing the behavior of a Reo connector. However, this pleasantly surprised the authors, as it provided a simple yet non-trivial, outside the category **Sets**, application of the categorical understanding and generalization of Angluin's algorithm. In order to obtain the algorithm we had to understand Reo automata as coalgebras for a functor. In this process, we show that they are essentially automata in the category **Posets**. We will define a poset of interactions to be used as the alphabet for the (categorical) Reo automaton—this highlights the algebraic structure of the actions (signal flows) in Reo and exposes the fact that interaction is a first-class concept.

Organization of the Paper. The rest of the paper is organized as follows. In Sect. 2, we recall the basic ingredients of Angluin's algorithm for deterministic automata and show how we can recast them in a categorical language. In Sect. 3, we change the base category in which the automata are considered from **Sets** to **Posets**, obtaining in this manner an algorithm for Reo automata [6].

2 Automata Learning: The Basic Algorithm

In this section we explain the ingredients of Angluin's original algorithm for learning deterministic finite automata and rephrase them using basic categorical constructs.

Let us first introduce some notation and basic definitions. Let A be a finite set of symbols, often called an alphabet, and A^* the set of finite words over A. We use λ to denote the empty word and, given two words $u, v \in A^*$, uv denotes their concatenation.

A language over A is a subset of words in A^*, that is $L \in 2^{A^*}$. We often switch between the representation of a language as a set and as its characteristic function. Given a language L and a word $u \in A^*$, we write $L(u)$ to denote 1 if $u \in L$ and 0 otherwise.

Given two languages U and V, we will denote by $U \cdot V$ (or simply UV) the concatenation of the two languages $U \cdot V = \{uv \mid u \in U, v \in V\}$. Given a language L and $a \in A$ we can define its left and right derivative by setting

$$a^{-1}L = \{u \mid au \in L\} \qquad \text{and} \qquad La^{-1} = \{u \mid ua \in L\}.$$

A language L is *prefix-closed* (resp. *suffix-closed*) if $La^{-1} \subseteq L$ (resp. $La^{-1} \subseteq L$) for all $a \in A$. We use $\downarrow u$ (resp. $\uparrow u$) to denote the set of prefixes (resp. suffixes) of a word $u \in A^*$.

$$\downarrow u = \{w \in A^* \mid w \text{ is a prefix of } u\} \qquad\qquad \uparrow u = \{w \in A^* \mid w \text{ is a suffix of } u\}$$

For the rest of this paper we fix a language $\mathcal{L} \in 2^{A^*}$ to be learned: the master language. This learning means that we seek a finite deterministic automaton that accepts $\mathcal{L}$. Many definitions and results are parametric in $\mathcal{L}$ but we do not always make this explicit.

2.1 Observation Tables

Angluin's algorithm incrementally constructs an observation table with Boolean entries. Rows are labeled by words in $S \cup S \cdot A$, where S is a finite *prefix-closed* language, and columns by a finite *suffix-closed* language E. Both S and E contain the empty word λ.

For arbitrary $U, V \subseteq A^*$, define $row\colon U \to 2^V$ by $row(u)(v) = \mathcal{L}(uv)$. Since row is fully determined by $\mathcal{L}$, we will from now on refer to an observation table as a pair (S, E), leaving $\mathcal{L}$ implicit. Formally, an observation table is a triple (S, E, row), where $row\colon (S \cup S \cdot A) \to 2^E$. Note that $\cup$ here is used for language union and not coproduct, but it will be convenient for us to split the function into $row\colon S \to 2^E$ and $row_A\colon S \cdot A \to 2^E$; handling the overlap between those efficiently is merely a practical consideration.

We can capture this structure more abstractly by observing that $\mathcal{L}$ induces a unique coalgebra homomorphism $l\colon A^* \to 2^{A^*}$, as shown in the following diagram.

$$\begin{tikzcd}
 & A^* \arrow[r, "c"] \arrow[dl, "\mathcal{L}"'] \arrow[d, dotted, "l"] & (A^*)^A \arrow[d, dotted, "l^A"] \\
2 & 2^{A^*} \arrow[l, "\lambda?"'] \arrow[r, "\partial"] & (2^{A^*})^A
\end{tikzcd}$$

Let us define the unknown ingredients in this diagram. On top we have A^* with a transition structure given by appending a letter to the end of the word:

$$c(u)(a) = ua.$$

On the bottom we have 2^{A^*}, the set of languages over A, with a transition structure given by the Brzozowski/left derivative of a language:

$$\partial(L)(a) = a^{-1}L = \{u \mid au \in L\}.$$

The map $\lambda?$ determines the inclusion of the empty word in the language: $\lambda?(L) = L(\lambda)$. The set 2^{A^*}, together with these two functions, is the final coalgebra of the functor $2 \times (-)^A$ on **Sets**. The map of coalgebras $l\colon A^* \to 2^{A^*}$ thus exists and is unique by finality. More concretely, it is given for all $u, v \in A^*$ by

$$l(u)(v) = \mathcal{L}(uv). \tag{1}$$

Note that we used the functional view on $\mathcal{L}$ here. The set A^*, together with the map c and the empty word $\lambda\colon 1 \to A^*$, is the initial algebra of the functor $1 + A \times -$; we could equivalently define l through initiality by regarding $\mathcal{L}$ as an element of 2^{A^*}.

In the following lemma we use the inclusion map $n\colon S \hookrightarrow A^*$ and the function $k\colon 2^{A^*} \to 2^E$ defined by $k(L)(e) = L(e)$ for every $L \in 2^{A^*}$ and $e \in E$. These can be seen as representations of the subsets S and E. Furthermore, it is convenient to use the curried version $\Lambda(row_A)\colon S \to (2^E)^A$ of row_A given by $\Lambda(row_A)(s)(a) = row_A(sa)$.

Lemma 1. *Given S and E, the observation table is defined by $row = k \circ l \circ n\colon S \to 2^E$ and $\Lambda(row_A) = k^A \circ \partial \circ l \circ n\colon S \to (2^E)^A$.*

Proof. For all $s \in S$ and $e \in E$, we have

$$\begin{aligned}
(k \circ l \circ n)(s)(e) &= (k \circ l)(s)(e) && \text{definition of } n \\
&= l(s)(e) && \text{definition of } k \\
&\overset{(1)}{=} \mathcal{L}(se)
\end{aligned}$$

and for each $a \in A$,

$$\begin{aligned}
(k^A \circ \partial \circ l \circ n)(s)(a)(e) &= k((\partial \circ l \circ n)(s)(a))(e) \\
&= (\partial \circ l \circ n)(s)(a)(e) && \text{definition of } k \\
&= (l \circ n)(s)(ae) && \text{definition of } \partial \\
&= l(s)(ae) && \text{definition of } n \\
&\overset{(1)}{=} \mathcal{L}(sae).
\end{aligned}$$

Thus, this yields $row(s)(e) = \mathcal{L}(se)$ and $row_A(sa)(e) = \Lambda(row_A)(s)(a)(e) = \mathcal{L}(sae)$, which is precisely the original definition. □

There are two crucial properties of the observation table that play a key role in the algorithm of [2], allowing for the construction of a deterministic automaton from an observation table: closedness and consistency.

Definition 1 (Closed and Consistent Table [2]**).** *An observation table* (S, E) *is* closed *if for all* $t \in S \cdot A$ *there exists an* $s \in S$ *such that* $row_A(t) = row(s)$*. An observation table* (S, E) *is* consistent *if whenever* s_1 *and* s_2 *are elements of* S *such that* $row(s_1) = row(s_2)$*, for all* $a \in A$*,* $row_A(s_1 a) = row_A(s_2 a)$*.*

In many categories each map $f\colon A \to B$ can be factored as $f = (A \twoheadrightarrow \bullet \rightarrowtail B)$, describing f as an epimorphism followed by a monomorphism. In the category **Sets** of sets and functions epimorphisms (resp. monomorphisms) are surjections (resp. injections). Using these factorizations we come to the following categorical reformulations.

Lemma 2. *An observation table* (S, E) *is closed (resp. consistent) if and only if there exists a necessarily unique map* i *(resp.* j*) such that the diagram on the left (resp. right) commutes.*

$$\begin{tikzcd}
S \cdot A \arrow[drr, bend left, "row_A"] \arrow[dr, dotted, "i"] & & \\
& \bullet \arrow[r, tail, "m"] & 2^E \\
S \arrow[ur, two heads, "e"'] \arrow[urr, bend right, "row"'] & &
\end{tikzcd}
\qquad
\begin{tikzcd}
& & 2^E \\
S \arrow[urr, bend left, "row"] \arrow[r, two heads, "e"] \arrow[drr, bend right, "\Lambda(row_A)"'] & \bullet \arrow[ur, tail, "m"] \arrow[dr, dotted, "j"'] & \\
& & (2^E)^A
\end{tikzcd}$$

closed *consistent*

Proof. Suppose the table is closed according to Definition 1. Then, for every $t \in S \cdot A$ there exists an $s \in S$ such that $row(s) = row_A(t)$. We define i by $i(t) = e(s)$. It remains to show that $m \circ i = row_A$.

$$\begin{aligned}
(m \circ i)(t) &= (m \circ e)(s) && \text{definition of } i \\
&= row(s) && \text{factorization of } row \\
&= row_A(t) && \text{closedness assumption.}
\end{aligned}$$

The uniqueness of i is immediate using that m is monic.

Conversely, suppose that there exists i such that $m \circ i = row_A$ and let $t \in S \cdot A$. Take s such that $e(s) = i(t)$ (which exists since e is epi). We need to show $row(s) = row_A(t)$.

$$\begin{aligned}
row(s) &= (m \circ e)(s) && \text{factorization of } row \\
&= (m \circ i)(t) && \text{assumption } e(s) = i(t) \\
&= row_A(t) && \text{assumption } m \circ i = row_A.
\end{aligned}$$

Suppose the table is consistent according to Definition 1. That is, if $s_1, s_2 \in S$ are such that $row(s_1) = row(s_2)$ then, for all $a \in A$, it holds that $row_A(s_1 a) = row_A(s_2 a)$. We define j by $j(e(s)) = \Lambda(row_A)(s)$, using that e is epi. By definition, $j \circ e = \Lambda(row_A)$. It remains to show that j is well-defined. Let s_1, s_2 be such that $e(s_1) = e(s_2)$. We need to show $\Lambda(row_A)(s_1) = \Lambda(row_A)(s_2)$.

$$\begin{aligned}
e(s_1) = e(s_2) &\Rightarrow row(s_1) = row(s_2) && \text{definition of } row \\
&\Rightarrow \forall a \in A.\, row_A(s_1 a) = row_A(s_2 a) && \text{consistency assumption} \\
&\Rightarrow \Lambda(row_A)(s_1) = \Lambda(row_A)(s_2) && \text{definition of } \Lambda.
\end{aligned}$$

The uniqueness of j follows directly from the fact that e is epi.

Conversely, suppose that there exists j such that $j \circ e = \Lambda(row_A)$, and let $s_1, s_2 \in S$ be such that $row(s_1) = row(s_2)$. Note that this is equivalent to $e(s_1) = e(s_2)$ because m is monic. We need to show $row_A(s_1a) = row_A(s_2a)$ for all $a \in A$, or, equivalently, $\Lambda(row_A)(s_1) = \Lambda(row_A)(s_2)$.

$$\begin{aligned} \Lambda(row_A)(s_1) &= (j \circ e)(s_1) && \text{assumption } \Lambda(row_A) = j \circ e \\ &= (j \circ e)(s_2) && \text{assumption } e(s_1) = e(s_2) \\ &= \Lambda(row_A)(s_2) && \text{assumption } \Lambda(row_A) = j \circ e. \end{aligned}$$

□

Closed and consistent observation tables are important in the algorithm of [2] because they can be translated into a deterministic automaton. We first describe the construction concretely and subsequently more abstractly using our categorical reformulation.

Definition 2 (Automaton associated with an observation table [2]). *Given a closed and consistent observation table (S, E) one can construct a deterministic automaton $M(S, E) = (Q, q_0, \delta, F)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, $q_0 \in Q$ is the initial state, and $\delta\colon Q \times A \to Q$ is the transition function. These are given by:*

$$\begin{aligned} Q &= \{row(s) \mid s \in S\} & q_0 &= row(\lambda) \\ F &= \{row(s) \mid s \in S, row(s)(\lambda) = 1\} & \delta(row(s), a) &= row_A(sa). \end{aligned}$$

To see that this is a well-defined automaton we need to check two facts: that F is a well-defined subset and that δ is a well-defined function.

Suppose s_1 and s_2 are elements of S such that $(\star)$ $row(s_1) = row(s_2)$. We must show

$$row(s_1) \in F \iff row(s_2) \in F \quad \text{and} \tag{2}$$

$$\delta(row(s_1), a) = \delta(row(s_2), a) \in Q, \text{ for all } a \in A. \tag{3}$$

We have:

$$row(s_1) \in F \iff row(s_1)(\lambda) = 1 \overset{(\star)}{\iff} row(s_2)(\lambda) = 1 \iff row(s_2) \in F.$$

This concludes the proof of (2) above. Since the observation table is consistent, we have for each $a \in A$ that $(\star)$ implies $row_A(s_1a) = row_A(s_2a)$, and hence we can calculate

$$\delta(row(s_1), a) = row_A(s_1a) = row_A(s_2a) = \delta(row(s_2), a).$$

It remains to show that $row_A(s_1a) \in Q$. Since the table is closed, there exists an $s \in S$ such that $row(s) = row_A(s_1a)$. Hence, $row_A(s_1a) \in Q$ and (3) above holds.

In our categorical reformulation of the construction of the automaton Q we use that epis/surjections and monos/injections in the category **Sets** form a factorization system (see e.g. [5]). This allows us to use the diagonal-fill-in property in the next result.

Lemma 3. *The transition function δ of the automaton associated with a closed and consistent observation table can be obtained as the unique diagonal in the following diagram,*

$$\begin{tikzcd}
S \arrow[r, two heads, "e"] \arrow[d, "\Lambda(i)"'] & Q \arrow[d, "j"] \arrow[dl, dashed, "\delta"'] \\
Q^A \arrow[r, tail, "m^A"] & (2^E)^A
\end{tikzcd}$$

Proof. The function δ obtained by diagonalization above satisfies:

$$\delta(e(s))(a) = \Lambda(i)(s)(a) = i(sa).$$

This is the same as the above definition of δ, since $e(s)$ and $i(sa)$ represent, respectively, $\mathit{row}(s)$ and $\mathit{row}_A(sa)$. □

Finally, the definitions of the initial and final states can be recovered from properties reminiscent of our reformulations of closedness and consistency.

Lemma 4. *The initial and final states can be obtained as the necessarily unique maps init and final making the diagrams below commute.*

$$\begin{tikzcd}
1 \arrow[dr, dotted, "\mathit{init}"] \arrow[drr, bend left, "k \circ l \circ \lambda"] & & \\
& Q \arrow[r, tail, "m"] & 2^E \\
S \arrow[ur, two heads, "e"'] \arrow[urr, bend right, "\mathit{row}"'] & &
\end{tikzcd}
\qquad
\begin{tikzcd}
& & 2^E \\
S \arrow[r, two heads, "e"] \arrow[urr, bend left, "\mathit{row}"] \arrow[drr, bend right, "\lambda? \circ l \circ n"'] & Q \arrow[ur, tail, "m"] \arrow[dr, dotted, "\mathit{final}"'] & \\
& & 2
\end{tikzcd}$$

Proof. We define $\mathit{init}(*) = e(\lambda)$ and $\mathit{final}(e(s)) = \mathit{row}(s)(\lambda)$ for all $s \in S$. These are equivalent to q_0 and F given in Definition 2 and satisfy

$$\begin{aligned}
(m \circ \mathit{init})(*) &= (m \circ e)(\lambda) && \text{definition of } \mathit{init} \\
&= \mathit{row}(\lambda) && \text{factorization of } \mathit{row} \\
&= (k \circ l \circ n)(\lambda) && \text{Lemma 1} \\
&= (k \circ l \circ \lambda)(*) && n(\lambda) = \lambda = \lambda(*)
\end{aligned}$$

and for all $s \in S$,

$$\begin{aligned}
(\mathit{final} \circ e)(s) &= \mathit{row}(s)(\lambda) && \text{definition of } \mathit{final} \\
&= (k \circ l \circ n)(s)(\lambda) && \text{Lemma 1} \\
&= (l \circ n)(s)(\lambda) && \text{definition of } k \\
&= (\lambda? \circ l \circ n)(s) && \text{definition of } \lambda?.
\end{aligned}$$

Uniqueness is again necessary because m is monic and e is epic. □

2.2 Minimal Conjectures

So far we have ignored the prefix-closedness of S and the suffix-closedness of E as they were not relevant for the construction of the automaton. The minimality result of [2], however, does depend on them. We encode these properties in two maps:

$$\begin{aligned} \rho &\colon S \to 1 + S \times A & \sigma &\colon 2 \times (2^E)^A \to 2^E \\ \rho(\lambda) &= * & \sigma(v,f)(\lambda) &= v \\ \rho(sa) &= (s,a) & \sigma(v,f)(ae) &= f(a)(e). \end{aligned}$$

The main point of these maps is that they come equipped with inductive principles corresponding to induction on the length of prefix- and suffix-closed words.

Lemma 5. *For each algebra $[\chi_1, \chi_A] \colon 1 + X \times A \to X$ there exists a unique function $f \colon S \to X$ making the diagram below on the left commute, and for every coalgebra $<\upsilon_2, \upsilon_A> \colon Y \to 2 \times Y^A$ there is a unique function $g \colon Y \to 2^E$ making the diagram below on the right commute.*

$$\begin{tikzcd} 1 + S \times A \arrow[r, dashed, "\mathrm{id}_1 + f \times \mathrm{id}_A"] & 1 + X \times A \arrow[d, "{[\chi_1, \chi_A]}"] \\ S \arrow[u, "\rho"] \arrow[r, dashed, "f"] & X \end{tikzcd} \qquad \begin{tikzcd} Y \arrow[r, dashed, "g"] \arrow[d, "<\upsilon_2, \upsilon_A>"'] & 2^E \\ 2 \times Y^A \arrow[r, dashed, "\mathrm{id}_2 \times g^A"] & 2 \times (2^E)^A \arrow[u, "\sigma"'] \end{tikzcd}$$

Proof. The requirement $f = [\chi_1, \chi_A] \circ (\mathrm{id}_1 + f \times \mathrm{id}_A) \circ \rho = [\chi_1, \chi_A \circ (f \times \mathrm{id}_A)] \circ \rho$ corresponds to a definition of f by induction on the length of words in S, thus providing existence and uniqueness:

$$\begin{aligned} f(\lambda) &= ([\chi_1, \chi_A \circ (f \times \mathrm{id}_A)] \circ \rho)(\lambda) &&= [\chi_1, \chi_A \circ (f \times \mathrm{id}_A)](*) &&= \chi_1(*) \\ f(sa) &= ([\chi_1, \chi_A \circ (f \times \mathrm{id}_A)] \circ \rho)(sa) &&= [\chi_1, \chi_A \circ (f \times \mathrm{id}_A)](s,a) &&= \chi_A(f(s), a). \end{aligned}$$

The condition $g = \sigma \circ (\mathrm{id}_2 \times g^A) \circ <\upsilon_2, \upsilon_A> = \sigma \circ <\upsilon_2, g^A \circ \upsilon_A>$ gives a definition of g by induction on the length of words in E:

$$\begin{aligned} g(y)(\lambda) &= (\sigma \circ <\upsilon_2, g^A \circ \upsilon_A>)(y)(\lambda) &&= \upsilon_2(y) \\ g(y)(ae) &= (\sigma \circ <\upsilon_2, g^A \circ \upsilon_A>)(y)(ae) &&= (g^A \circ \upsilon_A)(y)(a)(e) = g(\upsilon_A(y)(a))(e). \end{aligned}$$

□

One might wonder what the unique such maps into the initial algebra and out of the final coalgebra are. Our next result will be that these are precisely n and k, respectively, but first we need to be more explicit about what the initial algebra is. The reason that we can identify it with the maps λ and c is that the currying operation extends from concatenated languages to arbitrary products and has an inverse Ψ (uncurrying). Moreover, given $f \colon W \to X$, $g \colon X \times A \to Y$, and $h \colon Y \to Z$, we have

$$h^A \circ \Lambda(g) \circ f = \Lambda(h \circ g \circ (f \times \mathrm{id}_A)). \tag{4}$$

Explicitly, the initial algebra is the set A^* together with $[\lambda, \Psi(c)] \colon 1 + A^* \times A \to A^*$, where $\Psi(c)(u,a) = c(u)(a)$ for all $u \in A^*$ and $a \in A$.

Lemma 6. *The following diagrams commute.*

$$\begin{tikzcd}
1 + S \times A \arrow[r, "\mathrm{id}_1 + n \times \mathrm{id}_A"] & 1 + A^* \times A \arrow[d, "{[\lambda, \Psi(c)]}"] \\
S \arrow[u, "\rho"] \arrow[r, "n"] & A^*
\end{tikzcd}
\qquad
\begin{tikzcd}
2^{A^*} \arrow[r, "k"] \arrow[d, "<\lambda?, \partial>"'] & 2^E \\
2 \times (2^{A^*})^A \arrow[r, "\mathrm{id}_2 \times k^A"] & 2 \times (2^E)^A \arrow[u, "\sigma"']
\end{tikzcd}$$

Proof. Suppose f is the unique coalgebra-to-algebra morphism $S \to A^*$ provided by Lemma 5. Using the calculations in that lemma ($\star$) we prove by induction on the length of words in S that $f = n$:

$$\begin{aligned}
f(\lambda) &\overset{(\star)}{=} \lambda(*) &&= \lambda &&= n(\lambda) \\
f(sa) &\overset{(\star)}{=} \Psi(c)(f(s), a) &&\overset{(\mathrm{IH})}{=} \Psi(c)(n(s), a) &&= n(s)a = n(sa).
\end{aligned}$$

Similarly, let g be the unique coalgebra-to-algebra morphism $2^{A^*} \to 2^E$. Using induction on the length of words in E, we find that $g = k$:

$$\begin{aligned}
g(L)(\lambda) &\overset{(\star)}{=} \lambda?(L) &&= L(\lambda) &&= k(L)(\lambda) \\
g(L)(ae) &\overset{(\star)}{=} g(\partial(L)(a))(e) &&\overset{(\mathrm{IH})}{=} k(\partial(L)(a))(e) &&= \partial(L)(a)(e) = L(ae) = k(L)(ae).
\end{aligned}$$

□

An automaton is minimal if all states are reachable from the initial state and if no two different states recognize the same language (this property is referred to as observability).

Following this characterization that goes back to Kalman and was subsequently generalized by Arbib and Manes [3,9], these two properties can be nicely captured in the following diagram, where in the middle we have our automaton constructed from the observation table.

$$\begin{tikzcd}
1 \arrow[d, "\lambda"'] \arrow[dr, "init"] & & 2 \\
A^* \arrow[r, dashed, "r"] \arrow[d, "c"'] & Q \arrow[r, dashed, "o"] \arrow[d, "\delta"] \arrow[ur, "final"] & 2^{A*} \arrow[u, "\lambda?"'] \arrow[d, "\partial"] \\
(A^*)^A \arrow[r, dashed, "r^A"] & Q^A \arrow[r, dashed, "o^A"] & (2^{A*})^A
\end{tikzcd} \tag{5}$$

Recall that the structure on the left is the initial algebra. The map r thus exists and is unique by initiality; it sends every word to the state it reaches. The map o exists and is unique by finality; it assigns to every state the language it accepts.

Reachability and observability can now be rephrased in terms of properties of the functions r and o in (5): the automaton Q is *reachable* if $r\colon A^* \to Q$ is epic/surjective and it is *observable* if $o\colon Q \to 2^{A^*}$ is monic/injective.

Theorem 1. *The automaton associated with a closed and consistent observation table is minimal.*

Proof. We prove observability first because it is relatively straightforward. Note first that

$$m^A \circ \delta \circ e = j \circ e = \Lambda(row_A) = k^A \circ \partial \circ l \circ n, \tag{6}$$

using, from left to right, Lemmas 3, 2 and 1. Now observe that the diagram below on the left commutes by Lemma 6 and the definition of o,

$$\begin{tikzcd}
Q \arrow[r, "o"] \arrow[d, "<\mathit{final}, \delta>"'] & 2^{A^*} \arrow[r, "k"] \arrow[d, "<\lambda?, \partial>"] & 2^E \\
2 \times Q^A \arrow[r, "\mathrm{id}_2 \times o^A"] & 2 \times (2^{A^*})^A \arrow[r, "\mathrm{id}_2 \times k^A"] & 2 \times (2^E)^A \arrow[u, "\sigma"']
\end{tikzcd}
\qquad
\begin{tikzcd}
Q \arrow[r, "m"] \arrow[d, "<\mathit{final}, \delta>"'] & 2^E \\
2 \times Q^A \arrow[r, "\mathrm{id}_2 \times m^A"] & 2 \times (2^E)^A \arrow[u, "\sigma"']
\end{tikzcd}$$

and that the diagram on the right commutes because

$$\begin{aligned}
& \sigma \circ (\mathrm{id}_2 \times m^A) \circ <\mathit{final}, \delta> \circ\ e && \\
= {} & \sigma\ \circ <\mathit{final} \circ e, m^A \circ \delta \circ e> && \\
= {} & \sigma\ \circ <\lambda? \circ l \circ n, m^A \circ \delta \circ e> && \text{Lemma 4} \\
\overset{(6)}{=} {} & \sigma\ \circ <\lambda? \circ l \circ n, k^A \circ \partial \circ l \circ n> && \\
= {} & \sigma\ \circ (\mathrm{id}_2 \times k^A) \circ <\lambda?, \partial> \circ\ l \circ n && \\
= {} & k \circ l \circ n && \text{Lemma 6} \\
= {} & row && \text{Lemma 1} \\
= {} & m \circ e && \text{factorization of } row
\end{aligned}$$

and e is epic. From Lemma 5 it then follows that $k \circ o = m$, and hence o is monic.

For reachability we would like to do a similar proof, but as a result of a coalgebraic bias in some of our definitions the proof of

$$m \circ \Psi(\delta) \circ (e \times \mathrm{id}_A) = k \circ l \circ c \circ (n \times \mathrm{id}_A) \tag{7}$$

needs a little more work. In particular, it follows by using several times that Ψ and Λ are inverse to each other:

$$\begin{aligned}
m \circ \Psi(\delta) \circ (e \times \mathrm{id}_A) &= \Psi(\Lambda(m \circ \Psi(\delta) \circ (e \times \mathrm{id}_A))) && \\
&\overset{(4)}{=} \Psi(m^A \circ \Lambda(\Psi(\delta)) \circ e) && \\
&= \Psi(m^A \circ \delta \circ e) && \\
&\overset{(6)}{=} \Psi(k^A \circ \partial \circ l \circ n) && \\
&= \Psi(k^A \circ l^A \circ \Lambda(c) \circ n) && \text{definition of } l \\
&\overset{(4)}{=} \Psi(\Lambda(k \circ l \circ c \circ (n \times \mathrm{id}_A))) && \\
&= k \circ l \circ c \circ (n \times \mathrm{id}_A).
\end{aligned}$$

The diagram below on the left commutes by Lemma 6 and the definition of r,

$$\begin{tikzcd}
1 + S \times A \arrow[r, "\mathrm{id}_1 + n \times \mathrm{id}_A"] & 1 + A^* \times A \arrow[r, "\mathrm{id}_1 + r \times \mathrm{id}_A"] \arrow[d, "{[\lambda, \Psi(c)]}"'] & 1 + Q \times A \arrow[d, "{[\lambda, \Psi(\delta)]}"'] \\
S \arrow[u, "\rho"] \arrow[r, "n"] & A^* \arrow[r, "r"] & Q
\end{tikzcd}
\qquad
\begin{tikzcd}
1 + S \times A \arrow[r, "\mathrm{id}_1 + e \times \mathrm{id}_A"] & 1 + Q \times A \arrow[d, "{[\mathit{init}, \Psi(\delta)]}"'] \\
S \arrow[u, "\rho"] \arrow[r, "e"] & Q
\end{tikzcd}$$

and for the diagram on the right we simply note that

$$\begin{aligned}
& m \circ [\mathit{init}, \Psi(\delta)] \circ (\mathrm{id}_1 + e \times \mathrm{id}_A) \circ \rho && \\
={}& [m \circ \mathit{init}, m \circ \Psi(\delta) \circ (e \times \mathrm{id}_A)] \circ \rho && \\
\overset{(7)}{=}{}& [m \circ \mathit{init}, k \circ l \circ c \circ (n \times \mathrm{id}_A)] \circ \rho && \\
={}& [k \circ l \circ \lambda, k \circ l \circ c \circ (n \times \mathrm{id}_A)] \circ \rho && \text{Lemma 4} \\
={}& k \circ l \circ [\lambda, c] \circ (\mathrm{id}_1 + n \times \mathrm{id}_A) \circ \rho && \\
={}& k \circ l \circ n && \text{Lemma 6} \\
={}& \mathit{row} && \text{Lemma 1} \\
={}& m \circ e && \text{factorization of } \mathit{row}
\end{aligned}$$

and m is monic. From Lemma 5 it follows that $r \circ n = e$, so r must be epic. □

2.3 The Learning Algorithm

We present the algorithm of [2] in Fig. 1. In the algorithm, there is a teacher which has the capacity of answering two types of questions: yes/no to the query on whether a word belongs to the master language and yes/no to the question whether a certain guess of the automaton accepting the master language is correct. In the case of a negative answer of the latter question, the teacher also provides a counter-example. The learner builds an observation table by asking the teacher queries of membership of words of increasing length. Once the table is closed and consistent, the learner tries to guess the master language. We explain every step by means of an example, over the alphabet $A = \{a, b\}$.

Imagine the Learner receives as input a Teacher for the master language

$$\mathcal{L} = \{u \in \{a, b\}^* \mid \text{the number of } a\text{'s in } u \text{ is divisible by } 3\}.$$

In the first step of the while loop it builds a table for $S = \{\lambda\}$ and $E = \{\lambda\}$.

Step 1

	λ
λ	1
a	0
b	1

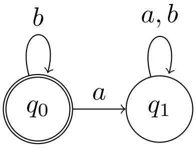
(S, E) consistent? ✓ (S, E) closed?
No, $\mathit{row}_A(a) = (\lambda \mapsto 0) \neq (\lambda \mapsto 1) = \mathit{row}(\lambda)$
Then, $S \leftarrow S \cup \{a\}$ and we go to **Step 2**.

We extend row index set S and we again check for closedness and consistency.

Step 2

	λ
λ	1
a	0
b	1
aa	0
ab	0

(S, E) consistent? ✓ (S, E) closed? ✓
Then, we guess the automaton:



where $q_0 = \mathit{row}(\lambda) = (\lambda \mapsto 0)$
$q_1 = \mathit{row}(a) = (\lambda \mapsto 1)$

Teacher replies with counter-example aaa.
$S \leftarrow S \cup \{\lambda, a, aa, aaa\}$ and we go to **Step 3**.

In the second step we managed to build a closed and consistent table which enabled us to make a first guess on the automaton. The guess was wrong so the

```
Input: Minimally Adequate Teacher of the master language $\mathcal{L}$.
Output: Minimal automaton accepting $\mathcal{L}$.
function LEARNER
    $S \leftarrow \{\lambda\}$; $E \leftarrow \{\lambda\}$.
    repeat
        while $(S, E)$ is not closed or not consistent do
            if $(S, E)$ is not consistent then
                find $s_1, s_2 \in S$, $a \in A$, and $e \in E$ such that
                    $row(s_1) = row(s_2)$ and $\mathcal{L}(s_1ae) \neq \mathcal{L}(s_2ae)$
                $E \leftarrow E \cup \{ae\}$.
            end if
            if $(S, E)$ is not closed then
                find $s_1 \in S$, $a \in A$ such that
                    $row_A(s_1a) \neq row_A(s)$, for all $s \in S$
                $S \leftarrow S \cup \{s_1a\}$.
            end if
        end while
        Make the conjecture $M(S, E)$.
        if the Teacher replies no to the conjecture, with a counter-example $t$ then
            $S \leftarrow S \cup \downarrow t$.
        end if
    until the Teacher replies yes to the conjecture $M(S, E)$.
    return $M(S, E)$.
end function
```

Fig. 1. Angluin's algorithm for deterministic finite automata [2]

Teacher provided a counter-example, which we use to extend the row index set, generating a larger table.

Step 3

	λ
λ	1
a	0
aa	0
aaa	1
b	1
ab	0
aab	0
$aaaa$	0
$aaab$	1

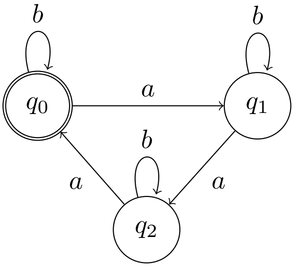
(S, E) consistent?
No, $row(a) = row(aa)$ but $row_A(aa) \neq row_A(aaa)$.
Then $E \leftarrow E \cup \{a\}$ and we go to (**Step 4**).

In the third step the test of consistency failed for the first time and hence we extend the column index set E from $\{\lambda\}$ to $\{\lambda, a\}$. This extension allows to distinguish states (that is, rows of the table) that were indistinguishable in the previous step though they could be differentiated after an a step.

Step 4

	λ	a
λ	1	0
a	0	0
aa	0	1
aaa	1	0
b	1	0
ab	0	0
aab	0	1
$aaaa$	0	0
$aaab$	1	0

(S,E) consistent? ✓ (S,E) closed? ✓
We make another guess:



The Teacher replies **yes**.

In the last step, we again constructed a closed and consistent table, which allowed us to make another guess of the automaton accepting the master language. This second guess yielded the expected automaton.

3 Application to Reo Automata

We now apply the categorical generalization of Angluin's algorithm derived in the previous sections to learn coordination protocols. Specifically, the type of automata that we learn will be Reo automata [6], one of the many semantics for the Reo coordination language. This presents a different generalization made possible by our formulation of Angluin's algorithm, namely by varying the base category for the coalgebra.

We need to find a suitable encoding of Reo automata; let us first recall their definition.

Definition 3. *Let Σ be a finite set of* ports*. A* Reo automaton *is a tuple $(Q, \rightarrow, q_0)$ where*

- *Q is a finite set of* states*, with $q_0 \in Q$ the* initial state
- *$\rightarrow \subseteq Q \times 2^{\Sigma} \times 2^{\Sigma} \times Q$ is a relation*

When $(q, U, V, q') \in \rightarrow$, we write $q \xrightarrow{U|V} q'$. Furthermore,

- *if $q \xrightarrow{U|V} q'$, then $U \subseteq V$ (*reactivity*)*
- *if $q \xrightarrow{U|V} q'$ and $U \subseteq V' \subseteq V$, then $q \xrightarrow{U|V'} q'$ (*uniformity*)*

For the sake of simplicity, we work with *normalized* Reo automata; every Reo automaton can be transformed into an equivalent normalized Reo automaton [6].

Each transition of a Reo automaton represents an interaction allowed by the Reo circuit, as well as the necessary change in (internal) state. In a transition $q \xrightarrow{U|V} q'$, the set U represents the *ports fired* in the interaction, while V represents the *ports available* at that point in time. Reactivity guarantees that only available ports are fired, while uniformity ensures that unfired but available ports becoming unavailable will not change the availability of the interaction, or the resulting state change [6].

A Reo automaton $M = (Q, \rightarrow, q_0)$ defines a language $L_M(q)$ for every state $q \in Q$ as follows: for all $q \in Q$, it holds that $\lambda \in L_M(q)$; furthermore, $(U, V)w \in L_M(q)$ if and only if there exists a $q' \in Q$ such that $q \xrightarrow{U|V} q'$ and $w \in L_M(q')$. We write L_M for $L_M(q_0)$.

To encode Reo automata coalgebraically, we switch the base category of the discussion in this section to **Posets**, the category of partially ordered sets and monotone functions. If X is a poset, we write $\leq_X$ for the accompanying partial order. We use 2 to denote the two-element poset $\{0, 1\}$, in which $0 \leq_2 1$. It is not hard to see that this category is *Cartesian closed*, that is, it comes with a terminal object (the singleton poset 1), products (the product poset $X \times Y$) and exponentials (the poset of monotone functions X^Y, ordered pointwise). We also have an analogue of the powerset functor: 2^X is the set of all monotone functions from X to 2, or, equivalently, the poset of all $\leq_X$-upclosed subsets of X, ordered by inclusion. We can verify that for a fixed poset A, the mappings $FX = A \times X$ and $FX = X^A$ are functorial, as is $FX = 2^X$; the actions of the former two on monotone functions can be lifted from **Sets**, while the latter sends $f : X \rightarrow Y$ to the function 2^f, mapping $\leq_X$-upclosed subsets of X to $\leq_Y$-upclosed subsets of Y:

$$2^f(U) = \{y \in Y : \exists x \in U.\ f(x) \leq y\}$$

We now have the ingredients to define a coalgebraic version of Reo automata, which takes the form of a deterministic automaton in **Posets** over a special alphabet, as follows.

Definition 4. *The* poset of interactions, *denoted $\mathbb{A}$, is $\{(U, V) : \emptyset \neq U \subseteq V \subseteq \Sigma\}$, ordered by the partial order $\leq_{\mathbb{A}}$, in which $(U, V) \leq_{\mathbb{A}} (U', V')$ if and only if $U = U'$ and $V' \subseteq V$.*

A Reo coalgebra *is a coalgebra for the functor $FX = 2 \times X^{\mathbb{A}}$.*

In what follows, we denote the elements (U, V) of $\mathbb{A}$ by writing $U|V$. We further abbreviate by denoting U and V as strings, i.e., when $U = \{\mathsf{A}, \mathsf{B}\}$ and $V = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$, we write $\mathsf{AB|ABC}$ for (U, V). Words over $\mathbb{A}$ are written as letters separated by semicolons, e.g., we write $\mathsf{A|A;B|B}$ for the word $(\{\mathsf{A}\}, \{\mathsf{A}\})(\{\mathsf{B}\}, \{\mathsf{B}\}) \in \mathbb{A}^*$.

Let us fix a Reo automaton $M = (Q, \rightarrow, q_0)$. We can represent $\rightarrow$ as a function $\delta : Q \rightarrow (2^Q)^{\mathbb{A}}$, by setting $\delta(q)(a) = \{q' \in Q : q \xrightarrow{a} q'\}$. If we equip Q with the discrete order, δ is monotone. Furthermore, for $q \in Q$, $\delta(q) : \mathbb{A} \rightarrow 2^Q$ is monotone as a consequence of uniformity. Now, δ gives rise to $\delta^\sharp : 2^Q \rightarrow (2^Q)^{\mathbb{A}}$, given by

$$\delta^\sharp(U)(a) = \{q' \in Q : \exists q \in U.\ q \xrightarrow{a} q'\}$$

We note that $\delta^\sharp$ is again monotone. In general, a monotone $\delta^\sharp : 2^Q \rightarrow (2^Q)^{\mathbb{A}}$ can be constructed from any monotone $\delta : Q \rightarrow (2^Q)^{\mathbb{A}}$, by recognizing that $2^{(-)}$ is a monad on **Posets**, and choosing for $\delta^\sharp$ the Kleisli extension of δ.

Our Reo automaton M now gives rise to a Reo coalgebra $(2^Q, <\epsilon^\sharp, \delta^\sharp>)$, where 2^Q and $\delta^\sharp$ are as above, and $\epsilon^\sharp : 2^Q \to 2$ is given by $\epsilon(U) = 1$ if and only if $U \neq \emptyset$.

We proceed to recover the language semantics of Reo automata from a Reo coalgebra, by finality. For this, we equip $\mathbb{A}^*$ with the pointwise extension of $\leq_{\mathbb{A}}$, i.e., $a_0 a_1 \cdots a_{n-1} \leq_{\mathbb{A}^*} a'_0 a'_1 \cdots a'_{m-1}$ if and only if $n = m$, and for $i \in \{0, 1, \ldots, n-1\}$ it holds that $a_i \leq_{\mathbb{A}} a'_i$. This is equivalent to defining $\mathbb{A}^* = \coprod_{i \in \mathbb{N}} \mathbb{A}^i$, using the coproducts and products of **Posets**. We know from the work of Arbib and Manes [3, Sect. 2.2] that $\mathbb{A}^*$ (resp. $2^{\mathbb{A}^*}$) defined as such provides the desired notion of reachability (resp. observability). Specifically, we have the exact same situation as in (5), but with $\mathbb{A}$ instead of A and with Q being a Reo coalgebra with an initial state in **Posets**. Reachability and observability maps are defined in complete analogy to their definition for a DFA.

The relation between the language semantics of a Reo automaton and its encoding into a Reo coalgebra can now be formulated as follows.

Lemma 7. *Let $M = (Q, \to, q^0)$ be a Reo automaton, and let $(2^Q, < \epsilon^\sharp, \delta^\sharp >)$ be the Reo coalgebra obtained from it. Furthermore, let h be the unique homomorphism into the final Reo coalgebra $(2^{\mathbb{A}^*}, < \lambda?, \partial >)$. If $U \in 2^Q$, then*

$$h(U) = \bigcup_{q \in U} L_M(q)$$

Proof. Let $w \in \mathbb{A}^*$; we proceed by induction on $|w|$. In the base, where $w = \lambda$, we have that $\lambda \in h(U)$ if and only if $U \neq \emptyset$, which holds precisely when $\lambda \in \bigcup_{q \in U} L_M(q)$.

For the inductive step, let $w = av$ for $a \in \mathbb{A}$ and $v \in \mathbb{A}^*$. In that case, $av \in h(U)$ if and only if $v \in h(d(U)(a))$; by induction, the latter holds if and only if $v \in L_M(q')$ for some $q' \in d(U)(a)$, i.e., $v \in L_M(q')$ for some $q \in U$ with $q \xrightarrow{a} q'$, which is equivalent to $av \in L_M(q)$, which in turn is equivalent to $av \in \bigcup_{q \in U} L_M(q)$.

To learn Reo coalgebras using the framework outlined earlier, we define an observation table in this setting. Consider finite subsets S and E of $\mathbb{A}^*$, with all ordering inherited from $\mathbb{A}^*$,[1] S prefix-closed, and E suffix-closed. Note that if $s, s' \in S$ and $e, e' \in E$ are such that $s \leq_S s'$ and $e \leq_E e'$, then $row(s)(e) \leq_2 row(s')(e')$ by monotonicity. Thus, if $row(s)(e) = 1$, we can immediately conclude $row(s')(e') = 1$ without having to query the latter; similarly, $row(s)(e) = 0$ whenever $row(s')(e') = 0$. A similar optimization applies to computing the function row_A. Note that the assumption of a prefix-closed language enables another optimization for the computation of these functions: if $s, s' \in S$ and $e, e' \in E$ are such that $s'e'$ is a prefix of se, then $row(s')(e') = 1$ whenever $row(s)(e) = 1$, and $row(s)(e) = 0$ whenever $row(s')(e') = 0$.

[1] What follows works also for discrete orders on S and E. Acknowledging the additional structure, however, allows us to explicitly save queries by exploiting the monotonicity of the row function.

Since **Posets** is locally finitely presentable, it admits strong epi-mono factorizations [1,10]. There is no difference between closedness and consistency for learning a regular language over the alphabet $\mathbb{A}$ and closedness and consistency in the present setting. Furthermore, like the notions of reachability and observability, the encodings of prefix-closedness and suffix-closedness as performed in Sect. 2.2 translate directly, as do Lemmas 5 and 6. Theorem 1 is therefore valid also in this setting.

It remains to show how we can obtain a Reo automaton from an observation table.

Definition 5. *For (S, E) closed and consistent, we define $M = (Q^+, \to, q_0)$ as follows*

$$Q^+ = \{row(s) : s \in S,\ row(s)(\lambda) = 1\} \qquad\qquad q_0 = row(\lambda)$$

$$row(s) \xrightarrow{a} row(t) \iff row(t) \leq_{2^E} row_A(sa)$$

The fact that M is well-defined is a consequence of closedness and consistency as before. Additionally, we remark that reactivity follows from the definition of $\mathbb{A}$, and uniformity is a consequence of the monotonicity of *row*. It remains to show that the translation above is faithful, i.e., that the languages of the states of this Reo automaton correspond to the interpretation of $(Q, {<}\mathit{final}, \delta{>})$ in the final Reo-coalgebra.

Lemma 8. *Let (S, E) be a closed and consistent observation table, obtained by learning the language of a Reo automaton, and let $M = (Q^+, \to, q_0)$ be the Reo automaton obtained from this table. Let h be the unique homomorphism from $(Q, {<}\mathit{final}, \delta{>})$ into the final Reo coalgebra. For $q \in Q^+$, we have that $L_M(q) = h(q)$.*

Proof. We start by proving that if $w \in h(q)$, then $\mathit{final}(q) = 1$; the proof proceeds by induction on $|w|$. In the base, where $w = \lambda$, we know that $1 = h(q)(\lambda) = \mathit{final}(q)$. For the inductive step, write $w = av$ and assume the claim holds for v. Since $v \in h(\delta(q)(a))$, we find that $\mathit{final}(\delta(q)(a)) = 1$ by induction. But then

$$1 = \mathit{final}(\delta(q)(a)) = row_A(sa)(\lambda) = \mathcal{L}(sa) \leq_2 \mathcal{L}(s) = row(s)(\lambda) = \mathit{final}(q)$$

in which $\mathcal{L}(sa) \leq_2 \mathcal{L}(s)$ follows from prefix-closure of $\mathcal{L}$. Consequently, $\mathit{final}(q) = 1$.

For the main claim, we should verify that for $w \in \mathbb{A}^*$ and $q \in Q^+$ it holds that $L_M(q)(w) = h(q)(w)$. Note that there exists an $s \in S$ such that $q = row(s)$. The proof again proceeds by induction on $|w|$. In the base, where $w = \lambda$, we have that $\lambda \in L_M(q)$, as well as $\lambda \in h(q)$, since $q \in Q^+$ and therefore $\mathit{final}(q) = row(s)(\lambda) = 1$.

For the inductive step, let $w = av$ for $a \in \mathbb{A}$ and $v \in \mathbb{A}^*$. On the one hand, if $w \in L_M(q)$, then there exists a $q' \in Q$ such that $q \xrightarrow{a}_d q'$ and $v \in L_M(q')$. By definition of $\to_d$, we find that $q' \in Q^+$ and $q' \leq_{2^E} row_A(sa)$. By induction and monotonicity:

$$v \in h(q') \subseteq h(row_A(sa)) = h(\delta(row(s))(a)) = \partial(h(row(s)))(a)$$

allowing us to conclude that $w = av \in h(row(s)) = h(q)$.

On the other hand, if $w \in h(q)$, then $v \in \partial(h(q))(a) = h(\delta(q)(a))$. By the first part of this proof, $\mathit{final}(\delta(q)(a)) = 1$, and so $\delta(q)(a) \in Q^+$. We then find by induction that $v \in L_M(d(q)(a))$. Since $q \xrightarrow{a}_d \delta(q)(a)$, we conclude that $w = av \in L_M(q)$.

3.1 Learning a Reo Circuit

We now review how the algorithm in Fig. 1 would work as applied to Reo automata. Imagine that the Learner receives as input a Teacher for the Reo circuit in Fig. 2, which represents what is called a "lossy FIFO" in Reo literature. The intended behavior of this circuit is as follows. When the buffer is empty, A can fire, and fill the buffer. If the buffer is full, three possibilities exist:

(i) the buffer is emptied by firing B, or
(ii) the buffer is emptied and immediately filled by firing A and B concurrently, or
(iii) A fires, but the input is discarded (and the first token remains in the buffer).

Important to note is that the last option should not be available when B is enabled; that is, data is only discarded when the buffer is full *and* the token in the buffer cannot be handed off through B. This distinction makes the circuit in Fig. 2 a prime example for learning a Reo automaton, since transitions carry information about ports fired and ports available [6]; if we compute the semantics of this circuit using Constraint Automata [4], this behavior cannot be modeled.

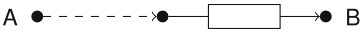


Fig. 2. The Reo circuit being learned

Before we dive into learning the circuit, let us first take a brief look at the poset of interactions we will be working with. Given that $\Sigma = \{\mathsf{A}, \mathsf{B}\}$, we can compute

$$\mathbb{A} = \{\mathsf{A}|\mathsf{A}, \mathsf{A}|\mathsf{AB}, \mathsf{B}|\mathsf{B}, \mathsf{B}|\mathsf{AB}, \mathsf{AB}|\mathsf{AB}\}$$

As for $\leq_{\mathbb{A}}$, there are only two (non-trivially) related pairs of letters; they are:

$$\mathsf{A}|\mathsf{AB} \leq_{\mathbb{A}} \mathsf{A}|\mathsf{A} \qquad\qquad \mathsf{B}|\mathsf{AB} \leq_{\mathbb{A}} \mathsf{B}|\mathsf{B}$$

The algorithm starts off by building a table for $S = \{\lambda\}$ and $E = \{\lambda\}$. Here, a membership query of $U_1|V_1;U_2|V_2;\cdots;U_n|V_n$ should be interpreted as "can I fire these ports, while these ports are available, in this order?". In this case, the entry for $row_A(\mathsf{A}|\mathsf{A})(\lambda)$ is 1 because, in the initial configuration, we can fire A if it is available, while the entry for $row_A(\mathsf{B}|\mathsf{B})(\lambda)$ is 0 because B cannot be fired in the initial configuration even if it is available.

Step 1

	λ
λ	1
A|A	1
A|AB	1
B|B	0
B|AB	0
AB|AB	0

(S, E) consistent? ✓ (S, E) closed?
No, $\mathit{row}_A(\mathsf{B|B}) \neq \mathit{row}(\lambda)$

Then, $S \leftarrow S \cup \{\mathsf{B|B}\}$.
We continue with **Step 2**.

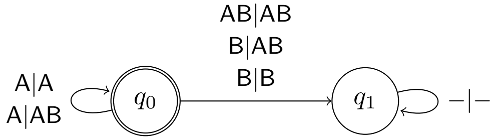
At this point, we note that we have made a membership query that could in principle have been skipped: the fact that $\mathit{row}_A(\mathsf{A|A})(\lambda) = 1$ follows from the fact that $\mathit{row}_A(\mathsf{A|AB})(\lambda) = 1$, since $\mathsf{A|AB} \leq_{\mathbb{A}} \mathsf{A|A}$, and row_A is monotone.

To make the table closed, we add B|B to S, and end up with the following table. Here, the row label "B|B;−" represents all rows labeled B|B;a for $a \in \mathbb{A}$; the value for entries in these rows is always 0, by prefix closure of the target language.

Step 2

	λ
λ	1
B|B	0
A|A	1
A|AB	1
B|AB	0
AB|AB	0
B|B;−	0

(S, E) consistent? ✓ (S, E) closed? ✓
Then, we guess the automaton



Teacher replies with counter-example: A|A;A|AB

Then, $S \leftarrow S \cup \{\lambda, \mathsf{A|A}, \mathsf{A|A;A|AB}\}$.
We continue with **Step 3**.

The counterexample given by the Teacher here tells us that, after firing A|A, our circuit ends up in a different state, since firing A|AB should not be possible (i.e., we should not be able to fire A but not B while both are available). Our current automaton does not account for this possibility: A|AB can be fired after A|A, and this brings us to an accepting state; the counterexample is therefore justified.

Extending the table with the new contents of S, we find the following

Step 3

	λ
λ	1
A|A	1
B|B	0
A|A;A|AB	0
A|AB	1
B|AB	0
AB|AB	0
B|B;–	0
A|A;A|A	1
A|A;B|B	1
A|A;B|AB	1
A|A;AB|AB	1
A|A;A|AB;–	0

(S, E) consistent?
No, because $\mathit{row}(\lambda) = \mathit{row}(\mathsf{A|A})$,
while $\mathit{row}_A(\mathsf{A|AB}) \neq \mathit{row}_A(\mathsf{A|A;A|AB})$.

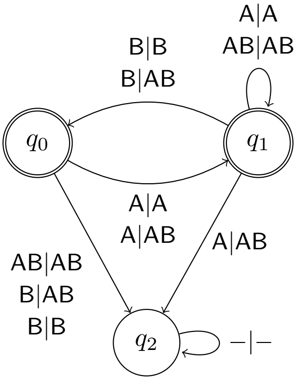
Then, $E \leftarrow \{\mathsf{A|AB}\}$.
We continue with **Step 4**.

Filling in the table with the updated E, we arrive at the following.

Step 4

	λ	A|AB
λ	1	1
A|A	1	0
B|B	0	0
A|A;A|AB	0	0
A|AB	1	0
B|AB	0	0
AB|AB	0	0
B|B;–	0	0
A|A;A|A	1	0
A|A;B|B	1	1
A|A;B|AB	1	1
A|A;AB|AB	1	0
A|A;A|AB;–	0	0

(S, E) consistent? ✓ (S, E) closed? ✓
Then, we guess the automaton



The Teacher replies **yes**.

In the last step, the algorithm stops, as the conjectured automaton faithfully represents the description of the lossy FIFO given at the start of this example. In particular, q_1 models a circuit with a full buffer: the edge towards q_0 represents firing B (emptying the buffer), while the loop back to q_1 represents the possibility of firing A (discarding an incoming token) or firing both A and B (shifting the incoming token into the emptied buffer); lastly, the edge towards q_2 encodes that A cannot be fired when B is also available.

4 Discussion

We have presented a categorical reformulation of Angluin's learning algorithm, originally defined for deterministic finite automata. The categorical reformulation enables us to explore two avenues of generalization: varying the functor (giving for instance different input/output for the automaton) and varying the category under study (changing for instance the type of computations involved). In a previous paper [7] we explored the former avenue and derived algorithms for Moore and Mealy machines, which generalize the output set of DFAs. The application we concretely considered in the present paper explored the latter avenue, yielding an algorithm for Reo automata—essentially, finite automata in the category **Posets**. What makes the change in category interesting is that it precisely captures the algebraic structure of the actions (signal flows) in Reo and highlights the fact that interaction is a first-class concept by defining a poset of interactions to be used as the alphabet for the Reo automaton.

We would like to explore linking this algorithm to other work of Farhad on compilation [8] to learn Reo patterns. The work in this paper enables us to learn a Reo automaton of the global connector. However, in order to learn the components of the connector we would have to compile the *global* automaton into the composition of smaller Reo automata that would in turn correspond to basic Reo connectors. For scalability, it would be ideal to integrate part of the splitting into the learning algorithm.

References

1. Adámek, J., Rosický, J.: Locally Presentable and Accessible Categories. Cambridge University Press, Cambridge (1994)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
3. Arbib, M.A., Manes, E.G.: Adjoint machines, state-behavior machines, and duality. J. Pure Appl. Algebra **6**(3), 313–344 (1975)
4. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in reo by constraint automata. Sci. Comput. Program. **61**(2), 75–113 (2006). https://doi.org/10.1016/j.scico.2005.10.008
5. Barr, M., Wells, C.: Toposes, Triples and Theories. Springer, Berlin (1985). Revised and corrected version available from www.cwru.edu/artsci/math/wells/pub/ttt.html
6. Bonsangue, M., Clarke, D., Silva, A.: Automata for context-dependent connectors. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 184–203. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02053-7_10
7. Jacobs, B., Silva, A.: Automata learning: a categorical perspective. In: van Breugel, F., Kashefi, E., Palamidessi, C., Rutten, J. (eds.) Horizons of the Mind. A Tribute to Prakash Panangaden. LNCS, vol. 8464, pp. 384–406. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06880-0_20
8. Jongmans, S.T.Q., Arbab, F.: Global consensus through local synchronization: a formal basis for partially-distributed coordination. Sci. Comput. Program. **115–116**, 199–224 (2016)
9. Kalman, R.: On the general theory of control systems. IRE Trans. Autom. Control **4**(3), 110 (1959)

10. Milius, S.: A sound and complete calculus for finite stream circuits. In: Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, Edinburgh, United Kingdom, 11–14 July 2010, pp. 421–430 (2010). https://doi.org/10.1109/LICS.2010.11
11. Vaandrager, F.W.: Model learning. Commun. ACM **60**(2), 86–95 (2017). https://doi.org/10.1145/2967606

Reo Connectors and Components as Tagged Signal Models

Marjan Sirjani[1,2], Fatemeh Ghassemi[3(✉)], and Bahman Pourvatan[2,4]

[1] School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden
marjan.sirjani@mdh.se
[2] School of Computer Science, Reykjavik University, Reykjavik, Iceland
bahman@ru.is
[3] School of Electrical and Computer Engineering, University of Tehran, Tehran, Iran
fghassemi@ut.ac.ir
[4] LIACS, Leiden University, Leiden, The Netherlands

Abstract. Tagged Signal Model (TSM) is a denotational framework and a meta-model to study certain properties of models of computation. To study the behavior of Reo connectors in a closed system, we propose two denotational semantics for Reo using TSM. TSM is very similar to the coalgebraic model of Timed Data Streams (TDS), the first formal semantics and the basis for most of the other formal semantics of Reo. There is a direct mapping between the *time* – *data* pairs of TDS, and *tag* – *value* of TSM. This work shows how treating tags to be either totally or partially ordered has a direct consequence on the results. We looked into five primitive connectors of Reo in both these settings and discuss the determinacy of systems.

Foreword

Tagged Signal Model (TSM) and Timed Data Streams (TDS) are very similar mathematical models. We observed the extreme similarity and started this research with the goal to use the rich set of theorems and techniques built around the TSM framework for reasoning about different properties of Reo circuits. But the path we went through was not at all what we have expected. We have aimed for a discussion on determinacy of Reo connectors. We considered the five primitive Reo connectors as processes in TSM. We close the model, by adding source and sink processes to produce input for, and consume the output of each connector. The determinacy of such compositions as closed models depend on the source and sink processes. Moreover, we could not find any theorem on determinacy of models for the Rendezvous model of computation.

As we know Farhad's love of Persian poetry we would like to cite the first Ghazal of Hafez here:

Ho! O Saki, pass around and offer the bowl:
For love appeared easy at first, but hardships have occurred.

We hope that Hafez forgives us for making an analogy between love and research.

F. de Boer et al. (Eds.): Arbab Festschrift, LNCS 10865, pp. 160–173, 2018.
https://doi.org/10.1007/978-3-319-90089-6_11

1 Introduction

Development of concurrent systems has many challenges due to the well-known problems such as race conditions, synchronization of events, etc. Component-based development paradigm brings up a revolution in the software development. A system is made by composing off-the-self previously developed components. The glue code, by composing the components together, plays the important role of orchestration and defines how the components are coordinated to remedy the concurrency problems. Coordination languages have emerged for building the interaction protocols among the components in a system independent of the behavior of components. Reo was introduced as an exogenous coordination language to specify the glue code in a compositional way [1]. By composing ready-to-use components and Reo connectors, a system can be constructed. One of the challenges in the composition of Reo connectors is the interpretation of feedback loops. Feedback is a useful control mechanism, present in many coordination patterns, which may lead to non-deterministic behavior that it is not appealing.

Several behavioral semantics based on different formal classes have been appeared for Reo, namely, based on coalgebraic models, operational models [3], and graph-coloring [4]. The coalgebraic model of *timed data streams* (TDS) was the first model used to give semantics to Reo connectors [2]. It defines which and when data items flow through each node of a connector. To provide tools to support implementations or analysis of Reo connectors with formal techniques other semantics were developed. The TDS semantics define the behavior of connectors independent of interacting components (processes). In an ideal environment, there is an assumption that all the source and sink components (processes) are always willing to generate and consume data (and willing to Rendezvous). However such an assumption is not valid for all off-the-shelf components and hence, their composition may lead to unexpected behaviors. Here, we study the behavior of connectors and processes together as a composition.

The intuitive way of thinking about flows through a Reo connector resembles the way the behavior of processes is defined by *tagged signal models* (TSM) [6]. The denotational formalism of tagged signal model is a meta-model to study certain properties of models of computation in a unified framework. In this framework, a system is modeled by the composition of a set of processes. Each process is defined as a relation/function between signals which can be partitioned into input/output signals. Composition is treated as combining the output signals of one process to the inputs of some processes. Each signal is a set of tag-value pairs. By restricting processes to functions in TSM, feedback makes such systems self-referential. Mathematically, the notion of self-reference is tackled as a fixed point problem, as illustrated by the simple system in Fig. 1 in which the input and output signals are the same due to the feedback connection. Such a system has a well-defined behavior if F has a fixed point. This imposes constraints on the functions that are used to model systems. There are a set of well-defined theorems in the TSM framework to show when a model with feedback has behavior.

TSM defines precisely processes, signals, and events, and gives a framework for identifying the essential properties of discrete-event systems, dataflow,

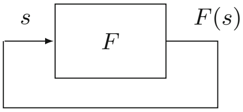


Fig. 1. A functional process in a feedback

rendezvous-based systems, Petri nets, and process networks. TSM is a meta-model, and TDS can be defined as a specific model, based on TSM, where TDS timestamps are a totally ordered set of tags in TSM. Furthermore, TSM provides sufficient means to describe connectors and interacting processes in a unified way as intended by Reo. In our approach, the effects of nodes are explicitly modeled by constraints in TSM. We use TSM framework to reason about the determinacy of a system composed of Reo connectors (and components). We will show that in the TSM for Reo, we can use the same totally ordered tag system as in the TDS, but most Reo circuits will be nondeterministic. We will also show the first steps towards a TSM for Reo with partially ordered set of tags, and how this model may be closer to the Rendezvous model of Reo.

In Sect. 2, we explain Reo and its Timed Data Stream semantics. We also explain the Tagged Signal Model, the possible structures of tag systems, and the determinacy of a system in this framework. In Sect. 3, we briefly point out to different semantics of Reo, and also to different places that TSM is used as the semantic framework for different models of computations. In Sect. 4, we show how to model Reo in the TSM framework with two different tag systems. Section 5 includes discussions and conclusions.

2 Preliminary Concepts

We first provide an overview of the syntax and semantics of the coordination language Reo, and then the meta-model of tagged signal framework.

2.1 Reo

Reo is a model for building component connectors in a compositional manner [1]. Each connector in Reo is, in turn, constructed compositionally out of simpler connectors, which are ultimately composed out of primitive channels.

A channel is a primitive communication medium with exactly two ends, each with its own unique identity. There are two types of channel ends: *source end* through which data enter and *sink end* through which data leave a channel. A channel must support a certain set of primitive operations, such as I/O, on its ends; beyond that, Reo places no restriction on the behavior of a channel.

A set of primitive Reo channels (together with their Timed Data Stream semantics) are shown in Table 1. Channels are connected to make a circuit. Connecting (or *joining*) channels is putting channel ends together in *nodes*. Thus, a set of channel ends is associated with a *node*. The semantics of a node depends on its type. Based on the types of its coincident channel ends, a node can have one of three types. If all channel ends coincident on a node are exclusively source (or sink) channel ends, the node is called a source (respectively, sink) node. Otherwise, it is called a mixed node.

A component can write data items to a source node that it is connected to. The write operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a *replicator*. A component can obtain data items, by an input operation, from a sink node that it is connected to. A take operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected nondeterministically. A sink node, thus, acts as a nondeterministic *merger*. A mixed node nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends.

Reo offers an open ended set of channels, but a set of primitive channels, shown in Fig. 1, are commonly used in Reo circuits. The behavior of every connector in Reo imposes a specific coordination pattern on the entities that perform normal (blocking) I/O operations through that connector, which itself is oblivious of those entities. This makes Reo a powerful *glue language* for compositional construction of connectors to combine component instances and Web services into a software system and exogenously orchestrate their mutual interactions.

2.2 Timed Data Stream

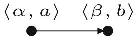
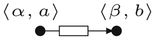
In [2], Arbab and Rutten introduce Timed Data Stream (TDS) models as the first formalization of the semantics of Reo connectors. Informally, a TDS model of a connector describes for each of its nodes which and when–in dense time–data items flow through this node. It does so by associating each node with a timed data stream. A timed data stream (α, a) consists of a *data stream* $\alpha \in Data^{\omega}$ and a monotonically increasing *time stream* $a \in R^{\omega}_{\geq}$ consisting of increasing positive real numbers including zero. The time stream a indicates for each data item α_n the moment a_n at which it is being input or output. By associating each node of a connector with its own TDS in a TDS tuple, a single execution of the connector is defined. To describe all possible executions of a connector, it has to be associated with a set of TDS tuples; we call such a set the TDS model of the connector. Usually, such a TDS model is defined as a predicate on TDSs that induces the set of admissible TDS tuples of a connector. (Enumerating all admissible TDS tuples of a connector becomes impossible because, not only each stream in a TDS itself is infinite, the set of admissible TDS tuples usually contains infinitely many elements.)

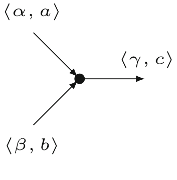
The Sync channel inputs the data elements in the stream α at time a, and outputs the date stream β at time b. All data elements that come in, come out in the same order, i.e., $\alpha = \beta$, and at the exact same time $a = b$. For a FIFO1 channel what comes in, comes out $(\alpha = \beta)$, but at a later time $(a < b)$.

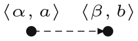
Moreover, at any moment the next data item can only be input after the present data item has been output ($b < a'$) which means $b(n) < a(n+1)$, for all $n \geq 0$. The connector Merger is a ternary relation with two input ends and one output end. This connector merges the two input data streams into a data stream on its end. Merger handles the data element on one of its input ends first which comes out on its output end. The LossySync channel passes a data element on its input end instantaneously on as an output element, and continues with the remainder of the streams as before. If the output end is not ready for the rendezvous then the input element is discarded. The SyncDrain channel has two input ends, and the data elements in the stream α and β enter the two input ends of this channel simultaneously, i.e., $a = b$. There is no constraint on the data streams, the data elements enter and disappear.

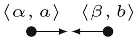
Constraint automata can be viewed as acceptors for tuples of timed data streams that are observed at certain input/output ports $A_1, \ldots, A_n$ of components. The rough idea is that such an automaton observes the data occurring at $A_1, \ldots, A_n$ and either changes its state according to the observed data or rejects it if there is no corresponding transition in the automaton.

Table 1. Primitive connectors and their corresponding semantics as Timed Data Stream

Reo Connectors	Timed Data Stream
Synchronous Channel (Sync) 	$\langle \alpha, a\rangle \longmapsto \langle \beta, b\rangle \equiv \alpha = \beta \wedge a = b$
FIFO with one buffer (FIFO1) 	$\langle \alpha, a\rangle \mathrel{-\!\circ\!\!\rightarrow} \langle \beta, b\rangle \equiv \alpha = \beta \wedge a < b < a'$

Merger Connector
(Merger)

	$M(\langle \alpha, a\rangle, \langle \beta, b\rangle, \langle \gamma, c\rangle) \equiv a(0) \neq b(0) \wedge$
	if $a(0) < b(0)$ $\alpha(0) = \gamma(0) \wedge a(0) = c(0) \wedge M(\langle \alpha', a'\rangle, \langle \beta, b\rangle, \langle \gamma', c'\rangle)$
	if $b(0) < a(0)$ $\beta(0) = \gamma(0) \wedge b(0) = c(0) \wedge M(\langle \alpha, a\rangle, \langle \beta', b'\rangle, \langle \gamma', c'\rangle)$

Lossy Synchrounous Channel
(LossySync)

	$\langle \alpha, a\rangle \dashrightarrow \langle \beta, b\rangle \equiv a(0) \leq b(0) \wedge$
	if $a(0) = b(0)$ $\alpha(0) = \beta(0) \wedge \langle \alpha', a'\rangle \dashrightarrow \langle \beta', b'\rangle$
	if $a(0) < b(0)$ $\langle \alpha', a'\rangle \dashrightarrow \langle \beta, b\rangle$

Synchronous Drain (SyncDrain) 	$\langle \alpha, a\rangle \longrightarrow\!\!\!\longleftarrow \langle \beta, b\rangle \equiv a = b$

2.3 Tagged Signal Model

Let T and V denote the set of tags and values, ranged over by t and v respectively. An event e is a pair of a tag and a value, where its tag may denote the time that the event has occurred while its value may represent the operand/result of a computation. A signal s is a set of events, and the set of all signals S is defined by the powerset $\wp(T \times V)$. A tuple $\mathbf{s}$ of N signals written by $\mathbf{s} = (s_1, \ldots, s_N)$, is used to model the behavior of a process; and a process which is a unit of computation is a set of behaviors. We use the position i in the tuple to denote the signal s_i. The set of all such tuples is denoted by S^N. The empty signal is denoted by $\lambda \in S$ while the tuple of empty signals by $\Lambda \in S^N$.

In this framework, a system is modeled by a composition of a set of processes. A process P is described in its general term as a subset of S^N, called its *sort*. A particular $\mathbf{s} \in S^N$ is called a *behavior* of P if $\mathbf{s} \in P$. For $N \geq 2$, a process can also be interpreted as a relation/function between the N signals in $\mathbf{s}$ which can be partitioned into input/output signals.

A composition of processes is simply defined by the intersection of the behaviors of the processes. In order to be able to compose processes, all processes have to be modeled using the same sort. This is one of the more subtle aspects of TSM. This means that the set of behaviors of a process includes signals that are neither inputs nor outputs to the process, that the process has nothing to do with. Every possible valuation of such signals can be found in the behaviors of the process. When composing processes by intersection, a process has no effect on signals that are irrelevant to it because all possible valuations of those signals are legitimate behaviors of the process, and when we form set intersection, these irrelevant signals are not constrained by the process in any way. So, for composition, processes are defined as a subset of the same sort by augmenting their tuples using cross product.

Interaction is defined by the particularly simple process $C \subset S^N$, called *connection*, where two (or more) of the signals in the N-tuple are constrained to be identical. Connections are useful to couple the behaviors of other processes. For example, the connection $C_{i,j} = \{s \in S^N \mid s_i = s_j\}$, intuitively models that the signals s_i and s_j are connected together. To hide some signals, the projection operator $\pi_I(\mathbf{s})$ is used which maps $s = (s_1, \ldots, s_N)$ to $(s_{i_1}, \ldots, s_{i_m})$ where $I = \{i_1, \ldots, i_m\}$ is an ordered set of indexes in the range $1 \leq i \leq N$. For instance the composite process in the right-side of Fig. 3 is defined by $\pi_{1,4}((P_1 \times S^2) \cap (S^2 \times P_2) \cap C_{2,3})$ which can be simply denoted by $\pi_{1,4}((P_1 \times P_2) \cap C_{2,3})$.

A process P is functional with respect to the index sets I and O for m and n input and output signals respectively, if for every $\mathbf{s} \in P$ and $\mathbf{s}' \in P$, $\pi_I(\mathbf{s}) = \pi_I(\mathbf{s}')$ implies $\pi_O(\mathbf{s}) = \pi_O(\mathbf{s}')$. Therefore, for the functional process P with respect to (I, O), a single-valued mapping $F : S^m \rightarrow S^n$ can be defined such that for all $\mathbf{s} \in P$, $\pi_O(\mathbf{s}) = F(\pi_I(\mathbf{s}))$. Note that a process may be functional with respect to more than one pair of index set (I, O). This property is preserved by the composition of functional processes as long as no feedback loop is involved.

Tag Systems. Intuitively tags are used to model time, precedence relations, synchronizations points, etc. The central role of a tag system is to establish ordering among events. The structure of a tag system distinguishes various concurrent models of computation, classified into *timed* and *untimed.* The former characterizes systems in which the order of events is deterministically defined relative to some physical or logical clock. Therefore, timed models of computation are characterized by a totally ordered set of tags while untimed ones by a partially ordered set of tags. In a timed model, the order of all events is clear and all tags are comparable, while in an untimed model, a subset of events can be ordered.

Determinacy. Many processes (not necessarily functional) have the notion of inputs which characterize events or signals that are defined outside the process. Formally, an input to the process $P \subseteq S^N$ is an externally imposed constraint $A \subseteq S^N$ such that $A \cap P$ is the total set of acceptable behaviors. The behavior of a process for a set of possible inputs, denoted by $B \subseteq \wp(S^N)$, can be defined by (P, B).

A process (P, B) is called *closed* if $B = \{S^N\}$, a set with only one element, $A = S^N$. Since $A \cap P = P$, no input constraints are imposed on a closed process. A process and its possible inputs is *open* if it is not closed.

A process is called deterministic "if for any input $A \in B$ it has exactly one behavior or exactly no behavior; i.e. $|A \cap P| = 1$ or $|A \cap P| = 0$, where $|X|$ is the size of the set X." Otherwise, it is called nondeterministic. Consequently, a closed process P is deterministic if $|P| = 1$ or $|P| = 0$ (because $B = \{S^N\}$ and $|S^N \cap P| = 1$ or $|S^N \cap P| = 0$).

A functional process with respect to (I, O) is obviously deterministic if I and O together contain all the indexes in $1 \leq i \leq N$.

3 Related Work

In recent years, many formalisms for describing the behavior of Reo connectors have emerged. Jongmans and Arbab provided an overview of thirty different semantic formalisms for Reo in [4]. These models include coalgebraic models, operational models, and models based on graph-coloring. In [4], the authors also investigate in more detail the expressiveness of constraint automata and coloring models. We encourage the interested reader to [4] for more detailed information.

The first formal semantics proposed for Reo is Timed Data Streams proposed in [2]. In [2], like in most other semantics proposed for Reo, the focus is on the set of connectors and their composition, and the behavior of components is abstracted away and substituted by (sometimes implicit) assumptions. Tagged Signal Model is introduced in [6] as a denotational framework for comparing models of computation. It is a generalization of the Signals and Systems approach to system modeling and specification [7]. Using Tagged Signal Model, one can give structure to the sets of signals, give structure to the functional processes, and develop static analysis techniques. Moreover, we can compare certain properties of the models of computation, such as their notion of synchrony, and define formal relations among signals and process behaviors.

The similarities between tagged signals and timed data streams motivated us to look into Reo semantics using TSM framework. The goal is to use the established theory around Tagged Signal Model for reasoning about different properties of Reo circuits. This paper is the first attempt in moving towards this goal. Although TSM and TDS are very similar in their structure, our work in this paper shows subtle problems that need to be solved to be able to use the fixed point theorems established for TSM in the context of Reo connectors. TSM is mainly used to reason about Kahn Process Networks and Discrete Event model of computation. Reo is using a Rendezvous model of computation, and there is no TSM proposed for such models. Comparing to the TDS model, the TSM framework will add the ability to also model processes rather than just connectors. We will show that in the TSM for Reo, we can use the same totally ordered tag system as in the TDS, but most Reo circuits will be nondeterministic. We will also show the first steps towards a TSM for Reo with partially ordered set of tags, and how this model may be closer to the Rendezvous model of Reo.

4 Reo Connectors as Tagged Signal Models

To study the notions of concurrency, determinacy and synchronization of Reo connectors, we define how these properties can be captured within the tagged signal model. We provide a denotational semantics for Reo depending on how the tag system is structured.

In the first setting, the tag system is considered to be totally ordered and is the set of non-negative real numbers. This structure is inspired from the semantics of Reo as Timed Data Streams in [2]. As there is a global view of time among the components, it can be considered that all components and connectors are local. This setting can be extended to provide a suitable model for a discrete-event simulator of Reo connectors.

In the second setting, the tag system is considered to be partially ordered. In this setting, the components interacting with connectors and the nodes of the connectors are considered to be distributed, and hence such a tag system reflects the inherent problem in having a consistent time view in the implementation of distributed systems. In this setting, we may receive signals with incomparable tags at each channel end of a primitive connector (primitive channels or the merger). So, keeping the Reo node behavior similar to the first setting, our proposed TSM in the second setting comes short in defining semantics for all the primitive connectors.

The model of computation in Reo nodes is rendezvous, and Reo nodes take care of signals received from different (sink) channel ends with (possibly) incomparable partially ordered tags and dispatch them accordingly to the (source) channel ends. This way, we assume comparable tags for input and output signals of each primitive connector (similar to [2]). To be able to capture the semantics of Reo properly, in our future work we intend to adopt a partially ordered tag system, keep the semantics of each primitive connector aligned with their TDS definitions in [2], and add a more elaborated semantics for Reo nodes as special TSM connectors which compose channel ends and enforce the rendezvous model. Such an adoption may make it possible to use the well-established theorems around Network Process [5] to reason about the determinacy systems composed of Reo connector with feedback loops. Establishing all the details of this third model is left as our future work, and in this paper we show how we moved towards this semantics.

In all the models, each Reo channel can be viewed as a process, and its input/output streams of data in to and out of channel ends are viewed as signals.

4.1 A Totally Ordered Tag Model

When the tag set is totally ordered, for any two distinct tags $t, t' \in T$, either $t < t'$ or $t' < t$. We consider T to be the set of non-negative real numbers. We say $e_1 < e_2$ when $e_1 = (t_1, v)$ and $e_2 = (t_2, v)$ for some $v \in V$ where $t_1 < t_2$. Let $T(e)$ denote the tag of the event e, and let $T(s)$ denotes the set of tags of all events of the signal s.

We express the semantics of the five basic connectors as shown in Table 2. We assume that signals s_1 and s_2 are totally ordered:

$$s_1 = \{e_i, i \in \mathbb{N}\}, \forall\, i, j \cdot i < j \Rightarrow T(e_i) < T(e_j)$$
$$s_2 = \{e'_i, i \in \mathbb{N}\}, \forall\, i, j \cdot i < j \Rightarrow T(e'_i) < T(e'_j)$$

In this setting, the process of Sync is functional which can be defined by the identity function. However, the FIFO1 connector is not a functional process. The asynchronous behavior of FIFO1 makes it non-deterministic as for any $(t, v) \in s_1$, it can be delivered to s_2 at any time $t' > t$.

Here, Merger can be seen as a partial function process. The constraint $T(s_1) \cap T(s_2) = \emptyset$ expresses that its behavior is not defined when the tags of two events at its input signals are equal. The semantics of LossySync is defined by a relation due to its non-deterministic behavior in loosing events. The process of SyncDrain is defined as a partial function.

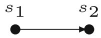
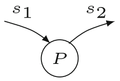
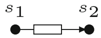
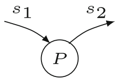
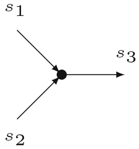
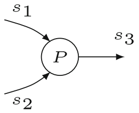
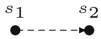
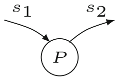
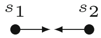
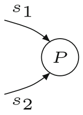
Two connectors can be composed through a node which contains the channel ends of the both connectors. Such a node imposes constraints on the signals representing the channel ends being contained by the node. For instance, the behavior of the mixed node in Fig. 2 is defined by the constraint $C'_{1,\{2,\ldots,n\}}$ which faithfully models the effect of nodes in Reo:

$$C'_{1,\{2,\ldots,n\}} = \{(s_1, \ldots, s_n) \mid \forall 2 < j \leq n \cdot (s_2 = s_j) \wedge \forall e_i \in s_1, e'_i \in s_2 \cdot (T(e_i) \leq T(e'_i))\}.$$

The semantics of a system is derived by the intersection of the behaviors of it's constituent processes.

Example 1. Consider a connector composed of two Sync channels connected in sequence, as shown in Fig. 3. The behavior of the composite connector is defined

Table 2. The primitive connectors Sync, FIFO1, Merger, LossySync, and SyncDrain and their corresponding semantics with a totally ordered tag model. Note that $s_1 = \{e_i, i \in \mathbb{N}\}$ where $\forall\, i, j \cdot i < j \Rightarrow T(e_i) < T(e_j)$, and $s_2 = \{e'_i, i \in \mathbb{N}\}$ where $\forall\, i, j \cdot i < j \Rightarrow T(e'_i) < T(e'_j)$, and we say $e_1 < e_2$ when $e_1 = (t_1, v)$ and $e_2 = (t_2, v)$ for some $v \in V$ where $t_1 < t_2$.

Reo Connector	Process	Behavior
		$P = \{(s_1, s_2) \mid s_1 = s_2\}$
		$P = \{(s_1, s_2) \mid \forall k \in \mathbb{N} \cdot (e_k < e'_k < e_{k+1})\}$
		$P = \{(s_1, s_2, s_3) \mid s_3 = s_1 \cup s_2 \wedge T(s_1) \cap T(s_2) = \emptyset\}$
		$P = \{(s_1, s_2) \mid s_2 \subseteq s_1\}$
		$P = \{(s_1, s_2) \mid T(s_1) = T(s_2)\}$

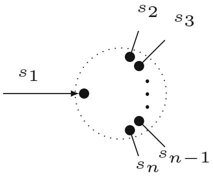


Fig. 2. A mixed node in Reo, including one source channel end (related to s_1), and multiple sink channel ends (related to s_2 to s_n)

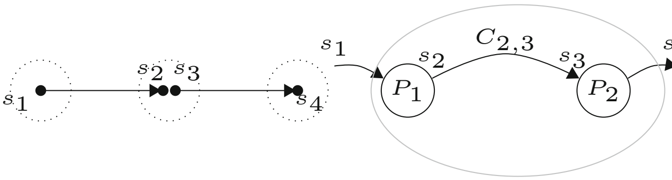


Fig. 3. Two Sync channel in sequence and its corresponding composite process

by the intersection of two functional processes P_1 and P_2 and the connection $C_{2,3}$. Therefore, Their composition is functional, and hence, since their composition is acyclic, it is deterministic. This can be validated by computing the behavior of the connector:

$$\pi_{1,4}((P_1 \times P_2) \cap C_{2,3}) = \{(s_1, s_4) \mid s_1 = s_4\}.$$

Example 2. Consider a system composed of a Merger connector and three components C_1, C_2, and C_3, as shown in Fig. 4. Assume that C_1 and C_2 are always willing to generate data while C_3 is always willing to consume data, defined by processes P_1, P_2, and P_3, respectively:

$$P_i = \{(s_1, s_2, s_3, s_4, s_5, s_6) \mid T(s_i) = R^{\omega}_{\geq}\},\ i \in \{1, 2, 3\}.$$

The behavior of the composite closed system is defined by the intersection of all processes, the corresponding process of Merger, modeled by M and the constraints $C'_{1,\{4\}}$, $C'_{2,\{5\}}$, and $C'_{6,\{3\}}$. The behavior of the composite system is:

$$P_1 \cap P_2 \cap P_3 \cap M \cap C'_{1,\{4\}} \cap C'_{2,\{5\}} \cap C'_{6,\{3\}}$$

where M is the augmented behavior of Merger:

$$M = \{(s_1, s_2, s_3, s_4, s_5, s_6) \mid s_6 = s_4 \cup s_5 \wedge T(s_4) \cap T(s_5) = \emptyset\}.$$

The composition yields an uncountably infinite set and hence, defines a nondeterministic system regarding the definition of determinacy in Sect. 2.3. Replacing C_1 and C_2 by components with only one behavior will make the system deterministic.

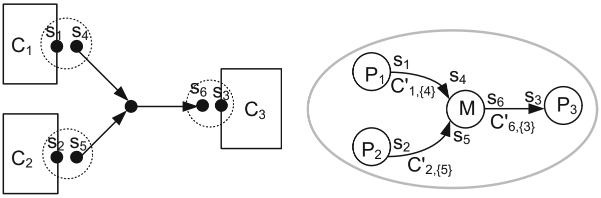


Fig. 4. A closed system composed of a Merger

Example 3. We change the system of Example 2 by connecting the inputs of the merger to a SyncDrain channel, as shown in Fig. 5. We assume again that C_1(similarly C_2) and C_3 are always willing to generate and consume data respectively. The behavior of the composite system is:

$$P_1 \cap P_2 \cap P_3 \cap D \cap M \cap C'_{1,\{4,7\}} \cap C'_{2,\{5,8\}} \cap C'_{6,\{3\}}$$

where M and D are the augmented behaviors of the Merger and SyncDrain channels, respectively:

$$M = \{(s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8) \mid s_6 = s_4 \cup s_5 \wedge T(s_4) \cap T(s_5) = \emptyset\}$$
$$D = \{(s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8) \mid T(s_7) = T(s_8)\}.$$

The composition yields an empty set and hence, results in a deterministic system.

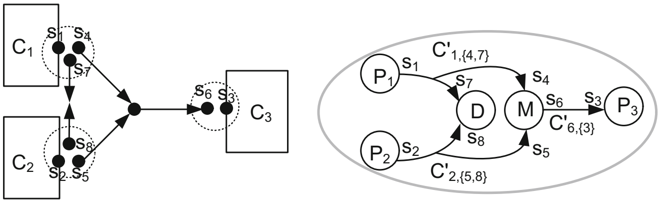


Fig. 5. A closed system composed of a Merger and a SyncDrain

4.2 A Partially Ordered Tag Model

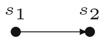
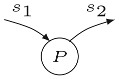
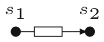
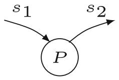
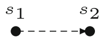
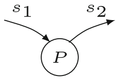
As explained before, a partially ordered tag model seems more suitable in a distributed setting for Reo connectors. Since different components have no consistent view of time, only events generated by the same primitive connector (component) at its sink channel ends (output ports of the component), and events coming into the primitive connector from the same source channel end (each input port of the component) are totally-ordered. When two sequences of events (generated by different channels/components) arrive at the channel ends of a Merger or SyncDrain, their tags cannot be compared. Therefore, the behavior of Merger and SyncDrain cannot be defined in this setting without further elaborations.

The semantics of Sync, FIFO1, and LossySync channels, provided in Table 3, are similar to the ones given in the timed model of computation (totally ordered tags). The Sync channel passes the events without manipulating their tags and values. The behavior of FIFO1 ensures that the tag of each outgoing event is greater then its corresponding incoming event but less then the next incoming event. The behavior of LossySync channel shows that if an event passes through the channel, its tag and value will not be changed, but passing an event is not guaranteed. In contrast to the Sync channel, due to its nondeterministic behavior in losing an event its process is not functional.

The behavior of a node is exactly modeled as in the totally ordered tags setting. It should be noted that we assumed that a node and its constituent ends are located at the same place, and hence their tags are comparable.

Providing a model for Merger, and SyncDrain channel needs more elaboration, and we leave it to our future work.

Table 3. The primitive connectors Sync, FIFO1, and LossySync channels and their corresponding semantics with a partially ordered tag model. Note that $s_1 = \{e_i, i \in \mathbb{N}\}$ where $\forall\, i, j \cdot i < j \Rightarrow T(e_i) < T(e_j)$, and $s_2 = \{e'_i, i \in \mathbb{N}\}$ where $\forall\, i, j \cdot i < j \Rightarrow T(e'_i) < T(e'_j)$, and we say $e_1 < e_2$ when $e_1 = (t_1, v)$ and $e_2 = (t_2, v)$ for some $v \in V$ where $t_1 < t_2$.

Reo Connector	Process	Behavior
		$P = \{(s_1, s_2) \mid s_1 = s_2\}$
		$P = \{(s_1, s_2) \mid \forall k \in \mathbb{N} \cdot (e_k < e'_k < e_{k+1})\}$
		$P = \{(s_1, s_2) \mid s_2 \subseteq s_1\}$

5 Discussion, Conclusion and Future Work

We observed the similarity between Tagged Signal Model presented in [6] and Timed Data Stream presented in [2]. Different techniques are established based on Tagged Signal Model to understand the behavior of a model of computation better, and to reason about the determinacy of the composition of processes at the syntactic level using fixed point theory. Our main motivation was to discuss the possible nondeterministic behaviors of different Reo connectors specially when a feedback loop is formed in the composition, by exploiting TSM framework and its results on systems with feedback loops. We provided two denotational semantics for Reo connectors in the two timed and untimed models of computation, based on totally and partially ordered sets of tags respectively.

Moving towards a partially ordered set of tags, we may stick to the way the primitive connectors are modeled in [2], but we need to show how we compare different tags coming from different sources where necessary, or show how and where we can resolve this comparison. We need to consider the rendezvous that is happening within Reo nodes to model the change of tags and propagation of change of tags on the upstream signals coming in from other processes, and downstream signals going to other processes. The details for this model is left as the future work.

Acknowledgment. We would like to thank Professor Edward Lee for his very useful comments and discussions.

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
2. Arbab, F., Rutten, J.J.M.M.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2002. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40020-2_2
3. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. **61**(2), 75–113 (2006)
4. Jongmans, T.Q., Arbab, F.: Overview of thirty semantic formalisms for Reo. Sci. Ann. Comput. Sci. **22**(1), 201–251 (2012)
5. Lee, E.A.: Concurrent Models of Computation - An Actor-Oriented Approach (Draft) (2011)
6. Lee, E.A., Sangiovanni-Vincentelli, A.L.: A framework for comparing models of computation. IEEE Trans. CAD Integr. Circuits Syst. **17**(12), 1217–1229 (1998)
7. Liu, X.: Semantic foundation of the tagged signal model. Ph.D. thesis, EECS Department, University of California, Berkeley (2005)

Generating Arduino C Codes from *Mediator*

Yi Li and Meng Sun(✉)

LMAM and Department of Informatics, School of Mathematical Sciences,
Peking University, Beijing, China
liyi_math@pku.edu.cn, sunmeng@math.pku.edu.cn

Abstract. Manual encoding is exceedingly time consuming and error prone, and has become a huge obstacle between reliable software models and trustworthy computer programs. To deal with this problem, dozens of code generators are developed to automatically convert different models into executable codes. In this paper we present a new code generator for the component-based modeling language *Mediator*. It aims to generate platform-dependent (Arduino in this case) programs that can be directly downloaded to the hardware without any manual adaption. We also present a case study where we use *Mediator* to develop a wheeled-robot controller, generate the corresponding program through our code generator, which has been successfully executed on an Arduino-based robot platform.

Keywords: Code generation · Mediator · Arduino
Component-based modeling

1 Introduction

With IoT techniques sweeping around the world, software systems are becoming more complicated, distributed and safety-critical, and thus the development of such systems is becoming notoriously difficult. Small failures in daily-used software, such as smart house controllers, payment applications, etc., may lead to severe butterfly effect. Under such circumstances, various approaches have been proposed in the past decades to help software development, such as object-oriented programming [20], aspect-oriented programming [15], component-based modeling and development [21], and so on. Among these approaches, component-based modeling is extremely popular and helpful in the development of embedded systems [12] and service-oriented applications [3].

Mediator [16] is a new component-based modeling language that provides an automata-based formal semantics and supports hierarchical modeling. With the help of a full-featured type system and powerful coordination mechanisms, this language can be used by both domain-specific experts and software engineers to guarantee the reliability of software system models.

F. de Boer et al. (Eds.): Arbab Festschrift, LNCS 10865, pp. 174–188, 2018.
https://doi.org/10.1007/978-3-319-90089-6_12

However, such powerful modeling languages can only help with the correctness of high-level models. In practice, lots of software errors are caused by the inconsistency between implementations and models. And it turns out that manual implementations do not always precisely follow abstract models and designs, especially for systems with high complexity. To address this problem, automatic code generation has been proposed to avoid errors caused by human activities in the implementation process [10]. In this paper, we present an algorithm that generates C code from *Mediator* models, which can be directly compiled and executed on Arduino [17], a popular open-source embedded platform.

Importance of code generation has been uncovered for a long time. As a result, a large number of formal and industrial code generation tools have been built for different target platforms. For example, Event-B [13] and SCADE [9] are very popular formal tools that can generate executable codes from abstract models. The code generator in SCADE is especially famous for its reliability, but its scalability is restricted by the Esterel language it accepts. As a synchronous language, Esterel only formalizes a single embedded control loop, which makes it hard to model concurrent and timed behavior. Furthermore, both Event-B and SCADE code generators only aim at x86 platform, i.e. they cannot generate code that directly works on embedded systems.

On the other hand, Ptolemy [19], MATLAB Simulink Toolbox [14] and LabVIEW [18] are the most famous industrial modeling tools that support platform-dependent code generation. These tools have large number of libraries and plugins that almost cover all commonly-used embedded platforms and programming languages. Nevertheless, power of these tools also becomes limitation when we try to perform testing or verification techniques on their models. For example, Ptolemy uses standard JAVA as its semantics, and Simulink enables users to write components directly through MATLAB language. As far as we know, models directly written in such full-featured programming languages, with loop and dynamic memory allocation, are very hard to be verified.

The rest of this paper is structured as follows: Sect. 2 briefly introduces the *Mediator* language and the Arduino platform. Then in Sect. 3 we illustrate how the *Mediator*-Arduino code generator works. Section 4 presents the wheeled-robot controller as a case study. Finally, Sect. 5 concludes the paper.

2 Background

This section gives a brief introduction on the background, including the grammar of *Mediator* and basic programming on Arduino [17] platforms. To avoid ambiguity, in the following texts we use `monospaced` to indicate any literals or source code fragments and *italic* to indicate non-terminal symbols.

2.1 *Mediator*

Mediator is a component-based modeling language proposed in [16]. As a hierarchical modeling language, *Mediator* provides formalisms for both high-level *system* layouts and low-level *automata*-based behavior units.

As implied by the name, the language tries to provide a mediator through different language elements. For example, *systems* are designed for software engineers who may have no background about formal methods, which makes it easier to construct software models with reliable components and connectors. On the other hand, these reliable components and connectors are supposed to be built using *automata* by researchers who know formal methods well.

Syntax tree of a typical *Mediator* program is defined as follows.

$$\boxed{\mathit{program} ::= (\ \mathit{typedef} \mid \mathit{function} \mid \mathit{automaton} \mid \mathit{system}\ )^*}$$

*Typedef*s are aliases for types. *Function*s are definitions of custom (or *native*, which will be interpreted later) functions. *Automaton*s and *system*s are the core modeling elements in *Mediator*. They are also called *entities* since they share the same declaration form.

Type System. Mediator provides various data types that are widely used in different formal modeling languages and programming languages. These data types are categorized as *primitive types* and *composite types*, which are presented in Tables 1 and 2, respectively.[1]

Table 1. Primitive data types

Name	Declaration	Term example
Integer	`int`	`-1,0,1`
Bounded Integer	`int lowerBound .. upperBound`	`-1,0,1`
Double	`double`	`0.1, 1E-3`
Boolean	`bool`	`true, false`
Character	`char`	`'a', 'b'`
Enumeration	enum item$_1$, ..., item$_n$	`enumname.item`

Table 1 shows the primitive types supported by *Mediator*: *integers and bounded integers, double values, boolean values, single characters* and *finite enumerations.*

Parameter Types. In some situations we may hope to reuse an automaton or a system with the help of generalization. For example, an encrypted communication system supports different encryption algorithms encapsulated as parameter functions or components. Parameter types make it possible to take functions and entity interfaces as template parameters. *Mediator* supports two parameter types:

[1] Some notations in Tables 1 and 2 are slightly different from the original language proposal in [16] since both the tool and language are still being frequently updated.

Table 2. Composite data types (`T` denotes an arbitrary data type)

Name	Declaration
Tuple	$(T_1,\dots,T_n)$
Union	$T_1 \mid \dots \mid T_n$
Array	T [*length*]
List	T []
Map	`map` [T_{key}] T_{value}
Struct	`struct` { $field_1 : T_1, \dots,\ field_n : T_n$ }
Initialized	T_{base} `init` *term*

1. *Interface*, denoted by `interface` $(port_1 : T_1, \cdots, port_n : T_n)$, defines a parameter that could be any *automaton* or *system* with exactly the same interface (i.e. number, types and directions of the ports perfectly match the declaration). Interfaces are only used in templates of *system*s.
2. *Function Type*, denoted by `func` $(arg_1 : T_1, \cdots, arg_n : T_n) : T$, defines a function that has the argument types $T_1, \cdots, T_n$ and result type T. Functions are permitted to appear in templates of *other functions*, *automata* and *system*s.

Parameter types can only be used in template parameters. It is impossible to declare a function or an interface as a local variable.

Functions. Mediator supports two types of functions, *common* functions and *native* functions. The syntax of functions is shown as follows.

$$\begin{aligned}
\mathit{funcDecl} &::= \texttt{native}^{?}\ \texttt{function}\ \mathit{template}^{?}\ \mathit{identifier}\ \mathit{funcInterface}\ \{\\
&\qquad (\ \texttt{variables}\ \{\ \mathit{varDecl}^{*}\ \}\ )^{?}\\
&\qquad \texttt{statements}\ \{\ (\ \mathit{assignStmt} \mid \mathit{iteStmt}\ )^{*}\ \mathit{returnStmt}\ \}\\
\mathit{funcInterface} &:: = \texttt{(}\ (\ \mathit{identifier}\ \texttt{:}\ \mathit{type}\ )^{*}\ \texttt{)}\ \texttt{:}\ \mathit{type}\\
\mathit{assignStmt} &::= \mathit{term}\ (\ \texttt{,}\ \mathit{term}\ )^{*}\ \texttt{:=}\ \mathit{term}\ (\ \texttt{,}\ \mathit{term}\ )^{*}\\
\mathit{iteStmt} &::= \texttt{if}\ \texttt{(}\ \mathit{term}\ \texttt{)}\ \mathit{stmt}^{+}(\ \texttt{else}\ \mathit{stmt}^{+})^{?}\\
\mathit{returnStmt} &::= \texttt{return}\ \mathit{term}\\
\mathit{varDecl} &::= \mathit{identifier}\ \texttt{:}\ \mathit{type}\ (\ \texttt{init}\ \mathit{term}\ )^{?}
\end{aligned}$$

Common functions are composed of *function interfaces* and *function bodies.* Function interfaces describe the input variables and return type of functions. Function bodies, including local variables and statements, specify the behavior of the functions. All user-defined functions are common functions.

Native functions, on the other hand, have no function bodies but only function interfaces. Similar to the function declarations in other programming languages like C headers, native functions are part of the *Mediator* plugins where the behavior of the functions cannot be described through *Mediator* statements directly. More discussions and examples of native functions will be presented in Sect. 3.

Entities. Both automata and systems are called *entities* in *Mediator*. All *Mediator* entities have their own templates and interfaces. However, ways to formalize their behavior are complete different.

Automata. Syntax tree of automata is shown as follows.

$$\begin{aligned}
\mathit{automaton} ::= {} & \texttt{automaton}\ \mathit{template}^{?}\ \mathit{identifier}\ \texttt{(}\ \mathit{port}^{*}\ \texttt{)}\ \{ \\
& (\,\texttt{variables}\ \{\ \mathit{varDecl}^{*}\ \}\,)^{?} \\
& \texttt{transitions}\ \{\ \mathit{transition}^{*}\ \}\ \} \\
\mathit{port} ::= {} & \mathit{identifier}\ \texttt{:}\ (\,\texttt{in} \mid \texttt{out}\,)\ \mathit{type} \\
\mathit{transition} ::= {} & \mathit{guardedStmt} \mid \texttt{group}\ \{\ \mathit{guardedStmt}^{*}\ \} \\
\mathit{guardedStmt} ::= {} & \mathit{term}\ \texttt{->}\ (\ \mathit{stmt} \mid \{\ \mathit{stmt}^{*}\ \}\ ) \\
\mathit{stmt} ::= {} & \mathit{assignStmt} \mid \mathit{iteStmt} \mid \texttt{sync}\ \mathit{identifier}^{+}
\end{aligned}$$

As the basic behavior unit in *Mediator*, *automaton* consists of four parts: *templates*, *interfaces*, *local variables* and *transitions*, which are interpreted respectively as follows.

1. *Templates.* Templates of an automaton include a set of parameter declarations. A parameter can be either a type (common type or parameter type) or a value. Concrete values or types are supposed to be provided when the automaton is instantiated (i.e. declared in systems).
2. *Interfaces.* Interfaces consist of directed ports and describe how automata interact with their contexts. Ports can be regarded as structures with three fields: `value`, `reqRead` and `reqWrite`, which correspondingly denote the values of parts, status of reading requests and status of writing requests.
3. *Local Variables.* Each automaton contains a set of local variables. Types of these variables are supposed to be *initialized*. We use the evaluations of local variables to represent states of an automaton.
4. *Transitions.* Behavior of an automaton is defined by guarded transitions. Each transition consists of a boolean term *guard* and a sequence of statements. Transitions are ordered by their priority. For example, if multiple transitions are activated at the same time, the one that has highest priority will be fired. On the other hand, non-deterministic firing is also supported by encapsulating part of the transitions through `group`.

Currently, *Mediator* supports three types of statements:

1. *Assignment* statements, each including an expression and an optional assignment target, evaluate the expression and assign the result to its target if possible.
2. *Ite* (if-then-else) statements act as conditional choice statements in other languages.
3. *Synchronizing* statements, labelled with `sync`, are the flags requiring synchronized communication with other entities.

According to the existence of synchronizing statement (i.e. external communication through ports), transitions are classified as either *internal* transitions or *external* ones.

Compared with automata models being widely-used in other formal tools (e.g. UPPAAL [4], Simulink/Stateflow [14]), an automata in *Mediator* has no explicitly declared locations. Instead, it uses the evaluation of local variables to represent its states. An example of *Mediator* automaton can be found in Example 2, Sect. 4, where automata are used to model drivers of motors.

$$\begin{aligned}
\mathit{system} ::= {}& \texttt{system}\ \mathit{template}^{?}\ \mathit{identifier}\ \texttt{(}\ \mathit{port}^{*}\ \texttt{)}\ \{ \\
& (\,\texttt{internals}\ \mathit{identifier}^{+})^{?} \\
& (\,\texttt{components}\ \{\ \mathit{componentDecl}^{*}\ \}\,)^{?} \\
& \texttt{connections}\ \{\ \mathit{connectionDecl}^{*}\ \}\ \} \\
\mathit{componentDecl} ::= {}& \mathit{identifier}^{+}\ \texttt{:}\ \ \mathit{systemType} \\
\mathit{connectionDecl} ::= {}& \mathit{systemType}\ \ \mathit{params}\ \ \texttt{(}\ \mathit{portName}^{+}\ \texttt{)}
\end{aligned}$$

Systems. As the textual representation of hierarchical entities to organize subentities (automata and simpler systems), *systems* with the above syntax tree are composed of:

1. *Components.* Entities can be placed and instantiated in systems as components. Each component is considered as a unique instance and executed in parallel with other components and connections. Ports of a component can be referenced through *comp.port* once the component is declared, where *comp* is the name of the component and *port* is the name of the referenced port.
2. *Connections.* Connections are used to connect *(a) the ports of the system itself, (b) the ports of its components, and (c) the internal nodes.* Inspired by the Reo project [6–8], complex connection behavior can also be determined by other entities.
3. *Internals.* Sometimes we need to combine multiple connections to perform more complex coordination behavior. Internal nodes, declared in `internals` segments, are untyped identifiers which are capable to weld two ports with consistent data-flow direction.

Systems also have *templates* and *interfaces* which have exactly the same forms as in automata. An example of a *Mediator* system is presented later in Sect. 4.

2.2 Arduino

Arduino [17] is an open-source electronics project that aims to build easy-to-use hardware and software. Arduino boards support various models of single-board micro-controllers, properly encapsulate them and expose a set of simple APIs to users. Here we give a brief introduction on program structure of Arduino C and some relevant hardware resources on a typical Arduino motherboard.

Program Structure. The Arduino community has developed a simple IDE that uses a dialect of C as its programming language. A typical Arduino C program describes its behavior through a `setup` function and a `loop` function.

- `setup()`: This function is called once when a sketch starts after power-up or reset. It is used to initialize variables, input and output pin modes, and other libraries needed in the sketch.
- `loop()`: After `setup` has been called, function `loop` is executed repeatedly in the main program. It controls the board until the board is powered off or is reset.

Hardware Resources. Pins are the most important hardware resources of Arduino boards. Through them the motherboard communicates with its accessories, e.g. sensors, motors and other devices. Numbers and types of pins vary a lot between different Arduino motherboards. Here we take Arduino Uno, one of the most popular Arduino motherboards, as an example to introduce types of pins. This motherboard is also used for the case study in Sect. 4.

1. *Analog Pins:* 6 analog pins named `A0 .. A5` are provided on Arduino Uno to perform analog signal transmission. Resolution of analog pins is 10 bits, in other words, value of an analog pin varies from 0 to 1023. Reading and writing operations on analogs pins are performed through builtin functions `analogRead(`*pin*`)` and `analogWrite(`*pin*, *value*`)`.
2. *Digital Pins:* Digital pins have only two possible values 0 and 1, or `LOW` and `HIGH`. Builtin functions `digialRead` and `digitalWrite` are used to read values from and write values to digital pins. Moreover, part of the digital pins provide Pulse-Width Modulation (PWM, [2]) feature to transfer analog value through binary encoding. In this case we are supposed to use `analogRead` and `analogWrite` instead.

An Arduino pin can be in either *INPUT* mode or *OUTPUT* mode. Modes of pins, no matter whether they are analog or digital, are configured through the builtin function `pinMode`(*pin*, *mode*).

3 Code Generation

In this section we introduce the Arduino code generator for *Mediator*. The code generator mainly consists of a native function library and a set of generators for different *Mediator* language elements, e.g. types, functions and entities.

The Arduino code generator is implemented as a plugin of the *Mediator* project [1], which is written in Java and based on Maven framework [5]. Executable Jar packages and help documents can be found in this repository.

3.1 Native Functions

Native functions are the bridges between software controllers and hardware resources in the *Mediator* framework. In other words, they are the exposed interfaces of hardware through which *Mediator* models may change the hardware behavior. Similar to C/C++ header files, these native functions are declared in a *Mediator* source file as a library. But their corresponding hardware behavior is defined by the code generator plugin.

The Arduino code generator supports the following native functions:

```
native function digitalRead (pin: int) : int 0..1;
native function digitalWrite (pin: int, val:int 0..1);
native function analogWrite (pin: int) : int 0..1023;
native function analogWrite (pin: int, val:int 0..1023);
native function delay (milliseconds: int);
```

- *DigitalRead* reads binary signals from a digital pin.
- *DigitalWrite* writes binary signals to either a digital or an analog pin. When it is performed on an analog pin, it configures the analog electronic level to either 1023 (`HIGH`) or 0 (`LOW`).
- *AnalogRead* reads analog signals from an analog pin.
- *AnalogWrite* writes analog signals to either an analog pin or a digital pin that supports PWM encoding.
- *Delay* forces the Arduino processor to suspend for a certain time delay.

3.2 Type Generator

Arduino C naturally supports most of the primitive types in Table 1 and part of the composite types in Table 2: unbounded integers `int`, float point numbers `double`, characters `char`, boolean values `bool` (as zero and non-zero integers), enumeration `enum`, union `union`, structure `struct` and finite arrays.

For the other types that are not directly supported by Arduino C, we use an attached runtime library to simulate their behavior. Such types include:

- *Bounded Integer*. In *Mediator*, bounded integers are mainly used to avoid overflow and unexpected values. For example, an Arduino analog signal varies between 0 and 1023, writing any other integers to an analog pin may lead to unknown behavior. Bounded integers are not supported by C. Moreover, according to its widely use we may suffer from performance degradation if we use complex structures to represent them. So we choose to generate assertions every time when a variable of bounded integer type is assigned. Assertions are mainly used for static model checking or reasoning tools, e.g. cbmc [11]. For example,

```
int a = 0; // type of a is int 0 .. 1 init 0
void loop() {
    // ...
    a = 1 - a;
    assert (a >= 0 && a <= 1);
    // ...
}
```

- *List.* C supports unbounded lists by pointers. However, C does not care about the capacity and consumption of them, which frequently lead to memory overflow and invalid dereference. In the *Mediator* runtime library, we encapsulate a void pointer to represent an unbounded list, and two integer fields to denote its capacity and the number of items existing in this list.

```
struct __MR_List {
    void * list;
    int capacity;
    int num_items;
    int item_size;
};
typedef struct __MR_List MR_List;

void init_empty_list (MR_List list, int item_size);
void list_add (MR_List list, void * item);
void * list_get (MR_List list, int index);
void list_del (MR_List list, int index);
```

  The definition of unbounded list, as shown here, is type-independent. In other words, when storing items to or obtaining items from an unbounded list, type casting is unavoidable. In this case the *Mediator* syntax checker is responsible to guarantee the type consistency.
- *Map.* Similar to the *list*, *Mediator map* in Arduino C is also type-independent. A map uses two unbounded lists to store keys and values, respectively.

```
struct __MR_Map {
    MR_List keys;
    MR_List values;
};
typedef struct __MR_Map MR_Map;

void init_empty_map (MR_Map map, int key_size, int value_size);
void map_put (MR_Map map, void * key, void * value);
void * map_get (MR_Map map, void * key);
void map_del (MR_Map map, void * key);
```

- *Initialized.* In an Arduino program, default value of a type is only used in the `setup` function to initialize the corresponding variable. As a result, we do not have to use initialized type in Arduino C explicitly. For example, `int init 0` will be simply replaced by `int` when the Arduino C code is generated.

3.3 Function Generator

As mentioned before, there are two types of functions to be considered here: common functions and native functions.

Common Functions. When designing *Mediator*, we deliberately restrict the expressiveness of common functions (transitions as well) so that they are easier to be verified formally. As a result, common functions in *Mediator* are very easy to be encoded in C.

Native Functions. When a native function is called in a transition, the code generator needs to replace the function with corresponding native API in Arduino C. According to Sect. 2.2, all native functions supported in this plugin can be mapped to an Arduino API with the same name.

3.4 Entity Generator

All Arduino motherboards are equipped with only one processor and there is no time-sharing operating system support. They do not support parallel execution. Consequently, typical *Mediator* systems including a set of parallel components and connections can not be directly encoded in Arduino C.

Fortunately, a scheduling algorithm that flats a hierarchical system into a single automaton has been introduced in [16]. The algorithm guarantees that its resulting automaton is always canonical, i.e., the automaton contains exactly one transition group, in which all transitions are also canonical. With help of this algorithm, all we need to do is to encode a single *Mediator* automata in Arduino C.

The following steps illustrate the sketch of the encoding process.

1. *Template and Interface.* When generating Arduino codes, we always assume that the source automaton has NO template parameters and NO ports. Ports are special elements in *Mediator* that are used to react with a *Mediator* context. Behavior of ports are undeclared in the Arduino C context.
2. *Local variables* are declared as global variables in Arduino C. According to [16], all local variables should be initialized, i.e., they are declared with default values. And these default values will be assigned to the global variables in the `setup` function.
3. *Statements.* Transitions are composed of sequences of statements. After being scheduled and canonicalized, an automaton contains only *assignment statements* and *ite statements* (which are inherently supported in C). Since we assume that no port exists in the automaton's interface, *synchronizing statements* can be simply omitted.
4. *Transitions.* Transitions are *activated* if their guards are satisfied by the current evaluation of the local variables. Since in a canonical automaton all transitions are encapsulated by a `group`, the transition selection process is fully non-deterministic, i.e. the transition to fire is randomly selected from all *activated* transitions. In our approach, we use the following three steps to perform transition selection and firing:
   - Step 1. *Activation checking.* At the beginning of each `loop`, we use a set of `if` statements to check which transitions are activated under the current evaluation of local variables. We use an array `cmd_activated` to store indexes of all transitions being activated.

- Step 2. *Random selection.* With the help of the `random` function in Arduino, it is easy to pick up a random index number from `cmd_activated`.
- Step 3. *Transition firing.* Another set of `if` blocks are used to encode the statements in transitions. Conditions of these blocks are used to check whether index of this transition is equal to the selected index.

Example 1. Consider a *Mediator* automaton `test` with one local variable x (initialized by 0) and two transitions: increasing x by 1 if x is less than zero, or decreasing x by 1 otherwise. Source model of this automaton in *Mediator* is shown as follows.

```
automaton test() {
    variables { x: int init 0; }
    transitions {
        x < 0 -> { x = x + 1; }
        true -> { x = x - 1; }
    }
}
```

Since the transitions are ordered by their priority in *Mediator*, the second transition can be fired iff. the first one is not activated. The generated Arduino C code is also presented:

```
int test_x;
int cmd; // stores the index of selected transition
int cmd_activated[1]; // the capacity depends on number of
     transitions that belongs to the automaton

void setup() { test_x = 0; }

void loop() {
    // STEP 1 collect activated transitions
    cmd_activated_counter = 0; // the stack pointer of cmd_activated
    if (test_x < 0) {
        cmd_activated[cmd_activated_counter] = 0;
        cmd_activated_counter ++;
    }
    if (test_x >= 0) {
        cmd_activated[cmd_activated_counter] = 1;
        cmd_activated_counter ++;
    }

    // STEP 2 pick up a transition randomly
    cmd = cmd_activated[random(cmd_activated_counter)];

    // STEP 3 fire the selected transition
    if (cmd == 0) test_x = test_x + 1;
    if (cmd == 1) test_x = test_x - 1;
}
```

The code generating process is summarized in Algorithm 1.

Algorithm 1. Generate Codes for a Specified Entity E in a Program P

```
Require: A program $P = \langle Typedefs, Functions, Automata, Systems\rangle$, an entity $E$
Ensure: Arduino C codes
$global, setup, loop \leftarrow$ “ ”
if $E \in Automata$ then
    $A \leftarrow$ Canonicalize($E$)
else
    $A \leftarrow$ Schedule($E$)
end if
if $A.Ports \neq \varnothing$ then
    return NULL
end if
for $var \in$ {local variables of $A$} do
    add variable declaration of $var$ to $global$ with the generated type
    add variable initialization of $var$ to $setup$
end for
for $t = guard \rightarrow statements \in$ {transitions of $A$} do
    add activation checking of $guard$ to $loop$
end for
add random index selection to $loop$
for $t = guard \rightarrow statements \in$ {transitions of $A$} do
    add the generated $statements$ to $loop$
    if pin $pin$ is involved in $statements$ then
        add pinMode to $setup$ to configure $pin$ correctly
    end if
end for
$setup \leftarrow$ “void setup(){” + $setup$ + “}”
$loop \leftarrow$ “void loop() {” + $loop$ + “}”
return $global + setup + loop$
```

4 Experiment

In this section, we show how to model a wheeled-robot controller in *Mediator* and generate Arduino C code through our code generator. The generated platform-dependent code has been directly compiled and flashed to the motherboard, without any manual modification.

The hardware platform being used is based on an Arduino Uno motherboard, which consists of 4 motors (divided into two group *left and* right) and an ultrasonic distance sensor.

The *Mediator* model of the wheeled-robot controller as shown in Fig. 1 contains the following parts:

- *UltraSonic*. This sensor detects the distance from nearest obstacles and sends the distance information to the controller.

- *Controller*. The core algorithm of this robot is encapsulated in the controller. It reads distance information from the ultrasonic sensor, and gives an abstract command (e.g. forward, backward, turn, stop) to the *speeder* according to the distance information.
- *Speeder*. This automaton updates the speed of two motor groups according to the abstract command it receives from the *controller*.
- *Motors*. 4 motors, divided into two groups, are equipped in this small robot. Each motor has two control pins, one for direction and the other for speed. The *Mediator* automaton shown in Example 2 is the driver for motors. It receives a single control signal *speed* and updates the electronic level of control pins correspondingly.

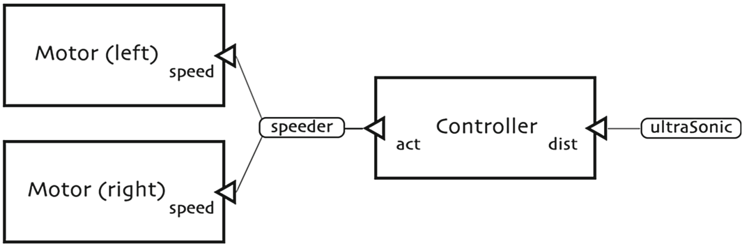


Fig. 1. Mediator model of the robot controller

The controller shown in Fig. 1 is captured by the following *Mediator* code, where the motors and the controller are defined as components, and the speeder and ultrasonic distance sensor are defined as connections.

```
system robot () {
    components {
        left_motor : motor<8, 9>;
        right_motor : motor<11, 10>;
        c  : controller;
    }
    connections {
        speeder(c.act, left_motor.speed, right_motor.speed);
        ultraSonicDist<6,7>(c.dist);
    }
}
```

Four *Mediator* automata are specified in this controller model: *motor*, *controller*, *speeder* and *ultraSonicDist*. Here we only show the definition of *motor*, further details can be found at [1].

Example 2 (Motor Driver). A typical driver of motors with two control signals is defined as an automaton in *Mediator*. The simple automaton contains no local variable, one internal transition and one external transition. The internal transition updates the status of port `speed`, which is supposed to be ready to accept control commands at any time. And the external transition receives target speed from the `speed` port and gives orders to the hardware correspondingly. The two template parameters describe where Arduino pins the motor is connected.

```
automaton <pinDirection,pinSpeed:int> motor (speed:in signedPWM) {
    variables {}
    transitions {
        !speed.reqRead -> speed.reqRead = true; // internal
        speed.reqRead && speed.reqWrite -> {
            sync speed; // external communication flag
            if (speed.value > 0) {
                digitalWrite(pinDirection, 1);
                analogWrite(pinSpeed, speed.value);
            } else {
                digitalWrite(pinDirection, 0);
                analogWrite(pinSpeed, -speed.value);
            }
        }
    }
}
```

Due to the length limitation, the generated program is omitted here and can be found at https://github.com/mediator-team/codegen-proposal/experiment.

5 Conclusion and Future Work

In this paper, we presented a fully-automatic code generator that converts *Mediator* models to executable Arduino C programs. Compared with plain Arduino C code, component-based *Mediator* models can be defined in a more intuitive, easier way. With the help of this code generator, engineers are able to build and review their models in *Mediator*, and generate Arduino C code automatically to avoid errors caused by manual encoding.

In the future we plan to use program verification tools to guarantee the reliability of generate codes. Due to the complexity, it is hard to formally prove the correctness of the code generator itself. So we plan to generate assertions automatically and insert them into the target code for formal verification. Hopefully they can be verified through program verification tools, e.g. CBMC [11]. Providing support for more hardware platforms and languages, e.g. Verilog/VHDL, System C, etc., are in our scope as well.

Acknowledgement. The work was partially supported by the National Natural Science Foundation of China under grant no. 61772038, 61532019, 61202069 and 61272160.

References

1. Mediator GitHub repository. https://github.com/mediator-team
2. Wikipedia page of pulse-width modulation. https://en.wikipedia.org/wiki/Pulse-width_modulation

3. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services - Concepts, Architectures and Applications. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-10876-5
4. Amnell, T., et al.: UPPAAL - now, next, and future. In: Cassez, F., Jard, C., Rozoy, B., Ryan, M.D. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 99–124. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45510-8_4
5. Apache Foundation: Apache Maven. http://maven.apache.org/
6. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
7. Arbab, F., Baier, C., de Boer, F., Rutten, J.: Models and temporal logical specifications for timed component connectors. Softw. Syst. Model. **6**(1), 59–82 (2007)
8. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. **61**(2), 75–113 (2006)
9. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2), 87–152 (1992)
10. Budinsky, F., Finnie, M., Vlissides, J., Yu, P.: Automatic code generation from design patterns. IBM Syst. J. **35**(2), 151–171 (1996)
11. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
12. Cmkovic, I.: Component-based software engineering for embedded systems. In: Roman, G.C., Griswold, W.G., Nuseibeh, B. (eds.) Proceedings of ICSE 2005, pp. 712–713. ACM (2005)
13. Cataño, N., Rivera, V.: EventB2Java: a code generator for Event-B. In: Rayadurgam, S., Tkachuk, O. (eds.) NFM 2016. LNCS, vol. 9690, pp. 166–171. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40648-0_13
14. Hahn, B., Valentine, D.T.: SIMULINK toolbox. In: Essential MATLAB for Engineers and Scientists, pp. 341–356. Academic Press (2016)
15. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0053381
16. Li, Y., Sun, M.: Component-based modeling in mediator. In: Proença, J., Lumpe, M. (eds.) FACS 2017. LNCS, vol. 10487, pp. 1–19. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68034-7_1
17. Margolis, M.: Arduino Cookbook. O'Reilly Media Inc., Sebastopol (2011)
18. National Instruments: LabVIEW. http://www.ni.com/zh-cn/shop/labview.html
19. Ptolemaeus, C. (ed.): System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014)
20. Sebesta, R.W.: Concepts of Programming Languages, 10th edn. Pearson, Upper Saddle River (2012)
21. Szyperski, C., Gruntz, D., Murer, S.: Component Software - Beyond Object-Oriented Programming, 2nd edn. Publishing House of Electronics Industry, Beijing (2003)

From Soft Agents to Soft Component Automata and Back

Carolyn Talcott(✉)

SRI International, Menlo Park, CA 94025, USA
carolyn.talcott@sri.com

Abstract. Rewriting Logic and Automata are complimentary approaches for developing executable models of concurrent/distributed systems that can be analyzed by prototyping, and multiple methods of model-checking. A joint project between my group at SRI and Farhad's group at CWI is developing formal methods to diagnose the cause of undesired behavior of autonomous (cyber physical) systems operating in unpredictable environments. CWI is working on theory development based on automata, exploring composition mechanisms in multiple dimensions, and developing logic that supports reasoning about compositionality. The SRI work is based on rewriting logic and is focused on methods for system specification and model-checking in the context of faults and environmental threats. The two approaches share a common feature, namely the assignment of preferences to possible actions to model locally robust adaptive behavior. Preferences are elements of constraint semirings (soft constraints), structures that provide operations for comparison and composition.

In this paper we explore the similarities, differences and synergies highlighting the insights that arise by pursuing complimentary approaches.

1 Introduction

We are interested in systems of (semi) autonomous cyber-physical agents that operate in unpredictable, possibly hostile, environments using locally obtainable information. How can we specify robust agents that are able to operate alone and/or in cooperation with other agents? What properties or invariants are important? How can they be verified? How can we determine the cause of undesired behavior?

Already there are impressive point solutions coming from both research and industrial settings. Capabilities for flying quadrotor robots for navigation in 3-dimensional space, sensing other entities, and forming ad hoc teams have

The work was partially supported by ONR grant N00014-15-1-2202.

This paper is dedicated to Farhad on the occasion of his retirement from CWI. Farhad is both a colleague and a friend. The paper is a start towards realizing the potential synergy between Reo and Maude that underlies much of our collaboration. Here's to the beginning of a new phase in a research agenda full of exciting ideas, and also an opportunity to explore new ideas and passions.

F. de Boer et al. (Eds.): Arbab Festschrift, LNCS 10865, pp. 189–207, 2018.
https://doi.org/10.1007/978-3-319-90089-6_13

been developed [12,27,28]. Case studies show that heterogeneous networks with smart phone, ground and flying robots can be controlled by a distributed logic and partially-ordered knowledge sharing [10,25,39]. Industrial applications for shipping, security, search and rescue, environmental monitoring, rail monitoring [8], oceanic data gathering [29], precision agriculture [12], automating inventory checking in large warehouses [36], and even air taxis [14] are being developed. Of course, there are Google's driverless cars and Amazon's drone delivery system.

Despite successful applications, little attention has been given to how to devise and validate strategies for cyber-physical agents that are going to (autonomously) carry out tasks. Such strategies must balance agent and environment safety with achievement of goals. Furthermore the strategies must be robust to faulty sensors and actuators and interference from external factors, such as winds or adversaries.

Formal executable models provide valuable tools for exploring system designs and verifying aspects of a systems expected behavior. Executable models are often cheaper and faster to build and experiment with than physical models, especially in early stages when ideas are developing. Such models can be built at different levels of detail and attention to faults and threats allowing the designer to focus on aspects of interest.

This Paper. We describe two complementary approaches to specification and analysis of robust cyber-physical agent systems: (i) abstract theoretical concepts based on automata and temporal logics, called Soft Component Automata; (ii) a concrete experimental approach, called Soft Agents, based on executable rewriting logic specifications, simulation, search, and model checking using the Maude system. Both approaches use *soft* constraints to model robust adaptive behavior.

- Soft Component Automata focuses on abstract characterization, compositionality of actions and systems and new logical concepts capturing compositionality properties;
- Soft Agents focuses on methods for specifying agents behavior, separation of environment model and agents knowledge, developing fault/threat models, and case studies including replacing environment by device simulators.

The question we address in this paper is how the two approaches complement each other.

Plan. In Sect. 2 we describe the soft component automata (SCA) and soft agents (SA) approaches. In Sect. 3 we show how the two approaches complement each other. In Sect. 4 we discuss related work in a number of dimensions. In Sect. 5 we summarize and suggest future directions.

2 Background

In this section we provide introductions to Soft Component Automata (SCA) and Soft Agents (SA), attempting to be sufficiently self-contained that comparing and contrasting makes sense. More details about SCA can be found in [23] and more information about SA can be found in [40,41].

A shared feature of the SCA and SA modeling formalisms is the use of a notion of *preference* to rank possible actions an agent may take. Preference may reflect importance of progress towards a goal, willingness to take risks, concern about maintaining safety and other invariants, to mention a few examples. The point is to be able to specify agent behavior that is robust to changes in situation by providing a means for evaluating and reasoning about options at each decision point. Thus we begin with a description of *constraint semirings* (aka *c-semirings*) [4,6]. A c-semiring provides an algebraic structure on preference values that allows us to both *compare* and to *compose* preferences.

2.1 Constraint Semirings

Definition 1. *A* c-semiring *is a tuple* $\langle \mathbb{E}, \bigoplus, \otimes, \mathbf{0}, \mathbf{1} \rangle$ *such that (1)* $\mathbb{E}$ *is the carrier set, with* $\mathbf{0}, \mathbf{1} \in \mathbb{E}$*; (2)* $\bigoplus : 2^{\mathbb{E}} \to \mathbb{E}$ *satisfies* $\bigoplus \emptyset = \mathbf{0}$*,* $\bigoplus \mathbb{E} = \mathbf{1}$*,* $\bigoplus \{e\} = e$*, and* $\bigoplus \{\bigoplus(E) : E \in \mathcal{E}\} = \bigoplus \bigcup \mathcal{E}$ *for* $e \in \mathbb{E}$ *and* $\mathcal{E} \subseteq 2^{\mathbb{E}}$*; and (3)* $\otimes : \mathbb{E} \times \mathbb{E} \to \mathbb{E}$ *is a commutative and associative operator, such that* $e \otimes \mathbf{0} = \mathbf{0}$*,* $e \otimes \mathbf{1} = e$*, and* $e \otimes \bigoplus E = \bigoplus \{e \otimes e' : e' \in E\}$ *for* $e \in \mathbb{E}$ *and* $E \subseteq \mathbb{E}$*.*

The induced comparison relation $\leq_{\mathbb{E}} \subseteq \mathbb{E} \times \mathbb{E}$ *is defined by* $e \leq_{\mathbb{E}} e'$ *if and only if* $e \oplus e' = e'$*.* $\leq_{\mathbb{E}}$ *is a partial order on* $\mathbb{E}$*, with* $\mathbf{0}$ *and* $\mathbf{1}$ *the minimal and maximal elements [4].*

Our c-semirings are complete lattices, with $\bigoplus$ the least upper bound operator. $\otimes$ has the property that $e \otimes e' \leq e$ for any $e, e' \in \mathbb{E}$. If $\otimes$ is idempotent, then $\otimes$ coincides with the greatest lower bound [4].

One example of c-semiring is the unit interval c-semiring $\mathbb{U} = \langle I, \max, \times, 0, 1 \rangle$ where I denotes the closed real interval between 0 and 1 ($[0, 1]$). (Also called the probabilistic c-semiring [15]) A useful family of c-semirings are the multi-level c-semirings $\mathbb{L}_n = \langle \{0, \ldots, n\}, \max, \min, 0, n \rangle$. These provide simple ways of ranking and refine the basic boolean c-semiring which is isomorphic to $\mathbb{L}_1$. A final example is the upside down or weighted c-semiring $\mathbb{W} = \left\langle \mathbb{R}_{\geq 0} \cup \{\infty\}, \inf, \hat{+}, \infty, 0 \right\rangle$ (the *weighted semiring*), where inf is the infimum and $\hat{+}$ is arithmetic addition generalized to $\mathbb{R}_{\geq 0} \cup \{\infty\}$, and $\leq_{\mathbb{W}}$ coincides with the obvious definition of the order $\geq$ on $\mathbb{R}_{\geq 0} \cup \{\infty\}$. One can think of elements of this semiring as measuring cost with preference for lower cost.

Composition operators for c-semirings include product composition [7] and (partial) lexicographic composition [15]. We refer to [22] for more details.

2.2 Soft Component Automata Formalism

Soft Component Automata (SCA) are based on Soft Constraint Automata [2,22], An SCA is a state-transition system where transitions are labeled with actions and preferences. Actions have a compositional structure given by a *Component Action System*, and preferences are taken from a c-semiring. The following draws on the presentation in [23].

Component Action Systems. A Component Action System (CAS) models how different components of a system contribute to actions that make up system level behavior. As an example, consider the battery and motion components of a simple robot. A move_r action of the robot is composed of a move action of the motion component and a $\mathsf{discharge}$ action of the battery component. Some component actions are independent and can happen alone or concurrently, some are inherently synchronous, and some are incompatible (for example move and charge). All of these possibilities are captured in the following definition (from [23]).

Definition 2. *A* Component Action System (CAS) *is a tuple $\langle \Sigma, \circledcirc, \boxdot \rangle$, such that Σ is a finite set of* actions*; $\circledcirc \subseteq \Sigma \times \Sigma$, the* composability relation*, is a reflexive and symmetric relation; and $\boxdot : \circledcirc \to \Sigma$, the* composition operator*, is an idempotent, commutative and associative operator on Σ up to $\circledcirc$ (i.e., $\boxdot$ is an operator defined only on elements of Σ related by $\circledcirc$). The* capture preorder *$\sqsubseteq$ on Σ is the induced relation such that for $a, b \in \Sigma$, $a \sqsubseteq b$ if and only if there exists $c \in \Sigma$ such that $a \circledcirc c$ and $a \boxdot c = b$.*

Soft Component Automata. Now we have the basis for defining SCA. From here on, we abuse notation by naming a c-semiring or a CAS by its set of elements $\mathbb{E}$ or Σ respectively. An SCA specifies the sequences of actions that are allowed, along with the preferences attached to such actions.

Definition 3. *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*, and $\to \subseteq Q \times \Sigma \times \mathbb{E} \times Q$ is a finite relation called the* transition relation*. We write $q \xrightarrow{a,\, e} q'$ when $\langle q, a, e, q' \rangle \in \to$.*

An SCA models the actions available in each state of the component, how much these actions contribute towards the goal and the way actions transform the state. The threshold, t, restricts the available actions to those with a preference bounded from below by t, either at run-time, or when the designer wants to reason about behaviors satisfying some minimum preference.

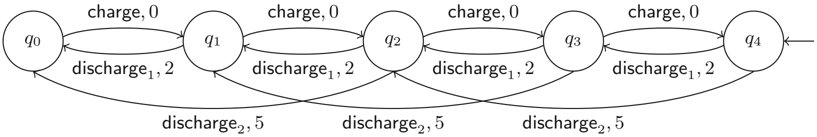


Fig. 1. A component modeling energy management, A_e.

Figure 1 shows A_e, an SCA modeling energy concerns. Its CAS contains the (incomposable) actions charge, $\mathsf{discharge}_1$ and $\mathsf{discharge}_2$. Its c-semiring is the weighted semiring $\mathbb{W}$. The threshold value t_e is left undefined for now. The states represent energy remaining, for example the initial state is named q_4, as the 4 indicates that 4 units of energy remain. Energy can be discharged one or two units at a time. Charging restores energy. The combined weight of spending two units of energy (i.e., executing the action $\mathsf{discharge}_1$ twice) is lower than the weight of spending a two units in one step (i.e., performing $\mathsf{discharge}_2$), indicates that the energy component prefers to spend its energy in small increments.[1]

Composition. Composition of SCAs is defined for pairs of automata with the same underlying c-semiring and CAS as follows.

Definition 4. *Let $A_i = \langle Q_i, \Sigma, \mathbb{E}, \rightarrow_i, q_i^0, t_i \rangle$ be an SCA for $i \in \{0, 1\}$. The* composition *of A_0 and A_1, denoted $A_0 \bowtie A_1$, is the SCA $\langle Q, \Sigma, \mathbb{E}, \rightarrow, q^0, t_0 \otimes t_1 \rangle$, where $Q = Q_0 \times Q_1$, $q^0 = \langle q_0^0, q_1^0 \rangle$, $\otimes$ is the composition operator of $\mathbb{E}$, and $\rightarrow$ is the smallest relation satisfying*

$$\frac{q_0 \xrightarrow{a_0,\ e_0}_0 q_0' \qquad q_1 \xrightarrow{a_1,\ e_1}_1 q_1' \qquad a_0 \circledcirc a_1}{\langle q_0, q_1 \rangle \xrightarrow{a_0 \boxdot a_1,\ e_0 \otimes e_1} \langle q_0', q_1' \rangle}$$

Note that in contrast to typical Reo inspired automata, where one of the components of a composition can choose to do nothing, in SCA there must be a composable action. This allows one component to prevent another from executing a given action. The designer can always choose to add `pass` actions, compatible with all actions, as in the snapshot automata, A_s, shown in Fig. 2.

To illustrate SCA composition, we introduce the Snapshot SCA A_s in Fig. 2, which models the concern of a crop surveillance drone that it should 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 $\mathbb{W}$.

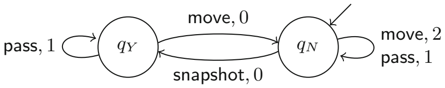


Fig. 2. A component modeling the desire to take a snapshot at every location, A_s.

We augment the CAS of A_s and A_e with composite actions α_i defined as action α composed with $\mathsf{discharge}_i$. charge is composable with pass and we define charge $\boxdot$ pass = charge in the combined CAS. The composition, $A_{\mathsf{e},\mathsf{s}}$, of A_s and A_e is depicted in Fig. 3.

The states of $A_{\mathsf{e},\mathsf{s}}$ reflect the states of its components; for instance, in state $q_{2,Y}$ there are two units of energy left, and a snapshot of the current position has been taken. The action $\mathsf{snapshot}_1$ is not available in states of the form $q_{i,Y}$, because the only action available in q_Y is pass, which does not compose into $\mathsf{snapshot}_1$.

[1] Recall that, in $\mathbb{W}$, higher values reflect a lower preference (a higher *weight*); thus, charge is preferred over $\mathsf{discharge}_1$.

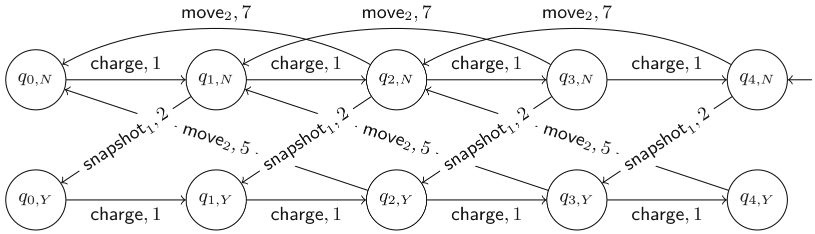


Fig. 3. The composition of the SCAs A_{e} and A_{s}, dubbed $A_{\mathsf{e,s}}$: a component modeling energy and snapshot management. We abbreviate pairs of states $\langle q_i, q_j \rangle$ by writing $q_{i,j}$.

Semantics. The (operational) semantics of an SCA is the set of traces it admits.

Definition 5. *Let* $A = \langle Q, \Sigma, \mathbb{E}, \rightarrow, q^0, t \rangle$*. A* trace *of* A *is a triple* $(\mu : Q^\omega, \sigma : \Sigma^\omega, \nu : \mathbb{E}^\omega)$ *such that* $\mu(0) = q^0$*, for all* $n \in \mathbb{N}$*,* $\mu(n) \xrightarrow{\sigma(n),\ \nu(n)} \mu(n+1)$*, and* $t \leq \nu(n)$*. The sequence of actions* σ *is called a* behavior *of* A*.*

Logic. In [23] a dialect of linear temporal logic to specify properties of SCA behaviors is proposed. The language includes two new operators that can be used to express properties concerning compositionality. Specifically the new operators are:

- The *captures* operator $\succ \phi$ describes every behavior that captures a behavior satisfying ϕ, i.e. $\sigma \models \succ \phi$ just if there are component action sequences σ_0, σ_1 such that $\sigma_0 \circledcirc \sigma_1$, $\sigma = \sigma_0 \boxdot \sigma_1$, and $\sigma_1 \models \phi$.[2]
- The *composable* operator, $\circledcirc \phi$, holds for every behavior composable with a behavior satisfying ϕ, i.e. $\sigma \models \circledcirc \phi$ just if there is a component action sequence σ_1 such that $\sigma \circledcirc \sigma_1$, and $\sigma_1 \models \phi$.

As an example, the property that the agent never misses an opportunity to take a snapshot of a new location can be expressed by

$$\phi_{\mathsf{w}} = \succ \Box(\mathsf{move} \rightarrow X(\neg \mathsf{move}\, U\, \mathsf{snapshot}))$$

This formula says "every behavior captures that, at any point, if the current action is a move, then it is followed by a sequence where we do not move until we take a snapshot". If $t_e \otimes t_s = 5$, the energy-snapshot automata $A_{\mathsf{e,s}}$ satisfies ϕ_{w}.

Diagnostics. Now that we can specify behaviors and their properties, an important question is how to deal with undesired behaviors. Suppose we have identified an undesired behavior σ of automata A. One design decision is the threshold each action preference must cross. How can we rationally adjust the threshold to eliminate σ (and still keep desired behaviors). The *diagnostic preference* of a

[2] $\circledcirc$, $\boxdot$ are lifted pointwise to action sequences.

behavior σ of A is the largest preference t such that σ is a behavior of $A(t)$ (A with the threshold replaced by t). Thus setting t above the diagnostic preference of σ will eliminate σ. An algorithm for computing the diagnostic preference is given in [23].

If A is composite, we can change the threshold of A by changing the thresholds of one or more components. The question is which components to adjust. Notions of *innocent* and *suspect* components are introduced in [23] along with an algorithm for computing them. Roughly, raising the threshold of an innocent component will not change the composite threshold sufficiently. The suspect components are those that are not innocent. They can be viewed as possible causes of undesired behavior.

Continuing the example above for property ϕ_{w}, suppose we choose $t_{\mathsf{e}} = 10$ and $t_{\mathsf{s}} = 1$; then $t_{\mathsf{e}} \otimes t_{\mathsf{s}} = 11$, thus $\sigma = \langle \mathsf{move}_2, \mathsf{charge}, \mathsf{charge} \rangle^{\omega}$ is a behavior of the snapshot automata, A_{s}, and hence $A_{\mathsf{e,s}}$ does not satisfy ϕ_{w}. The diagnostic preference of σ (in $A_{\mathsf{e,s}}$) is 7 so to eliminate σ as a behavior we must raise the threshold above 7 (i.e. make it less than 7 in the normal ordering). The threshold of A_{s} is already above the diagnostic preference ($7 \leq_{\mathbb{W}} 1$) so even raising the threshold of A_{s} to its maximum, 0, the combined threshold $t_e \otimes t_s$ will be 10 which is not good enough. Thus we raise the threshold of A_{e}, that is, we can consider t_e to be suspect. Indeed, making t_e equal 5 we have $t_{\mathsf{e}} \otimes t_{\mathsf{s}} = 6$, then σ will be eliminated as a behavior of $A_{\mathsf{e,s}}$. In the case of more than two components, the idea is to find subsets of components such that the composition of thresholds (product) is below the diagnostic preference. These are the suspect groups, and diagnosis looks for one or more thresholds is such groups to raise. Of course, eliminating one counter example by this method may leave other counter-examples to deal with.

2.3 Soft Agents

Soft Agents (SA) is formalized in the rewriting logic language Maude [11]. SA is a class of rewrite theories that extend the basic Soft Agents rewrite theory. Soft agents are introduced in [40] and described in some detail in [41]. In the following we give a brief overview.

Rewriting Logic and Maude. Rewriting logic [33] is based on two simple ideas: states of a system are represented as elements of an algebraic data type, specified in an equational theory, and the behavior of a system is given by local transitions between states described by *rewrite rules*. A rewrite rule has the form $t \Rightarrow t'$ *if* c where t and t' are terms possibly containing variables and c is a condition (a boolean term). Such a rule applies to a system in state s if t can be matched to a part of s by supplying the right values for the variables, and if the condition c holds when supplied with those values. In this case the rule can be applied by replacing the part of s matching t by t' using the matching values for the place holders in t'. The process of application of rewrite rules generates computations. The semantics of a system specified in rewriting logic is the set of computations generated by the rules, starting with the initial state.

Maude is a language and tool set based on rewriting logic [11,32]. Maude provides a high performance rewriting engine featuring matching modulo associativity, commutativity, and identity axioms; and search and model-checking capabilities. Thus, given a specification S of a concurrent system, one can execute S to find one possible behavior; use search to see if a state meeting a given condition can be reached; or model-check S to see if a temporal property is satisfied, and if not to see a computation that is a counter example. The IOP-IMaude framework [31] supports integration of Maude with other tools, for example statistical model-checkers.

Soft Agent Features. Key features of soft agents include

- separate representation of cyber (decision) and physical (action) aspects
- uniform representation of agent and environment state using partially ordered knowledge items
- locality–there is no central planner with perfect global knowledge
- distributed–agents act and interact asynchronously
- resource constrained–limited communication range, actions take time and energy, finite load capacity, . . .
- agent sensor readings, actuator effects, and communications can suffer perturbations due to faults or environment conditions such as a gust of wind, or rough terrain
- robustness to unexpected situations using soft constraint problem solving.

The Soft Agent Framework. System state is a multiset of agent and environment components. The state of an agent component is an attribute set including

- a local knowledge base attribute, `lkb`, specifying the agents knowledge–model of the world, goals, policies, . . .
- an eventset attribute, `evs`, that includes pending actions, tasks, and incoming shared knowledge
- a cached knowledge base attribute, `ckb`, specifying knowledge to be shared
- a sensor attribute, `sensors`, specifying the available sensors.

The environment component encapsulates a knowledge base, `ekb`, specifying each agent's physical state and agent independent aspects of the environment. System behavior is specified by two rewrite rules `doTask` and `timeStep`.[3]

```
crl[doTask]:
[id : cl | lkb : lkb, evs : ((task @ 0) evs), ckb : ckb,
           sensors : sset, ats]
[eid | ekb ]
=>
[id : cl | lkb : lkb', evs : evs',  ckb : ckb', sensors : sset, ats]
[eid | ekb' ]
```

[3] We chose this minimal rule set to reduce the search space by minimizing interleaving that is not important for the properties of interest.

```
if t := getTime(lkb)
/\ {ievs,devs} := splitEvents(evs,none)
/\ {skb,ekb'} := readSensors(id,sset,ekb)
/\ {lkb', evs', kb} kekset := doTask(cl, id, task,ievs, devs, skb, lkb)
/\ ckb' := addK(ckb,kb).
```

The `doTask` rule specifies an agent's process for selecting possible next actions. The rule premiss (above the `=>`) specifies the state of an agent ready to carry out this process. In particular the event set attribute, `evs:`, must contain `task @ 0`, indicating zero delay for this event. The rule conclusion (below the `=>`) specifies the agent's and environment's state after the transition using the values determined by the bindings in the condition (following `if`). The selection process is encapsulated by the `doTask` function, which is a composition of several auxiliary functions. These functions process incoming shared knowledge (`ievs`) and sensor information (`skb`) to update the agent's knowledge base (`lkb`).

Finally, the soft constraint problem (SCP) of finding and ranking available actions is solved to produce tuples {`lkb',evs',kb` } containing the updated local knowledge base, `lkb'`, the updated event set, `evs'`, that includes an action and a task event `task @ d` to schedule another `doTask` in `d` time units, and knowledge to be shared with other agents, `kb`. The SCP problem for SA ranks actions by assigning each action a value in a c-semiring. The assignment mapping depends on the current state as represented in the local knowledge base. Actions with maximal rank according to the c-semiring partial order are used to generate the tuples returned by `doTask`. The pattern {`lkb', evs', kb`} `kekset` specifies a non-deterministic choice among action tuples made by the rewriting engine. Using search, all choices will be explored. Constraint solving problems are illustrated in the case study at the end of this section.

```
crl[timeStep]:
{ aconf }
=>
{ aconf2 }
if nzt := mte(aconf)
/\ t := getTime(envKB(aconf))
/\ ekb0 := doEnvAct(t, nzt, envKB(aconf), effActs(aconf))
/\ ekb' := resolveKB(getEnvId(aconf), ekb0, envKB(aconf))
/\ aconf0 := updateEnv(ekb',timeEffect(aconf,nzt))
/\ aconf1 := shareKnowledge(aconf0)
/\ aconf2 := updateConf(aconf1).
```

The `timeStep` rule coordinates the execution of actions posted by agents and manages the passage of time. `mte(aconf)` determines how much time can pass before a task event is enabled (the minimum `d` on any `task @ d`). The variable `nzt` has sort `NonZeroTime`. Thus, if there is a task with zero delay the conditional will fail and the rule will not be enabled. In particular, all agents with `task @ 0` in their event set will get a chance to schedule actions before time passes. `doEnvAct` is a concurrent composition of `doUnitEnvAct(t,ekb,act)` over actions, `act`, posted by each agent. `doUnitEnvAct(t,ekb,act)` computes the effects of action `act` executing for unit time on the environment local to the agent executing `act`, as well as any effects external to the agent, for example occupying a location or moving an object. The expected effects (according to the physical model) may be modified by faults, threats, and natural conditions such as wind, rain, obstacles, etc., all of which are represented as knowledge items in the environment knowledge base, `ekb`.

The function `resolveKB` resolves conflicts, for example two agents attempting to move to the same location or to pick up the same object. The resolution possibilities include voiding the conflicting actions and giving one action preference. This is a modeling decision. `timeEffect(aconf,nzt)` advances time in `aconf` by `nzt` (incrementing the clock and decrementing delays).

The default `shareKnowledge` function takes each agent's posted knowledge and transmits it to all other agents that are within communication range, a model parameter. This could be refined by policies constraining the types of information delivered based on agent's interest or access control rules (represented as knowledge items).

The default `updateConf` function is the identity. It can be used to instrument the configuration to track properties of interest, for example logging visits to specific locations (the time or order), or monitoring minimum energy levels.

What Can We Do with an Executable Specification? Once a model is specified, we can define specific agent system configurations and explore the behavior by rewriting, search and statistical model-checking. We can watch the system run by rewriting according to different builtin or user defined strategies to choose next steps. In this way, specific executions and their event traces can be examined. Search allows exploration of all possible executions of a given configuration (up to some depth), to look for desired or undesirable conditions. Using statistical model checking one can obtain quantitative measures of effect of faults and threats and the expected degree of satisfaction of requirements and goals.

Patrol Bot Case Study. *Patrol bots* are an abstraction of surveillance or monitoring scenarios. A patrol bot moves on a 2-D grid (the grid dimensions are a scenario parameter). It has a preferred Y coordinate and the goal is to repeatedly go from one edge to the other and back. There is a charging station in the middle of the grid, and the bot should avoid running out of energy by recharging when energy is low.[4]

The c-semiring for a patrol bot is the lexicographic composition of $\mathbb{L}_2$, a 3 level energy c-semiring (with levels 0, 1, 2) and $\mathbb{U}$, a unit interval c-semiring. $\mathbb{L}_2$ has 3 elements named `bot`, `mid`, and `top`. Elements of the composition are represented by pairs modulo the equivalence of any two pairs where the first element is `bot`, the 0 element. This gives the energy concern priority and ensures that a move that violates energy constraints with be given 0 preference.

[4] The patrol bot case study is based on an unpublished case study consisting of a Maude specification of the SA framework and modules specifying patrol bots that are used by the project to test ideas.

The energy preference of a move action is `top` if, according to the Bot's model, after the move the energy reserve (beyond what is needed to get to the charging station) is more than the caution level. It is the `mid` value if the energy reserve is greater than 0 and the move is in the direction of the charging station. It is `bot` otherwise.

The patrol concern of a move prefers moves that get the Patrol Bot back on its designated track, `myY`, if it is off track, otherwise it prefers moves in the current direction `myDir`. If the world works according to the Patrol Bot's model the energy preference will keep the Bot from running out of energy, even with very low caution by directing it to the charging station when energy is low.

Consider a Bot on a 7 × 5 grid ($0 \leq x \leq 6$, $0 \leq y \leq 4$) with a charging station at $(3, 2)$. Suppose the Bot's designated track is $y = 1$, the energy is measured as the fraction of capacity, so 1 means fully charged, the cost of a unit move (`E,W,N,S`) is .1 (using 10% of the battery capacity), and the caution level is .1. In the current state the direction is `E`, the location is $(4, 1)$ and the energy is .5. After any move the new energy will be .4. If the move direction is `E`, the new location is $(5, 1)$ and the reserve energy is .1 so the energy preference is `bot` since `E` moves away from the station. By similar reasoning, the energy preference is `top` for directions `W` and `N` as they move closer to the station. The patrol preference for directions `W` and `N` is `0`. (`top`, `0`) is not the 0 element of the composed c-semiring, so this is a possible action.

This simple setup already provides a basis for exploring effects of grid size, energy fraction per move, as well as the effects of two or more bots competing for use of the charging station and how much caution can do to ensure safe robust behavior.

3 Soft Component Automata vs Soft Agents

Now we have the background needed to discuss relations between the Soft Agent and Soft Component Automata formalisms: how the different approaches can inform each other, which ideas from each one can be leveraged in the other.

For concreteness we will discuss synergies in the context of the Patrol Bot example. Although Patrol Bots are simplistic, they exhibit many subtle issues that specifications must consider for achieving robust adaptive agents that operate in unpredictable environments.

3.1 Preferences and Thresholds

A soft agent currently picks maximally ranked/preferred actions as candidates for execution. The SCA architecture and diagnostics results suggests a refactoring where the `doTask` function returns a set of actions paired with their preferences. The set could be restricted to preferences that pass a specified threshold or the threshold restriction could be implemented by a condition in the rewrite rule. This refactoring would facilitate automated exploration of design choices and implementation of preference guided diagnostics. Admitting actions that are not of maximal rank would allow exploring an agents behavior in environment where these actions are no longer available using Maude's search capability.

As seen in Sect. 2.3, the Patrol Bot constraint solving problems are formulated with a caution parameter that is used to determine preference for the energy concern. Actions that reduce energy reserve below the caution level are ruled out. Some undesired behaviors can be eliminated by raising the caution level. To make better use of thresholding, preference evaluation can be transformed to measure the energy reserve in the $\mathbb{U}$ c-semiring and use the threshold to play the role of caution to eliminate the same undesired behaviors.

For example, recalling the Patrol Bot scenario from the end of Sect. 2.3, the energy preference for move in direction `E` is .1, while for direction `W` it is .3 An energy threshold of .11 will eliminate moving `E` but admit moving `W`.

Note that replacing caution by thresholds provides additional flexibility, since each concern can have its own threshold, the threshold for the Bot's behavior will be a composition of those for the individual concerns. We could of course add caution levels for each concern to the constraint solving problem. This would be more expressive, but thresholds provide a simple uniform mechanism, that is explicit in the semantics and simple to control and reason about.

3.2 Composition

SCA composition happens at several levels: the c-semiring, the action algebra level, and the automata level. Automata composition is uniform and the same mechanism works for composing an agent automata from multiple action automata, and for composing a system from multiple agents.

SA composition also happens at several levels: the c-semiring, the knowledge base level, and the system configuration level. The environment knowledge base is a composition (by multiset union) of knowledge items specific to each agent and global/agent independent knowledge. For example, in the case of two agents the environment knowledge base, `ekb`, has the form `ekb0 ekb1 ekbg` where `ekb0` is the environment local to agent 0 and `ekbg` is the global environment. Agent specific knowledge includes the agents physical state as well as fault models for sensors and actions. Agent independent knowledge includes location of geographic, infrastructure, or building features, weather conditions, and possibly threat/attack models. System configuration composition is simply multiset union of agents and environment. These are essentially structural/static compositions. The dynamic aspects of composition require coordination: between an agent and its local environment in the `doTask` rule; and between the global environment and all the local environments in the `timeStep` rule. The conditions in rewrite rules play a coordination role that is played by languages like Reo [1] in automata inspired systems. The coordination in `doTask` is essentially information/data flow between the agent component and its local environment, which is accomplished by a combination of sensor reading (synchronization between the agent and sensor) and function level composition that dictates the flow of information to and from the participating components. The rule matching condition that requires `task @ 0` to be a pending event is analogous to a constraint automata condition that a port be active.

Coordination in the `timeStep` rule is more complex. It is achieved by the rule conditions that specify when the rule can fire (that requires looking at all the agent event sets), and the functions that carryout concurrent processing of actions, accumulate results, resolve conflicts, and propagate changes. Function composition provides a compact representation of the information flow coordination, but it requires some analysis to derive a flow graph or visual representation.

In the process of comparing SA and SCA, we realized that the system state could be refactored to simplify communication and coordination. As described in Sect. 2.3, in a SA system each agent's state includes `evs`, `ckb`, and `sensors` attributes. We chose to put the information in the agent as it is information about/local to the agent and is used by the agent, along with information from the environment (reading sensors) to execute the `doTask` rule. This information is also needed to decide when a rule is enabled. Since the environment knowledge base can be factored into knowledge items local to each agent plus global knowledge, it makes sense to put the `evs`, `ckb`, and `sensors` attributes in the agent specific environment component. Thus information needed for coordination decisions about enabledness and the information needed for and updated by executing actions in the `timeStep` rule are contained in the (composed) environment.

3.3 Diagnostics

To complete the connection with SCA we need to define the labeled transition system associated with an SA system specification. A transition is an instance, $l^{\sigma}_{p} : q \rightarrow q'$ of a rewrite rule. Here the rule label is l, the premiss (left-hand side) matches a subterm of ground term q at position p using substitution σ, the rules condition, if any, instantiated with σ must rewrite to `true` and q' is the result of replacing the subterm of q at p by the rule's conclusion (right-hand side) instantiated by σ. In [3], Meseguer proposes atomic proof terms as labels. These terms have the form $q(l(v_1,\ldots,v_n) \downarrow p)$, where l is the rule label, and an order on variables x_i of the rule has been chosen such that and $\sigma(x_i) = v_i$. He then abstracts from these terms for atomic formulae of the Temporal Logic of Rewriting. We propose to abstract from the specific state and start with terms of the form $l(v_1,\ldots,v_n)$. Since we are interested in actions and preferences and not in the decision process, we abstract label terms for the `doTask` rule to a silent label. We abstract the label terms for the `timeStep` rule to the set of action events, `effActs(aconf)`, executed by the rule instance. Each action event can be considered as a triple (id, a, e) where id is the agent identity, a is the action description, and e is the preference. We give the system level action the preference assigned by the agent (cyber component) via `doTask` soft constraint solving, viewing the environment as having no preference, or equivalently preference 1 for every action attempt.

Following the mechanism for composing automata, the preference for the concurrent actions in a `timeStep` transition is the product (in the agent c-semiring) of the individual action preferences, and the threshold for the system will be the product of individual agent thresholds.

This seems natural if the composition is viewed as synchronous execution of the actions. However, the intended interpretation is independent, concurrent execution. Thus they could be executed asynchronously in different orders. Perhaps there is an interesting parallel composition other than product to investigate.

In any event, we can now apply the logic and diagnostic methods developed in [23]. For example, consider a two Patrol bot system where one of the Bots has a faulty actuator that only moves the Bot every other try, but still uses energy. Then using thresholds that would ensure that non-faulty Bots don't run out of energy we find behaviors in which one Bot runs out of energy. The diagnostic preference will tell how much the system threshold must be raised, and the algorithm for finding suspects can help identify the faulty Bot. With the automata inspired structuring of preferences and thresholds, we can raise the energy reserve threshold for just the faulty Bot. In the current SA framework we would need to raise the caution for both Bots. More subtly, consider a two Patrol bot system with equivalent Bots where the Bots can only tell if the charging station is occupied by trying to enter. Thus there are behaviors in which one Bot is recharging and the second Bot is trying to enter and will run out of energy trying if the energy reserve is low. In this case we can identify the diagnostic preference, but now both Bots will be considered suspect since they are equivalent.

Limitations of Thresholds. Raising the threshold in the case of competition for the charging station will allow a bot to 'bang its head against the wall' longer and perhaps get access to the charging station. But this may not be the best solution. Perhaps thresholds can be used to identify the point at which things go wrong, and other methods developed to transform agent specifications.

In the Patrol Bot scenario we discovered, using Maude's reachability model checking, that if an obstacle is placed on the preferred y of a Bot, using the SCP described in Sect. 2.3 the Bot will go `N` or `S` when it reaches the obstacle, but then `S` or `N` back to where it started until it runs out of energy. Raising the caution level/threshold does not help. The problem seems to be conflating the concerns of achieving a goal (reaching the opposite side of the grid) with the concern of staying on track. Thus we propose investigating decomposing concerns, that is finding sub-concerns that can be composed to replace the single concern.

To address the cycling Patrol Bot problem we define the `val-patrol` c-semiring to be the join of two unit c-semirings and define the patrol preference for actions to be the semiring product of the preference for staying on track and the goal preference for reaching the other side. Define the 'staying on track' preference of a move to be .8 if the move goes closer to on track, stays on track or stays off track by at most 1 unit for at most 2 times; and .5 if the move goes off track by 2 or more units, or stays off for more than 2 times. Define the goal preference for a move to be .8 if the move goes closer to the goal, .5 if the move stays the same distance from the goal, and .3 if the move goes away from the goal. Now if the Bot, going `W`, has moved one unit `N` to avoid an obstacle, then its on track preferences for moves will be .8 for directions `W` or `S`, and its goal preferences will be .8 for direction W and .5 for direction S. Thus the patrol preferences will be $.8 \times .8 = .64$ for direction W and $.8 \times .5 = .40$ for direction S. Thus the Bot will prefer to move W and avoid cycling. With the new SCP search finds executions that succeed.

3.4 Fault Models

Recently, we have added basic generic fault models to the SA framework. Faults for each sensor and action have two parameters representing the probability of happening and the magnitude of the effect. Maude has a built in (pseudo) random number generator that the framework uses to sample probability distributions at execution time. This allows exploring the effects of faults by simply executing (with sufficiently high probability of fault). In addition we can use statistical model-checking to make semi-quantitative assessments of the effects of sensor or action faults.

One possibility for modeling faults in SCA would be to identify a small number of discrete effects that cover the range of possibilities (with high probability). The environment controls faults, so the relevant environment component(s) could assign lower preference to less likely fault levels. Then the SCA diagnostic methods could be used to explore effects of fault thresholds on the ability of a system to satisfy given properties. It is also interesting to consider SCA with probabilistic transitions.

4 Related Work

The *Constraint Semiring* algebraic structure for preferences was proposed in [4,6,7]. Further exploration of the compositionality of such structures appears in [15,20,22]. In [18] a generalization called Monoidal Soft Constraints is proposed to support more complex preference relations.

The paper [5] proposes a quantitative modal logic using elements of semirings as truth values; and the notion of threshold is used to limit the behavior of agents. A protocol for agents to adapt and cooperate by decomposing a goal specified in the logic leading to subgoals for agents to achieve.

The use of partially ordered knowledge to represent state and communication in the soft agent framework builds on partially ordered knowledge sharing for communication in disrupted environments [9,26,38,39], and the modeling of time draws from the Real-time Maude approach to modeling timed systems [35].

In [19] a mathematical system model for ensembles is presented. Similar to soft agents, the mathematical model treats both cyber and physical aspects of a system. A notion of fitness is defined that supports reasoning about level of satisfaction. In contrast to the soft-agent framework which provides an executable model, the system model for ensembles is denotational.

A closely related area is work on Collective Adaptive Systems (CAS) [24]. CAS consist of a large number of spatially distributed heterogeneous entities with decentralized control and varying degrees of complex autonomous behavior that may be competing for shared resources, even when collaborating to reach common goals. CARMA (Collective Adaptive Resource-sharing Markovian Agents) [30] is a language and tool set for modeling CAS.

Work on *Fault Diagnosis* is presented in [13,34,37]. In Fault Diagnosis, one tries to find out *whether or not* an error took place, and *which* error occurred. The focus of SA and SCA is to identify the *cause* of the error. A general framework for fault ascription in concurrent systems based on *counterfactuals* is presented in [16,17].

5 Conclusion and Future Directions

Rewriting logic provides a compact representation of transitions over complex states, and a common language for representing c-semirings, structure of agent and environment states and semantics of actions. The knowledge based approach to state representation makes it easy to introduce and reason about fault and threat models. Maude provides built in support for execution, search and model checking; and for modeling probabilistic transitions and associated tools for statistical model checking.

SCA provide a uniform notion of composition and a level of abstraction that focuses on the big picture and simplifies developing analysis algorithms. The extension to LTL is a promising step toward lifting the model compositional structure to compositional reasoning. SCA can naturally be extended to support more detailed descriptions by adding *memory cells* [21] that can store structured information about aspects of agent state, for example energy level or location.

Looking for the compositional structure underlying soft agent system specifications guided by Soft Component Automata has lead to a number of insights and ideas for improving the SA framework. In particular, turning the caution parameter into a preference threshold and making preferences explicit in the transitions allows us to take advantage of the diagnostics tools and extended LTL for formally specifying properties. Another insight is the role of conditions in rewrite rules in coordinating the composition of agent and environment models.

Work is ongoing to map SCA (more generally, Soft Constraint Automata with memory, and hence Reo models) to Maude to be able to execute system specifications, and analyse them using Maude's builtin search and model-checking. This will also provide a mechanism to experiment with extensions such as data structure definitions and probabilistic transitions and to help understand when environment preferences can be used as abstractions of probability distributions.

Future Directions. The comparison of the SA framework and SCA has lead to some new problems to study.

One problem is determining when actions/concerns and their preferences need to be decomposed further to determine the guilty party or parties and eliminate undesired behavior. This includes deciding which concern to decompose and how, so that in the resulting preference system, raising the threshold becomes a useful way of tuning the behavior.

Another issue to explore is synchronous vs asynchronous composition of preferences. Is there a composition operation that better reflects the nature of the asynchronous composition of agents in the SA framework that does not require making all the interleavings explicit?

Beyond using thresholds for diagnosis, it is also interesting to investigate methods to decompose system level properties or project them onto agents to identify faulty components or component interactions.

Finally, an intriguing possibility is to see how the coordination of agent systems imposed by the conditions in a rewrite rule can be represented, visualized, and reasoned about using concepts from Reo. This might be especially useful to a system designer to reason about security concerns, but also in cases where the interactions have more complex coordination constraints.

References

1. Arbab, F., Mavaddat, F.: Coordination through channel composition. In: Arbab, F., Talcott, C. (eds.) COORDINATION 2002. LNCS, vol. 2315, pp. 22–39. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46000-4_6
2. Arbab, F., Santini, F.: Preference and similarity-based behavioral discovery of services. In: ter Beek, M.H., Lohmann, N. (eds.) WS-FM 2012. LNCS, vol. 7843, pp. 118–133. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38230-7_8
3. Bae, K., Meseguer, J.: The linear temporal logic of rewriting Maude model checker. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 208–225. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16310-4_14
4. Bistarelli, S.: Semirings for Soft Constraint Solving and Programming. LNCS, vol. 2962. Springer, Heidelberg (2004). https://doi.org/10.1007/b95712
5. Bistarelli, S., Martinelli, F., Matteucci, I., Santini, F.: A formal and run-time framework for the adaptation of local behaviours to match a global property. In: Kouchnarenko, O., Khosravi, R. (eds.) FACS 2016. LNCS, vol. 10231, pp. 134–152. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57666-4_9
6. Bistarelli, S., Montanari, U., Rossi, F.: Constraint solving over semirings. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), pp. 624–630 (1995)
7. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. J. ACM **44**(2), 201–236 (1997)
8. Why BNSF railway is using drones to inspect thousands of miles of rail lines. http://fortune.com/2015/05/29/bnsf-drone-program/. Accessed 11 Mar 2016
9. Choi, J.S., McCarthy, T., Yadav, M., Kim, M., Talcott, C., Gressier-Soudan, E.: Application patterns for cyber-physical systems. In: IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications, pp. 52–59 (2013)
10. Choi, J.-S., McCarthy, T., Kim, M., Stehr, M.-O.: Adaptive wireless networks as an example of declarative fractionated systems. In: Stojmenovic, I., Cheng, Z., Guo, S. (eds.) MindCare 2014. LNICST, vol. 131, pp. 549–563. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11569-6_43
11. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude: A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1

12. Das, J., Cross, G., Qu, C., Makineni, A., Tokekar, P., Mulgaonkar, Y., Kumar, V.: Devices, systems, and methods for automated monitoring enabling precision agriculture. In: IEEE International Conference on Automation Science and Engineering (2015)
13. Debouk, R., Lafortune, S., Teneketzis, D.: Coordinated decentralized protocols for failure diagnosis of discrete event systems. Discret. Event Dyn. Syst. **10**(1–2), 33–86 (2000)
14. Autonomous Taxi Drones. https://www.forbes.com/sites/parmyolson/2017/02/14/dubai-autonomous-taxi-drones-ehang/#54543d934702. Accessed 11 Mar 2017
15. Gadducci, F., Hölzl, M., Monreale, G.V., Wirsing, M.: Soft constraints for lexicographic orders. In: Castro, F., Gelbukh, A., González, M. (eds.) MICAI 2013. LNCS (LNAI), vol. 8265, pp. 68–79. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45114-0_6
16. Goessler, G., Astefanoaei, L.: Blaming in component-based real-time systems. In: International Conference on Embedded Software, EMSOFT 2014, pp. 7:1–7:10 (2014)
17. Gössler, G., Stefani, J.-B.: Fault ascription in concurrent systems. In: Ganty, P., Loreti, M. (eds.) TGC 2015. LNCS, vol. 9533, pp. 79–94. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-28766-9_6
18. Hölzl, M., Meier, M., Wirsing, M.: Which soft constraints do you prefer? In: Seventh International Workshop on Rewriting Logic and Its Applications (WRLA'2008). Electronic Notes in Theoretical Computer Science. Elsevier (2008)
19. Hölzl, M., Wirsing, M.: Towards a system model for ensembles. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 241–261. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24933-4_12
20. Hölzl, M.M., Meier, M., Wirsing, M.: Which soft constraints do you prefer? Electr. Notes Theoret. Comput. Sci. **238**(3), 189–205 (2009)
21. Jongmans, S.T., Kappé, T., Arbab, F.: Constraint automata with memory cells and their composition. Sci. Comput. Program. **146**, 50–86 (2017)
22. Kappé, T., Arbab, F., Talcott, C.L.: A compositional framework for preference-aware agents. In: Proceedings of Workshop on Verification and Validation of Cyber-Physical Systems (V2CPS), pp. 21–35 (2016)
23. Kappé, T., Arbab, F., Talcott, C.: A component-oriented framework for autonomous agents. In: Proença, J., Lumpe, M. (eds.) FACS 2017. LNCS, vol. 10487, pp. 20–38. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68034-7_2
24. Kernbach, S., Schmickl, T., Timmis, J.: Collective adaptive systems: challenges beyond evolvability. In: Fundamentals of Collective Adaptive Systems. European Commission (2009)
25. Kim, M., Stehr, M.-O., Talcott, C.: A distributed logic for networked cyber-physical systems. In: Arbab, F., Sirjani, M. (eds.) FSEN 2011. LNCS, vol. 7141, pp. 190–205. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29320-7_13
26. Kim, M., Stehr, M.O., Talcott, C.L.: A distributed logic for networked cyber-physical systems. Sci. Comput. Program. **78**(12), 2453–2467 (2013)
27. Vijay Kumar Lab. http://www.kumarrobotics.org/. Accessed 11 Mar 2016
28. Robots that Fly and Cooperate (2015). TED talk: https://www.ted.com/talks/vijay_kumar_robots_that_fly_and_cooperate?language=en. Accessed 07 Mar 2016
29. Liquid Robotics. http://liquidr.com. Accessed 11 Mar 2016

30. Loreti, M., Hillston, J.: Modelling and analysis of collective adaptive systems with CARMA and its tools. In: Bernardo, M., De Nicola, R., Hillston, J. (eds.) SFM 2016. LNCS, vol. 9700, pp. 83–119. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-34096-8_4
31. Mason, I.A., Talcott, C.L.: IOP: the InterOperability platform & IMaude: an interactive extension of Maude. In: Fifth International Workshop on Rewriting Logic and Its Applications (WRLA'2004). Electronic Notes in Theoretical Computer Science. Elsevier (2004)
32. The Maude System. http://maude.cs.uiuc.edu. Accessed 15 Nov 2014
33. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoret. Comput. Sci. **96**(1), 73–155 (1992)
34. Neidig, J., Lunze, J.: Decentralised diagnosis of automata networks. In: IFAC Proceedings, vol. 38, no. 1, pp. 400–405 (2005)
35. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of real-time Maude. High.-Order Symb. Comput. **20**(1–2), 161–196 (2007)
36. Invetory Robotics. http://www.pinc.com/inventory-robotics-cycle-counting-drones. Accessed 11 Apr 2017
37. Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., Teneketzis, D.: Failure diagnosis using discrete-event models. IEEE Trans. Control Syst. Technol. **4**(2), 105–124 (1996)
38. Stehr, M.-O., Kim, M., Talcott, C.: Partially ordered knowledge sharing and fractionated systems in the context of other models for distributed computing. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. LNCS, vol. 8373, pp. 402–433. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54624-2_20
39. Stehr, M.-O., Talcott, C., Rushby, J., Lincoln, P., Kim, M., Cheung, S., Poggio, A.: Fractionated software for networked cyber-physical systems: research directions and long-term vision. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 110–143. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24933-4_7
40. Talcott, C., Arbab, F., Yadav, M.: Soft agents: exploring soft constraints to model robust adaptive distributed cyber-physical agent systems. In: De Nicola, R., Hennicker, R. (eds.) Software, Services, and Systems. LNCS, vol. 8950, pp. 273–290. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15545-6_18
41. Talcott, C., Nigam, V., Arbab, F., Kappé, T.: Formal specification and analysis of robust adaptive distributed cyber-physical systems. In: Bernardo, M., De Nicola, R., Hillston, J. (eds.) SFM 2016. LNCS, vol. 9700, pp. 1–35. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-34096-8_1

Argumentation as Exogenous Coordination

Leendert van der Torre[1](✉), Tjitze Rienstra[2], and Dov Gabbay[1,3,4]

[1] University of Luxembourg, Esch-Sur-Alzette, Luxembourg
leon.vandertorre@uni.lu
[2] Institute for Web Science and Technologies, University of Koblenz-Landau, Koblenz, Germany
tjitze@gmail.com
[3] Department of Informatics, King's College London, London, UK
dov.gabbay@kcl.ac.uk
[4] Department of Computer Science, Bar Ilan University, Tel Aviv, Israel

Abstract. Formal argumentation is one of the most popular approaches in modern logic and reasoning. The theory of abstract argumentation introduced by Dung in 1995 has shifted the focus from the internal structure of arguments to relations among arguments, and temporal dynamics for abstract argumentation was proposed by Barringer, Gabbay and Woods in 2005. In this tradition, we see arguments as reasoning processes, and the interaction among them as a coordination process. We argue that abstract argumentation can adopt ideas and techniques from formal theories of coordination, and as an example we propose a model of sequential abstract argumentation loosely inspired by Reo's model of exogenous coordination. We show how the argumentation model can represent the temporal dynamics of the liar paradox and predator-prey like behaviour.

1 Introduction

The theory of abstract argumentation introduced by Dung in 1995 [14] started a new stage in the development of formal argumentation theory. In his model, the acceptance or rejection of an argument depends on the relation between the argument with other arguments, and the acceptance status of these other arguments. In contrast, traditionally argumentation was based on a formal analysis of the logical structure of arguments, and whether an argument is accepted or rejected depended only on the argument itself, not on the other arguments. In other words, in Dung's model it is no longer sufficient to point at deficiencies in an argument to reject it, but one is required to phrase the criticism itself as an argument, such that these critical arguments themselves are open for criticism as

The first author has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skodowska-Curie grant agreement No 690974 for the project MIREL: MIning and REasoning with Legal texts.

F. de Boer et al. (Eds.): Arbab Festschrift, LNCS 10865, pp. 208–223, 2018.
https://doi.org/10.1007/978-3-319-90089-6_14

well. In abstract argumentation, we say that when an argument attacks another argument, the attacking argument itself can be attacked by a third argument. In such a case, the argument originally attacked is *defended* by the third argument, and consequently the first argument may be *reinstated*.

Formally, abstract argumentation is a graph based reasoning formalism generalising the notion of stable sets in directed graphs. As discussed in the handbook series on formal argumentation, of which the first volume appears in 2018 [8], Dung's theory constitutes a turning point for the modern stage of formal argumentation theory, much similar to the introduction of possible worlds semantics for the theory of modality. This means that nothing could remain the same as before 1995—it should be a focal point of reference for any study of argumentation, especially if the study is critical about Dung's theory. However, in modal logic, the introduction of the possible worlds semantics has led to a complete paradigm shift, both in tools and new subjects of studies, whereas this is still not fully true for what is going on in formal argumentation theory. In this paper our aim is to inspire new tools and studies for formal argumentation based on models of formal coordination, in particular the exogenous coordination language Reo [2–4].

It is not very difficult to relate abstract argumentation to the data-flow coordination language Reo, because there are various superficial similarities and differences between the two approaches. Concerning similarities, as we explain in more detail later, both are based on a graph based representation, both use graph colouring to give compositional semantics to the graphs, and consequently both can be seen as instances of causal or explanatory non-monotonic reasoning (see [10] for a modern introduction). In particular, graph colouring is used in argumentation for the key properties of admissibility and directionality, and in Reo to deal with the context sensitive behaviour of lossy channels. Moreover, argumentation graphs have been given temporal dynamics [9], and they have been extended to input/output graphs [7,15] that reflect the flow or directionality of reasoning in logic and argumentation [20,21]. Concerning the superficial differences, Reo has many aspects without a directly corresponding counterpart in abstract argumentation, such as stream semantics, or buffers representing memory. Likewise, abstract argumentation has aspects which do not seem to have a direct correspondence in coordination, such as complementary theories of structured argumentation.

Besides the superficial similarities and differences between the two approaches, we believe that we can define a deeper similarity between them, based on the concept of exogenous coordination. Arbab [4] defines exogenous coordination as follows.

> "Locus of coordination refers to where coordination activity takes place, classifying coordination models as endogenous or exogenous. Endogenous models, such as Linda, provide primitives that must be incorporated within a computation for its coordination. In contrast, exogenous models, such as Manifold and Reo, provide primitives that support coordination of entities from without. In applications that use exogenous models, primitives that affect the coordination of each module are outside the module itself.

> Endogenous models are sometimes more natural for a given application. However, they generally lead to an intermixing of coordination primitives with computation code, which entangles the semantics of computation with coordination protocols. This intermixing tends to scatter communication/coordination primitives throughout the source code, making the cooperation model and the coordination protocol of an application nebulous and implicit: generally, there is no piece of source code identifiable as the cooperation model or the coordination protocol of an application, that can be designed, developed, debugged, maintained, and reused, in isolation from the rest of the application code. ...
> On the other hand, exogenous models encourage development of coordination modules separately and independently of the computation modules they are supposed to coordinate. Consequently, the result of the substantial effort invested in the design and development of the coordination component of an application can manifest itself as tangible "pure coordinator modules" which are easier to understand, and can also be reused in other applications." [4]

In this paper we see arguments as reasoning processes, and we characterise the interaction among such abstract argument processes as a way of coordinating arguments. In other words, we rephrase the core idea of interaction among arguments reflected by the graph based framework and language introduced by Dung in 1995 as the introduction of exogenous coordination in logic and reasoning. This reflects a separation of concerns between the logical structure of an argument and its acceptability, and facilitates the reuse of arguments as well as the reuse of argumentation frameworks.

The deeper relation between the two approaches suggests that abstract argumentation can learn from more general models of coordination, and in particular of exogenous models of coordination. Indeed, we believe it is straightforward to incorporate ideas from Reo into formal argumentation, and to find corresponding notions in informal argumentation. For example, argumentation memory is clearly present in the argumentation of people, companies, organisations, political parties, and other kinds of socially constructed entities, for example due to bounded rationality. Also in scientific argumentation only a paradigm shift leads to the rejection of conventional wisdom. In the media, daily news articles change the opinions and the arguments of the people, and reveals a dark side of spreading alternative facts and fake news. We do not claim any authority on organisational theory, philosophy of science or media sciences, but it seems clear to us that a formal study of more temporal abstract argumentation model incorporating streams of data and arguments is both natural and useful.

As an example, and a first step to bring the two approaches closer together, this paper proposes a model of sequential abstract argumentation inspired by Reo's model of exogenous coordination. This model builds on our earlier work. Multi-sorted argumentation [26] partitions the set of arguments, for example in epistemic and goal arguments, and applies different kinds of semantics to each block in the partitioning. Input/output argumentation [7] is a model of compositionality for abstract argumentation, that makes explicit how the semantics of the individual blocks can be composed into the semantics of the whole graph. In multi-agent argumentation [5], the composition of the acceptance semantics reflects a game theoretic equilibrium between the individual acceptance functions. Traditional game theoretic semantics assumes that the agents accept their arguments at the same time, just like agents in a prisonner's dilemma choose their decisions independently of each other.

Our model of temporal dynamics [9] in this paper mimics the steps of a dialogue. Like in extensive game models, we proceed step by step. The agents in a dialogue listen to the arguments the other agents accept, and decide which arguments to accept based on this information.

Given the nature of this special volume, we assume that the readers are familiar with the challenges of coordination, the concept of exogeneous coordination, the Reo coordination language, and its semantics. Moreover, we assume that not all readers are familiar with abstract argumentation, so we repeat the basic concepts and ideas.

The layout of this paper is as follows. In Sect. 2 give an overview of abstract argumentation, including graph colouring and input/output argumentation, in Sect. 3 we introduce our variant of sequential argumentation and we discuss the liar paradox, and in Sect. 4 we consider temporal dynamics and predator-prey like behavior [9].

2 Abstract Argumentation Semantics

In this section we consider abstract argumentation semantics, and in the next section we introduce sequence semantics. We first recall Dung's abstract argumentation semantics, the commonly adopted generalisation to three valued labelings, and the generalisation to input/output argumentation.

2.1 Abstract Semantics

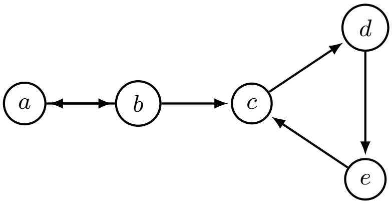
For completeness and reference below we briefly summarise Dung's abstract argumentation semantics. An argumentation framework is a directed graph whose nodes $\mathscr{A}$ are called arguments and whose edges $\mathscr{R}$ represent attack among the arguments, a kind of asymmetric inconsistency. A set $B \subseteq \mathscr{A}$ is *conflict-free* if and only if there exist no arguments a_1 and a_2 in B such that $(a_1, a_2) \in \mathscr{R}$. Argument $a \in \mathscr{A}$ is *defended* by a set $B \subseteq \mathscr{A}$ (also called a_1 is *acceptable* with respect to B) if and only if for all $a_2 \in \mathscr{A}$, if $(a_2, a_1) \in \mathscr{R}$, then there exists $a_3 \in B$ such that $(a_3, a_2) \in \mathscr{R}$. We say that a set $B \subseteq \mathscr{A}$ is *admissible*, if and only if it is *conflict-free* and defends all of its members. Based on the notion of admissible sets, Dung defines various kinds of sets of acceptable arguments called extensions. Formally, we have the following definition.

Definition 1 (Dung semantics). *Let $\mathscr{F} = (\mathscr{A}, \mathscr{R})$ be a graph called an argumentation framework, and $B \subseteq \mathscr{A}$ a set of arguments.*

- *B is* conflict-free *if and only if* $\nexists a_1, a_2 \in B$, *s.t.* $(a_1, a_2) \in \mathscr{R}$.
- *An argument* $a_1 \in \mathscr{A}$ *is defended by B (equivalently* a_1 *is* acceptable *w.r.t. B), if and only if* $\forall (a_2, a_1) \in \mathscr{R}$, $\exists \gamma \in B$, *s.t.* $(a_3, a_2) \in \mathscr{R}$.
- *B is* admissible *if and only if B is conflict-free, and each argument in B is defended by B.*
- *B is a* complete *extension if and only if B is admissible and each argument in* $\mathscr{A}$ *that is defended by B is in B.*
- *B is a* preferred *extension if and only if B is a maximal (w.r.t. set-inclusion) complete extension.*
- *B is a* grounded *extension if and only if B is the minimal (w.r.t. set-inclusion) complete extension.*
- *B is a* stable *extension if and only if B is conflict-free, and* $\forall a_1 \in \mathscr{A} \setminus B$, $\exists a_2 \in B$ *s.t.* $(a_2, a_1) \in \mathscr{R}$.

We use $sem \in \{cmp, prf, grd, stb\}$ to denote complete, preferred, grounded, or stable semantics, respectively. A set of argument extensions of $\mathscr{F} = (\mathscr{A}, \mathscr{R})$ is denoted as $sem(\mathscr{F})$.

Example 1. Consider the argumentation framework visualized below. The complete extensions are $E_1 = \emptyset$, $E_2 = \{a\}$, and $E_3 = \{b, d\}$. The former is the unique grounded extension and the latter two are the preferred extensions, and only E_3 is a stable extension.



Dung's graph based theory has been further refined using abstract rules and assumptions, and extensions of the graph based representation have been studied as abstract dialectical frameworks. Argumentation as inference developed by Dung has been complemented by argumentation as dialogue, based on argumentation semantics as formal discussion, and argumentation schemes. In addition, computational problems have been studied, including their complexity, and implementations have been built. Formal analysis is based on a principle based approach to formal argumentation, including the use of rationality postulates to evaluate argumentation semantics. The relations between formal argumentation and other areas of formal reasoning, in particular logic, has been studied. We refer to the first volume of the Handbook of Formal Argumentation [8] for further details.

2.2 Labelling Semantics

Input/output argumentation uses the labelling-based approach to the definition of argumentation semantics. A labelling assigns to each argument of an argumentation framework a label taken from a predefined set Λ. For technical reasons, we define labellings both for argumentation frameworks and for arbitrary sets of arguments.

Definition 2 (Labeling). *Let $\Lambda = \{\mathtt{in}, \mathtt{out}, \mathtt{undec}\}$ be a set of labels. Given a set of arguments B, a* labelling *of B is a total function $Lab : B \longrightarrow \Lambda$. The set of all* labellings *of B is denoted as $\mathfrak{L}_B$. Given an argumentation framework $\mathscr{F} = (\mathscr{A}, \mathscr{R})$, a* labelling *of $\mathscr{F}$ is a labelling of $\mathscr{A}$. The set of all* labellings *of $\mathscr{F}$ is denoted as $\mathfrak{L}(\mathscr{F})$. For a labelling Lab of B, the restriction of Lab to a set of arguments $B' \subseteq B$, denoted as $Lab{\downarrow_{B'}}$, is defined as $Lab \cap (B' \times \Lambda)$.*

The label `in` means that the argument is accepted, the label `out` means that the argument is rejected, and the label `undec` means that the status of the argument is undecided. Given a labelling Lab, we write $\mathtt{in}(Lab)$ for $\{a \mid Lab(a) = \mathtt{in}\}$, $\mathtt{out}(Lab)$ for $\{a \mid Lab(a) = \mathtt{out}\}$ and $\mathtt{undec}(Lab)$ for $\{a \mid Lab(a) = \mathtt{undec}\}$.

A labelling-based semantics prescribes a set of labellings for each argumentation framework.

Definition 3 (Labeling semantics). *Given an argumentation framework $\mathscr{F} = (\mathscr{A}, \mathscr{R})$, a labelling-based semantics* **S** *associates with $\mathscr{F}$ a subset of $\mathfrak{L}(\mathscr{F})$, denoted as $\mathbf{L_S}(\mathscr{F})$.*

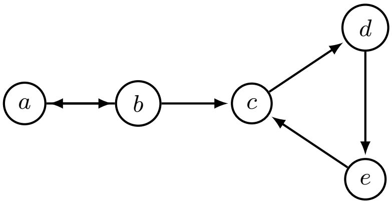
Though labelings are more general than Dung semantics, we now apply a common trick in formal argumentation: we reduce the more general notion to Dung semantics. (Just like semantics of extensions of Turing machines are reduced to Turing machines) In particular, for every Dung semantics there is a labeling based version defined by the following translation from extensions to labelings:

Definition 4 (Dung2labeling). *Given an argumentation framework $\mathscr{F} = (\mathscr{A}, \mathscr{R})$ an extension $B \subseteq \mathscr{A}$ translates to a labelling $Lab \in \mathfrak{L}_{\mathscr{A}}$ iff*

- *$Lab(a) = \mathtt{in}$, if $a \in B$,*
- *$Lab(a) = \mathtt{out}$, if $a \notin B$ and there is a $b \in B$ such that $(b, a) \in \mathscr{R}$,*
- *$Lab(a) = \mathtt{undec}$, otherwise.*

In particular, we use $sem \in \{cmp, prf, grd, stb\}$ to denote complete, preferred, grounded, or stable labeling semantics, defined in this way in terms of the corresponding Dung semantics.

Example 2 (Continued from Example 1). Reconsider the argumentation framework visualized below. The complete labelings are
$L_1 = \{(a, \mathtt{undec}), (b, \mathtt{undec}), (c, \mathtt{undec}), (d, \mathtt{undec}), (e, \mathtt{undec})\}$,
$L_2 = \{(a, \mathtt{in}), (b, \mathtt{out}), (c, \mathtt{undec}), (d, \mathtt{undec}), (e, \mathtt{undec})\}$, and
$L_3 = \{(a, \mathtt{out}), (b, \mathtt{in}), (c, \mathtt{out}), (d, \mathtt{in}), (e, \mathtt{out})\}$.
The former is the unique grounded labeling and the latter two are the preferred labelings, and only L_3 is a stable labeling.



2.3 Baroni *et al.*'s Notion of Local Function

Local functions define semantics for a part of the argumentation framework, called a sub-framework. In this section we repeat some basic concepts regarding local functions from Baroni *et al.*. We refer to their paper [7] for further explanations and examples. Similar notions are also defined by Liao [18,19]. Local functions are more general than Dung semantics in the sense that they give extensions to a graph together with an input. If the input is empty, a local function coincides with a Dung semantics.

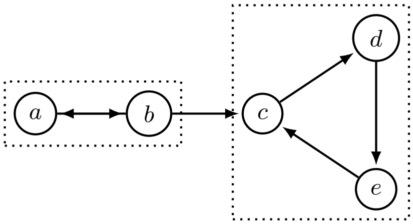
We start with the input of a subframework. Intuitively, given an argumentation framework $\mathscr{F} = (\mathscr{A}, \mathscr{R})$ and a subset B of its arguments, the elements affecting $\mathscr{F}{\downarrow_B}$, which is $(\{a \in \mathscr{A} \mid a \in B\}, \{(a_1, a_2) \in \mathscr{R} \mid a_1, a_2 \in B\})$, include the arguments attacking B from the outside, called *input* arguments, and the attack relation from the input arguments to B, called *conditioning relation.*

Definition 5 (Input). *Given $\mathscr{F} = (\mathscr{A}, \mathscr{R})$ and a set $B \subseteq \mathscr{A}$, the* input *of B, denoted as B^{inp}, is the set $\{a_2 \in \mathscr{A} \setminus B \mid \exists a_1 \in B, (a_2, a_1) \in \mathscr{R}\}$, the* conditioning relation *of B, denoted as B^R, is defined as $\mathscr{R} \cap (B^{inp} \times B)$.*

An *argumentation framework with input* consists of an argumentation framework $\mathscr{F} = (\mathscr{A}, \mathscr{R})$ (playing the role of a partial argumentation framework), a set of external input arguments $\mathscr{I}$, a labelling $L_{\mathscr{I}}$ assigned to them and an attack relation $R_{\mathscr{I}}$ from $\mathscr{I}$ to $\mathscr{A}$. A *local function* which, given an argumentation framework with input, returns a corresponding set of labellings of $\mathscr{F}$.

Definition 6 (Framework with input). *An* argumentation framework with input *is a tuple $(\mathscr{F}, \mathscr{I}, L_{\mathscr{I}}, R_{\mathscr{I}})$, including an argumentation framework $\mathscr{F} = (\mathscr{A}, \mathscr{R})$, a set of arguments $\mathscr{I}$ such that $\mathscr{I} \cap \mathscr{A} = \emptyset$, a labelling $L_{\mathscr{I}} \in \mathfrak{L}_{\mathscr{I}}$ and a relation $R_{\mathscr{I}} \subseteq \mathscr{I} \times \mathscr{A}$. A* local function *assigns to any argumentation framework with input a (possibly empty) set of labellings of $\mathscr{F}$, i.e. $f(F, \mathscr{I}, L_{\mathscr{I}}, R_{\mathscr{I}}) \in 2^{\mathfrak{L}(F)}$.*

Example 3 (Continued from Example 2). Reconsider the argumentation framework in Example 1 and 2, together with the partitioning visualised below. The block $P_1 = \{a, b\}$ has an empty input, and block $P_2 = \{c, d, e\}$ has input$\{b\}$ with conditioning relation $\{(b, c)\}$.



For any semantics, a "sensible" local function, called *canonical local function*, is the one that describes the labellings of the so-called standard argumentation frameworks.

Definition 7 (Standard argumentation framework). *Given an argumentation framework with input $(\mathscr{F}, \mathscr{I}, L_{\mathscr{I}}, R_{\mathscr{I}})$, the standard argumentation framework w.r.t. $(\mathscr{F}, \mathscr{I}, L_{\mathscr{I}}, R_{\mathscr{I}})$ is defined as $\mathscr{F}' = (\mathscr{A} \cup \mathscr{I}', \mathscr{R} \cup R'_{\mathscr{I}})$, where $\mathscr{I}' = \mathscr{I} \cup \{a' \mid a \in \texttt{out}(L_{\mathscr{I}})\}$ and $R'_{\mathscr{I}} = R_{\mathscr{I}} \cup \{(a', a) \mid a \in \texttt{out}(L_{\mathscr{I}})\} \cup \{(a, a) \mid a \in \texttt{undec}(L_{\mathscr{I}})\}$.*

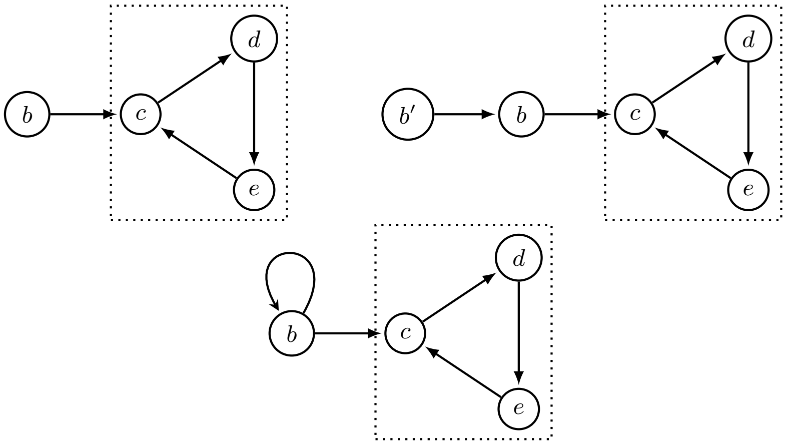
Roughly speaking, the standard argumentation framework puts $\mathscr{F}$ under the influence of $(\mathscr{I}, L_{\mathscr{I}}, R_{\mathscr{I}})$, by adding $\mathscr{I}$ to $\mathscr{A}$ and $R_{\mathscr{I}}$ to $\mathscr{R}$, and by enforcing the label $L_{\mathscr{I}}$ for the arguments of $\mathscr{I}$ in this way:

- for each argument $a \in \mathscr{I}$ such that $L_{\mathscr{I}}(a) = \texttt{out}$, an unattacked argument a' is included which attacks a, in order to get A labelled `out` by all labellings of $\mathscr{F}'$;
- for each argument $a \in \mathscr{I}$ such that $L_{\mathscr{I}}(a) = \texttt{undec}$, a self-attack is added to a in order to get it labelled `undec` by all labellings of F';
- each argument $a \in \mathscr{I}$ such that $L_{\mathscr{I}}(a) = \texttt{in}$ is left unattacked, so that it is labelled `in` by all labellings of $\mathscr{F}'$.

Definition 8 (Labeling2localfunction).
Given a semantics $\mathbf{S}$*, the* canonical local function *of* $\mathbf{S}$ *(also called local function of* $\mathbf{S}$*) is defined as $f_{\mathbf{S}}(\mathscr{F}, \mathscr{I}, L_{\mathscr{I}}, R_{\mathscr{I}}) = \{Lab{\downarrow_{\mathscr{A}}} \mid Lab \in \mathbf{L}_{\mathbf{S}}(\mathscr{F}')\}$, where $\mathscr{F} = (\mathscr{A}, \mathscr{R})$ and $\mathscr{F}'$ is the standard argumentation framework w.r.t. $(\mathscr{F}, \mathscr{I}, L_{\mathscr{I}}, R_{\mathscr{I}})$.*

Moreover, we use $sem \in \{cmp, prf, grd, stb\}$ to denote complete, preferred, grounded, or stable local functions, defined in this way in terms of the corresponding labeling semantics.

Example 4 (Continued from Example 3). Reconsider the argumentation framework in Examples 1–3, together with the block $P_2 = \{c, d, e\}$ with input $\{b\}$ and conditioning relation $\{(b, c)\}$. The argumentation framework with input can make argument b either `in`, `out` or `undec`, which leads to the following three standard argumentation frameworks with respect to the argumentation frameworks with input.



We refer to the paper of Baroni et al. [7,26] for further discussion.

3 Sequential Abstract Argumentation

One requirement for a dynamic semantics is to be able to represent the liar paradox: If "this sentence is false" is true, then the sentence is false, but if the sentence states that it is false, and it is false, then it must be true. It is related to Epimenides paradox, Epimenides, a Cretan, said that "All Cretans are liars," and other paradoxes such as Russell's paradox. In a dynamic semantics, the truth value of the sentences toggles between true and false, and there is consequently no fixed point. In this section we show how our sequential semantics can mimic this behaviour.

Multi-agent argumentation considers a generic argumentation framework $AF = (\mathscr{A}, \mathscr{R})$ together with an arbitrary partition of $\mathscr{A}$, i.e. a set $\{P_1, \ldots, P_n\}$ such that $\forall i \in \{1, \ldots, n\}$ $P_i \subseteq \mathscr{A}$ and $P_i \neq \emptyset$, $\bigcup_{i=1\ldots n} P_i = \mathscr{A}$ and $P_i \cap P_j = \emptyset$ for $i \neq j$. Such a partition identifies the restricted argumentation frameworks $AF{\downarrow_{P_1}}, \ldots, AF{\downarrow_{P_n}}$, that affect each other with the relevant input arguments and conditioning relations as stated in Definition 5.

A multi-agent argumentation framework extends an argumentation framework with a partitioning.

Definition 9 (Multi-agent argumentation framework). *A multi-agent argumentation framework is a tuple* $\mathscr{F} = (\mathscr{A}, \mathscr{R}, \mathscr{P})$ *extending an argumentation framework* $(\mathscr{A}, \mathscr{R})$ *with a partition* $\mathscr{P} = \{P_1, \ldots, P_n\}$ *of* $\mathscr{A}$*.*

The semantics of a multi-agent argumentation framework in Examples 1–4 can be based on first computing the extension of block P_1, and thereafter the extension of block P_2 using the extension of block P_1 as input. This is the basis of a well known recursive algorithm. Multi-agent argumentation raises the question what to do when the blocks attack each other? In that case, a simple recursive algorithm does not suffice. Game theory suggests two approaches:

- **Nash equilibrium.** In case of cycles among agents, the semantics can be based on a game-theoretic equilibrium, such as for example Nash equilibria. This approach is followed by Arisaka et al. [5]. At one moment in time, the output of the agents must be identical to the input of the other agents. For example, in a prisonner's dilemma, each agent has to make a decision at the same moment without any coordination, and game theory defines states where the strategies of the agents are in a stable equilibrium.
- **Dialogue.** Extensive games such as dialogues are based on the idea that agent act one after the other, basing their actions on the observed actions of other agents. Sequential argumentation as we consider in this paper is based on local functions, together with the idea that the output of each framework is used as input for the next step in the sequence.

A sequence semantics prescribes a set of sequences of labellings for each argumentation framework. The sequence of extensions reflects a kind of dialogue between the blocks of the partitioning.

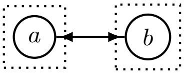
Definition 10 (Sequence semantics). *Given an argumentation framework* $\mathscr{F} = (\mathscr{A}, \mathscr{R})$*, a labelling-based sequence semantics* $\mathbf{S}$ *associates with* $\mathscr{F}$ *a set of sequences of* $\mathfrak{L}(\mathscr{F})$*, denoted as* $\mathbf{L_S}(\mathscr{F})$*.*

We use a Dung semantics to define a labeling semantics, and a labeling semantics to define an input/output semantics. Now we use an input/output semantics to define a sequence semantics. We assume that every labeling of the sequence is conflict free, though also stronger conditions may be considered. For example, one may require that every labeling of the sequence is an admissible set of F, or even a complete labeling.

Definition 11 (localfunction2sequence). *Consider a local function* f*. The sequence semantics of a framework* F *is a sequence of conflict free labelings of* F*, such that except for the first element of the sequence, every extension is computed using the local function* f *with the previous labeling of the sequence as the input.*

Again, we use $sem \in \{cmp, prf, grd, stb\}$ to denote complete, preferred, grounded, or stable sequence semantics, defined in this way in terms of the corresponding local functions.

Example 5. Consider a framework consisting of two arguments a and b attacking each other, and each argument originating from a different agent.



Since there are no cycles within a block of the partitioning, a labelling is completely determined by the input and the complete, preferred, grounded and stable labellings coincide. Three complete, preferred, grounded and stable sequences are:

$\langle\{(a,\texttt{undec}),(b,\texttt{undec})\},\{(a,\texttt{undec}),(b,\texttt{undec})\},\ldots\rangle$,
$\langle\{(a,\texttt{in}),(b,\texttt{out})\},\{(a,\texttt{in}),(b,\texttt{out})\},\ldots\rangle$,
$\langle\{(a,\texttt{out}),(b,\texttt{in})\},\{(a,\texttt{out}),(b,\texttt{in})\},\ldots\rangle$,
We can also have cyclic behaviour:
$\langle\{(a,\texttt{undec}),(b,\texttt{in})\},\{(a,\texttt{out}),(b,\texttt{undec})\},\{(a,\texttt{undec}),(b,\texttt{in})\},\ldots\rangle$,
$\langle\{(a,\texttt{out}),(b,\texttt{out})\},\{(a,\texttt{in}),(b,\texttt{in})\},\{(a,\texttt{out}),(b,\texttt{out})\},\ldots\rangle$,

If we represent the above sequences as sequences of extensions (an argument is in the extension iff it is labeled in) then we obtain the characteristic sequence of a liar paradox, which shows that
$\langle\{b\},\emptyset,\{b\},\emptyset,\ldots\rangle$
The cyclic behaviour in the previous example occurs when the initial labelling of the sequence is itself not an admissible labelling. The following propositions show that this is no coincidence.

Proposition 1. *If a labelling in a sequence is a preferred labelling, then it is a fixed point: all following labelings in the sequence will be the same.*

Proposition 2. *If a labelling in a sequence is a complete labelling, then all following labellings will be refinements, where L_1 refines L_2, written as $L_1 \sqsubseteq L_2$, iff $in(L_1) \subseteq in(L_2)$.*

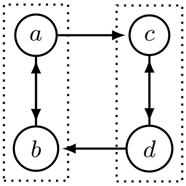
4 Predator-Prey Models

Barringer et al. [9] generalise argumentation frameworks in several directions. Following various other work in formal argumentation, they allow also support relations between arguments, they allow for varying strengths of attack and support, and such strengths of attacks or support are themselves subject to attack or support. They also introduce two new ideas. First, they allow for the strengths of attack or support to be time dependent, enabling them to model the phenomenon of "Let's lie low and wait for the argument to blow away". Secondly, they examine loop-resolution in argumentation networks, and explores similarities between such loops and predator-prey models in mathematical biology.

A requirement for temporal dynamics is to mimic the predator-prey behaviour. The predator-prey equations, also known as the Lotka-Volterra equations, are a pair of first-order, nonlinear, differential equations frequently used to describe the dynamics of biological systems in which two species interact, one as a predator and the other as prey. The populations change through time according to the pair of equations. The Lotka-Volterra system of equations is an example of a Kolmogorov model. In abstract argumentation, the requirement is to have two arguments that are accepted, then rejected, then accepted and so on.

The following example illustrates that such predator-prey behaviour can also be mimicked in our sequence semantics, without introducing numbers, support relations, attacks on attacks and so on.

Example 6. Consider a framework consisting of four arguments a, b, c and d, where the first two arguments belong to the first agent, and the latter two arguments belong to the second agent. Intuitively, the first agent can choose between accepting a or b (or none), and the second agent can choose between accepting c or d. However, these decisions are interdependent. When the first agent chooses a, the second agent no longer can choose c, and when the second agent chooses d, the first agent can no longer choose b.



Since there are loops in the argumentation frameworks of the agents, the four semantics no longer coincide. We consider the grounded semantics. In this case, given a labeling of the sequence, the following label is completely defined. If we start with a complete labeling, then all elements of the sequence are identical:

$\langle\{(a, \texttt{undec}), (b, \texttt{undec}), (c, \texttt{undec}), (d, \texttt{undec})\}, \ldots\rangle$,
$\langle\{(a, \texttt{in}), (b, \texttt{out}), (c, \texttt{out}), (d, \texttt{in})\}, \ldots\rangle$,
$\langle\{(a, \texttt{out}), (b, \texttt{in}), (c, \texttt{in}), (d, \texttt{out})\}, \ldots\rangle$,

We can also have cyclic behaviour:

$\langle\{(a, \texttt{in}), (b, \texttt{out}), (c, \texttt{undec}), (d, \texttt{undec})\}, \{(a, \texttt{undec}), (b, \texttt{undec}), (c, \texttt{out}), (d, \texttt{in})\}, \ldots\rangle$,

If we represent the latter sequence as a sequence of extensions, then we obtain the characteristic sequence of a predator-prey model.

$\langle\{a\}, \{d\}, \{a\}, \{d\}, \ldots\rangle$

The same model can be used also to describe the pork cycle, hog cycle, or cattle cycle[1] in economics, describing the phenomenon of cyclical fluctuations of supply and prices in livestock markets.

5 Related Work

Multi-sorted argumentation [26] considers a generic argumentation framework $AF = (\mathscr{A}, \mathscr{R})$ together with an arbitrary partition of $\mathscr{A}$. Such a partition identifies the restricted argumentation frameworks $AF{\downarrow_{P_1}}, \ldots, AF{\downarrow_{P_n}}$, that affect each other with the relevant input arguments and conditioning relations as stated in Definition 5.

A multi-sorted argumentation framework extends an argumentation framework with a partitioning and for each block P of the partitioning, a local function f_P.

Definition 12. *A multi-sorted argumentation framework is a tuple $\mathscr{F} = (\mathscr{A}, \mathscr{R}, \mathscr{P}, f)$ extending an argumentation framework $(\mathscr{A}, \mathscr{R})$ with a partition $\mathscr{P} = \{P_1, \ldots, P_n\}$ of $\mathscr{A}$, and a function f associating a local function f_P with every element P of $\mathscr{P}$.*

Any labelling of a restricted framework is used by f for computing the other ones: L_{P_i} plays a role in determining $L_{P_1}, \ldots, L_{P_{i-1}}, L_{P_{i+1}}, \ldots, L_{P_n}$ and vice versa. This means that $L_{P_1}, \ldots, L_{P_n}$ are "compatible" if each L_{P_i} is produced by f for $AF{\downarrow}_{P_i}$ with the input arguments ${P_i}^{\text{inp}}$ labelled according to $L_{P_1}, \ldots, L_{P_{i-1}}, L_{P_{i+1}}, \ldots, L_{P_n}$. Definition 14 synthesizes all these considerations.

The extensions of a multi-sorted argumentation framework are defined as follows.

Definition 13. $\mathbf{L_S}(F) = \mathscr{U}(\mathscr{P}, AF, f)$ *where* $\mathscr{U}(\mathscr{P}, AF, f) = \{L_{P_1} \cup \ldots \cup L_{P_n} \mid L_{P_i} \in f(AF{\downarrow}_{P_i}, {P_i}^{inp}, (\bigcup_{j=1\ldots n, j \neq i} L_{P_j}){\downarrow}_{{P_i}^{inp}}, {P_i}^{R})\}$.

Also see the recent paper of Giacomin [15], who argues that disagreements are in general heterogeneous and thus should be treated in different ways according both to their nature and to the specific agents features. Moreover, he discusses a general model of abstract argumentation based on input/output argumentation, able to handle heterogeneous disagreements by means of multiple argumentation semantics at a local level.

Baroni et al. [7,26] aim at introducing a formal notion of semantics decomposability. To this purpose, consider a generic argumentation framework $AF = (\mathscr{A}, \mathscr{R})$ and an arbitrary partition of $\mathscr{A}$. Such a partition identifies the restricted argumentation frameworks $AF{\downarrow}_{P_1}, \ldots, AF{\downarrow}_{P_n}$, that affect each other with the relevant input arguments and conditioning relations as stated in Definition 5. Intuitively a semantics $\mathbf{S}$ is decomposable if $\mathbf{S}$ can be put in correspondence with a local function f such that:

- every labelling prescribed by $\mathbf{S}$ on AF, namely every element of $\mathbf{L_S}(F)$, corresponds to the union of n "compatible" labellings $L_{P_1}, \ldots, L_{P_n}$ of the restricted argumentation frameworks, all of them obtained applying f;
- in turn, each union of n "compatible" labellings $L_{P_1}, \ldots, L_{P_n}$ obtained applying f to the restricted frameworks gives rise to a labelling of AF.

The "compatibility" constraint mentioned above reflects the fact that any labelling of a restricted framework is used by f for computing the other ones: L_{P_i} plays a role in determining $L_{P_1}, \ldots, L_{P_{i-1}}, L_{P_{i+1}}, \ldots, L_{P_n}$ and vice versa. This means that $L_{P_1}, \ldots, L_{P_n}$ are "compatible" if each L_{P_i} is produced by f for $AF{\downarrow}_{P_i}$ with the input arguments ${P_i}^{\text{inp}}$ labelled according to $L_{P_1}, \ldots, L_{P_{i-1}}, L_{P_{i+1}}, \ldots, L_{P_n}$. Definition 14 synthesizes all these considerations.

Definition 14. *A semantics* $\mathbf{S}$ *is* fully decomposable *(or simply* decomposable*) iff there is a local function* f *such that for every argumentation framework* $AF = (\mathscr{A}, \mathscr{R})$ *and every partition* $\mathscr{P} = \{P_1, \ldots, P_n\}$ *of* $\mathscr{A}$, $\mathbf{L_S}(F) = \mathscr{U}(\mathscr{P}, AF, f)$ *where* $\mathscr{U}(\mathscr{P}, AF, f) = \{L_{P_1} \cup \ldots \cup L_{P_n} \mid L_{P_i} \in f(AF{\downarrow}_{P_i}, {P_i}^{inp}, (\bigcup_{j=1\ldots n, j \neq i} L_{P_j}){\downarrow}_{{P_i}^{inp}}, {P_i}^{R})\}$.

Argumentation by autonomous agents have been studied mostly in the context of strategic argumentation games, e.g. [1,17,22–24,27]. An agent in negotiation dialogues as studied in [1] characterise changes in the set of accepted arguments in response to new arguments another agent introduces into his/her local scope, which, as ours, respects agents locality. In comparison, the focus of our work is more on analysing how derivation, as done by local agents, of their local semantics influences arguments acceptance globally. local agent semantics. Agents attributes are discussed in [22]. While many studies on game-theoretic argumentation games have presupposed complete information (see [16]), realistic legal examples often involve uncertainty of the belief state of other agents', and a theory that adapts to incomplete information is highly relevant.

Rahwan and Larson [25] contemplate (re)construction of an argumentation framework from the arguments in a given argumentation framework that are distributed across agents. In the construction process, the agents may or may not reveal the global outcome to be obtained varies with their decisions.

Judgement aggregation [6,11,12,28] to determine acceptable arguments based on social choice theory or aggregation of argumentation frameworks [13] are being studied. While they are not the main focus of this paper, such studies become important when we deal with agents perception of other agents' local argumentation. We aim to extend our theory for that kind of a situation in a future work.

The contributions in the first volume of the Handbook of Formal Argumentation (HOFA) highlight the main innovations of this new stage of formal argumentation theory, appealing to all disciplines, including logic, computer science, law, philosophy, and linguistics. Maybe the most pressing question is how this theory of formal argumentation, developed from the area of non-monotonic logic and artificial intelligence, can be used as the foundations for informal argumentation in areas such as linguistics and law. Future volumes of the handbook series will consider extensions of Dung's theory, including numerical ones, dynamics and update, dialogue, and applications, for example in artificial intelligence, computer science, linguistics or legal reasoning. Please visit the website for more information: http://formalargumentation.org/

6 Conclusion

Dung introduced in 1995 a model of abstract argumentation focussing on the relation among arguments, in the sense that the acceptance of arguments depends on the acceptance of other agents. In his examples, arguments are derivations in logic programming, default logic, or game theory.

Many people have given a more dynamic interpretation to abstract argumentation, for example developing dialogue based decision procedures to determine whether an argument is accepted or not, or developing input/output argumentation frameworks.

Inspired by Reo, in this paper we go one step further and suggest that Dung frameworks can characterise argumentation in terms of the *interaction* among arguments, and that abstract argumentation can be characterised as the exogenous coordination of arguments. This implies that arguments themselves should not be seen as static derivations anymore, but as dynamic argumentation processes.

As an example, we showed how the argument graph can give rise not only to sets of extensions, as in Dung's semantics, but also to sequences of such extensions. Moreover, we show that the ecology interpretation of Barringer et al. can also be represented in our model without introducing numbers.

There are many issues for further study. For example, other elements of Reo can be introduced in abstract argumentation, more realistic examples can be modelled using the idea of dynamic arguments, and the formal methods of Reo can be compared to the formal techniques used in abstract argumentation.

References

1. Amgoud, L., Vesic, S.: A formal analyis of the role of argumentation in negotiation dialogues. J. Log. Comput. **5**, 957–978 (2012)
2. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
3. Arbab, F.: Abstract behavior types: a foundation model for components and their composition. Sci. Comput. Program. **55**(1–3), 3–52 (2005)
4. Arbab, F.: Composition of interacting computations. In: Goldin, D., Smolka, S., Wegner, P. (eds.) Interactive Computation, pp. 277–321. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-34874-3_12
5. Arisaka, R., Satoh, K., van der Torre, L.: Anything you say can be used against you in a court of law. In: AICOL (2018)
6. Awad, E., Booth, R., Tohmé, F., Rahwan, I.: Judgement aggregation in multi-agent argumentation. J. Log. Comput. **27**(1), 227–259 (2017)
7. Baroni, P., Boella, G., Cerutti, F., Giacomin, M., van der Torre, L., Villata, S.: On the input/output behavior of argumentation frameworks. Artif. Intell. **217**, 144–197 (2014)
8. Baroni, P., Gabbay, D., Giacomin, M., van der Torre, L. (eds.): Handbook of Formal Argumentation, vol. 1. College Publications (2018)
9. Barringer, H., Gabbay, D., Woods, J.: Temporal dynamics of support and attack networks: from argumentation to zoology. In: Hutter, D., Stephan, W. (eds.) Mechanizing Mathematical Reasoning. LNCS (LNAI), vol. 2605, pp. 59–98. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32254-2_5
10. Bochman, A.: Explanatory Nonmonotonic Reasoning. World Scientific, Singapore (2005)
11. Bodanza, G.A., Auday, M.R.: Social argument justification: some mechanisms and conditions for their coincidence. In: Sossai, C., Chemello, G. (eds.) ECSQARU 2009. LNCS (LNAI), vol. 5590, pp. 95–106. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02906-6_10
12. Caminada, M., Pigozzi, G.: On judgement aggregation in abstract argumentation. Auton. Agent. Multi-Agent Syst. **22**(1), 64–102 (2011)
13. Coste-Marquis, S., Devred, C., Konieczny, S.: On the merging of Dung's argumentation systems. Artif. Intell. **171**(10–15), 730–753 (2007)
14. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming, and n-Person games. Artif. Intell. **77**(2), 321–357 (1995)
15. Giacomin, M.: Handling heterogeneous disagreements through abstract argumentation (extended abstract). In: An, B., Bazzan, A., Leite, J., Villata, S., van der Torre, L. (eds.) PRIMA 2017. LNCS (LNAI), vol. 10621, pp. 3–11. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69131-2_1

16. Governatori, G., Olivieri, F., Rotolo, A., Scannapieco, S., Sartor, G.: Two faces of strategic argumentation in the law. In: JURIX, pp. 81–90 (2014)
17. Grossi, D., van der Hoek, W.: Audience-based uncertainty in abstract argument games. In: IJCAI, pp. 143–149 (2013)
18. Liao, B.: Toward incremental computation of argumentation semantics: a decomposition-based approach. Ann. Math. Artif. Intell. **67**(3–4), 319–358 (2013)
19. Liao, B., Jin, L., Koons, R.C.: Dynamics of argumentation systems: a division-based method. Artif. Intell. **175**(11), 1790–1814 (2011)
20. Makinson, D., van der Torre, L.: Input/output logics. J. Philos. Log. **29**(4), 383–408 (2000)
21. Makinson, D., van der Torre, L.: Constraints for input/output logics. J. Philos. Log. **30**(2), 155–185 (2001)
22. Parsons, S., Sklar, E.: How agents alter their beliefs after an argumentation-based dialogue. In: Parsons, S., Maudet, N., Moraitis, P., Rahwan, I. (eds.) ArgMAS 2005. LNCS (LNAI), vol. 4049, pp. 297–312. Springer, Heidelberg (2006). https://doi.org/10.1007/11794578_19
23. Procaccia, A., Rosenschein, J.: Extensive-form argumentation games. In: EUMAS, pp. 312–322 (2005)
24. Rahwan, I., Larson, K.: Argumentation and game theory. In: Simari, G., Rahwan, I. (eds.) Argumentation in Artificial Intelligence, pp. 321–339. Springer, Boston (2009). https://doi.org/10.1007/978-0-387-98197-0_16
25. Rahwan, I., Larson, K.: Mechanism design for abstract argumentation. In: AAMAS, pp. 1031–1038 (2008)
26. Rienstra, T., Perotti, A., Villata, S., Gabbay, D.M., van der Torre, L.: Multi-sorted argumentation. In: Modgil, S., Oren, N., Toni, F. (eds.) TAFA 2011. LNCS (LNAI), vol. 7132, pp. 215–231. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29184-5_14
27. Riveret, R., Prakken, H.: Heuristics in argumentation: a game theory investigation. In: COMMA, pp. 324–335 (2008)
28. Tohmé, F.A., Bodanza, G.A., Simari, G.R.: Aggregation of attack relations: a social-choice theoretical analysis of defeasibility criteria. In: Hartmann, S., Kern-Isberner, G. (eds.) FoIKS 2008. LNCS, vol. 4932, pp. 8–23. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-77684-0_4

Extending Paradigm with Data

Luuk P. J. Groenewegen[1], Jan H. S. Verschuren[1], and Erik P. de Vink[2,3](✉)

[1] Leiden Institute of Advanced Computer Science, Leiden University,
Leiden, The Netherlands
[2] Department of Mathematics and Computer Science,
Eindhoven University of Technology, Eindhoven, The Netherlands
evink@win.tue.nl
[3] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands

Abstract. We discuss an extension of the coordination modeling language Paradigm. The extension is geared towards data-dependent interaction among components, where the coordination is influenced by possibly distributed data. The approach is illustrated by the well-known example of a bakery where tickets are issued to serve clients in order. Also, it is described how to encode Paradigm models with data in the process language of the `mCRL2` toolset for further analysis of the coordination.

1 Introduction

The so-called IWIM model for the coordination of concurrent components as proposed by Farhad Arbab and co-workers [3,6] distinguishes ideal workers and ideal managers. Among others, IWIM forms the conceptual framework for the coordination language Manifold [4,7]. The central ideas of IWIM evolved into the theory of Reo connectors [5], which exploits constraint automata for its semantics and whose distributed implementation approach separates coordination from parallelism [11].

Rather than considering hierarchies of components with atomic workers at the bottom layer and one overall manager at the top as for IWIM, the coordination modeling language Paradigm [9] takes networks of components as starting point, where each component exhibits both worker and manager activity. The worker activity is the internal behavior of the component that executes as local transitions asynchronously from other components; the manager activity consists of the synchronous interaction with other (groups of) components governed by so-called consistency rules. In terms of constraint automata, consistency rules comprise the atomic dataflow among synchronizing components. However, via a mechanism of phases and traps it is guaranteed that the local behavior, the worker level of a component, remains aligned with the global behavior, the manager level of the component.

In this paper an extension to Paradigm including data is proposed. In this extension, consistency rules incorporate the local variables of the components and expressions thereof, in particular to compare or communicate their value. So,

F. de Boer et al. (Eds.): Arbab Festschrift, LNCS 10865, pp. 224–244, 2018.
https://doi.org/10.1007/978-3-319-90089-6_15

our data extension will be geared towards interaction and coordination thereof. Cast in terms of Reo, the data constraints are enriched with data and values. For Paradigm with data, the local memory of components can be accessed (via their ports) at the coordination level. Consequently, the communicated data itself can be stored too. However, this requires that the phases-and-trap mechanism of Paradigm needs to be adapted, somewhat complicating the semantics. An encoding scheme for Paradigm, without data, into the model checking toolsuit `mCRL2`[1], as proposed in [1], brings the advantage of formal analysis of the coordination among components. For a concrete coordination problem, we will describe a Paradigm model with data of a bakery, describe its encoding in `mCRL2`'s specification language.

Outline Sect. 2 provides a formal definition of Paradigm with data and provides its operational semantics. Section 3 illustrates and further explains the underlying concepts for the case of a bakery where clients need to be served in order of arrival. Section 4 discusses how formal analysis of Paradigm using the `mCRL2` toolset can be obtained. Section 5 wraps up the paper.

2 Formal Definitions

We subsequently introduce components with variables, Paradigm models and consistency rules with data, and configurations with local transitions and global transfers among them. An example illustrating the above notions is presented in the next section.

Definition 1. *Let, for some index set I, a number of local variables v_i of type D_i, respectively, be given. Put $E = \prod_{i \in I} D_i$. Furthermore, fix a set of actions A. For E and A, a Paradigm component C is a tuple $C = (\Sigma, T, \Psi)$ where*

(i) for some set S, the elements of which are called states, $\Sigma = S \times E$ is the set of extended states of C
(ii) $T \subseteq \Sigma \times A \times \Sigma$ is the transition relation of C
(iii) $\Psi = (\Phi_1, \ldots, \Phi_n)$, for some $n \geqslant 0$, is a tuple of partial functions, called the roles of C, where each $\Phi : \mathcal{P}(T) \hookrightarrow \mathcal{P}(\mathcal{P}(\Sigma))$ is such that if $\sigma \in \theta$, $\theta \in \Phi(\varphi)$, and $\langle \sigma, a, \sigma' \rangle \in \varphi$ then also $\sigma' \in \theta$.

By definition, an extended state $\sigma \in \Sigma$ is a pair $\sigma = (s, e)$ of a state $s \in S$ and a tuple of 'current' values of the variables. We write $\sigma \xrightarrow{a} \sigma'$ for a transition in T, rather than $\langle \sigma, a, \sigma' \rangle \in T$. For a role Φ, i.e. a coordinate of Ψ, an element φ of $\mathrm{dom}(\Phi)$ is called a phase of Φ. An element θ of $\Phi(\varphi)$ is called a trap of φ. The idea is, a transition, $\sigma \xrightarrow{a} \sigma'$ starting from an extended state σ in a trap θ of a phase φ say, does not move outside of the trap. Hence, it is required for such a transition $\sigma \xrightarrow{a} \sigma'$ that the extended state σ' lies in θ too. So, in phase φ, once having entered θ, control is trapped in θ. The phases constituting $\mathrm{dom}(\Phi)$ of a

[1] See www.mcrl2.org.

role Φ will typically partially overlap, their overlaps being traps. One may think of a trap as a final stage within a phase. Reaching a trap of a phase indicates that a transfer to another phase is about to happen.

Suppose $\varphi_i \in \Phi_i$, for $i = 1, \ldots, n$, for the roles $\Phi_1, \ldots, \Phi_n$ of component C, and suppose $\sigma \xrightarrow{a} \sigma'$ is a transition of C, i.e. an element of T, such that the transition $\sigma \xrightarrow{a} \sigma'$ is an element of each φ_i too. Then the transition $\sigma \xrightarrow{a} \sigma'$ is called an admitted transition with respect to the phases $\varphi_1, \ldots, \varphi_n$.

Definition 2.

(a) A Paradigm model with data consists, for some index set H, of a tuple $(C_h)_{h \in H}$ of Paradigm components

$$C_h = (\,\Sigma_h, T_h, \Psi_h\,)$$

with their own local variables and actions, as well as extended states in Σ_h, transition relations T_h, and roles $\Psi_h = (\Phi_{h,1}, \ldots, \Phi_{h,n_h})$, for $h \in H$.

(b) A consistency rule γ for $(C_h)_{h \in H}$ consists, for an index set R, of a tuple $(\,C_r(\Phi_r) : \varphi_r(e_r) \xrightarrow{\theta_r} \varphi'_r(e'_r)\,)_{r \in R}$ where Φ_r is a role of component C_r, φ_r and φ'_r are phases of Φ_r, e_r and e'_r are values for the variables of C_r, and θ_r is a trap of φ_r.

(c) A set of consistency rules Γ is called closed if for each rule $(\,C_r(\Phi_r) : \varphi_r(e_r) \xrightarrow{\theta_r} \varphi'_r(e'_r)\,)_{r \in R}$ of Γ, if there exists, for some $r \in R$, a state s_r of C_r for which both $(s_r, e_r), (s_r, \bar{e}_r) \in \theta_r$, then Γ contains, for some $\bar{e}'_r$, a rule $\bar{\gamma}$ of the form $(\,C_r(\Phi_r) : \varphi_r(\bar{e}_r) \xrightarrow{\theta_r} \varphi'_r(\bar{e}'_r)\,)_{r \in R}$ as well.

For clarity we assume that different components have distinct names for states, variables, and actions, and hence distinct roles, phases, and traps. However, in a consistency rule, a component may have multiple occurrences, viz. in different roles. Also, a component may not be involved in a consistency rule at all. The rules are called *consistency* rules in Paradigm because the requirement of each θ_r to be a trap of phase φ_r guarantees that the 'coarse-grained' rule can only be applied if *consistent* with the current 'fine-grained' local state of each component involved. The closedness condition on sets of rules will guarantee that global behavior, to be defined in a minute, cannot be essentially influenced by local behavior respecting the traps mentioned in a rule.

Next, we define the behavior of a Paradigm model, with intra-component behavior (a so-called local transition) affecting the extended state of a single component vs. inter-component behavior (a global transfer) exchanging values and changing phases based on a trap in some of the roles of a number of components.

Definition 3. *Let $\Pi = (C_h)_{h \in H}$ be a Paradigm model and let Γ be a closed set of consistency rules for Π.*

(a) A configuration of Π is a tuple $\langle s_h, e_h, \psi_h \rangle_{h \in H}$, where for each index $h \in H$, (s_h, e_h) is an extended state of component C_h, and $\psi_h = (\varphi_{h,1}, \ldots, \varphi_{h,n_h})$ is a tuple of phases such that $\varphi_{h,i} \in \Phi_{h,i}$, for $i = 1, \ldots, n_h$.

(b) A local transition $\langle s_h, e_h, \psi_h \rangle_{h \in H} \xrightarrow{a} \langle s'_h, e'_h, \psi_h \rangle_{h \in H}$ *of* $\mathit{\Pi}$ *is an admitted transition of one of the components of* $\mathit{\Pi}$*, i.e. for some* $h_0 \in H$ *it holds that (i)* $\langle s_{h_0}, e_{h_0} \rangle \xrightarrow{a} \langle s'_{h_0}, e'_{h_0} \rangle$ *is an admitted transition for component* C_{h_o} *with respect to the phases* $\psi_{h_0} = (\varphi_{h_0,1}, \ldots, \varphi_{h_0,n_{h_0}})$*, and (ii)* $s_h = s'_h$ *and* $e_h = e'_h$ *for each index* $h \neq h_0$ *in* H*.*

(c) A global transfer $\langle s_h, e_h, \psi_h \rangle_{h \in H} \xrightarrow{\gamma} \langle s_h, e'_h, \psi'_h \rangle_{h \in H}$ *of* $\mathit{\Pi}$ *based on a consistency rule* $\gamma = (\, \hat{C}_r(\mathit{\Phi}_r) : \varphi_r(\hat{e}_r) \xrightarrow{\theta_r} \varphi'_r(\hat{e}_h)\,)_{r \in R}$ *updates phases and values as prescribed by* γ*, i.e. (i) if, for* $h \in H$*,* $i = 1, \ldots, n_h$*, it holds that* $C_h = \hat{C}_r$ *and* $\mathit{\Phi}_{h,i} = \mathit{\Phi}_r$*, for some index* $r \in R$*, then* $(s_h, e_h) \in \theta_r$*,* $e_h = \hat{e}_r$ *and* $e'_h = \hat{e}_r$*,* $\varphi_{h,i} = \varphi_r$ *and* $\varphi'_{h,i} = \varphi'_r$*, and (ii) if, for* $h \in H$*,* $C_h \neq \hat{C}_r$ *for each* $r \in R$ *then* $e_h = e'_h$*, and, for* $h \in H$*,* $i = 1, \ldots, n_h$*,* $\mathit{\Phi}_{h,i} \neq \mathit{\Phi}_r$ *for each* $r \in R$ *then* $\varphi_{h,i} = \varphi'_{h,i}$*.*

A configuration $\langle s_h, e_h, \psi_h \rangle_{h \in H}$ of a Paradigm model $\mathit{\Pi} = (C_h)_{h \in H}$ holds for each component C_h the current extended state (s_h, e_h) as well as the current phase $\varphi_{h,i}$ for each role $\mathit{\Phi}_{h,i}$ of C_h.

Note, in a local transition $\langle s_h, e_h, \psi_h \rangle_{h \in H} \xrightarrow{a} \langle s'_h, e'_h, \psi_h \rangle_{h \in H}$, say for component C_{h_0}, component C_{h_0} nor any of the other components changes phase; the tuple of phases ψ_h is for each component the same in the source configuration $\langle s_h, e_h, \psi_h \rangle_{h \in H}$ and the target configuration $\langle s'_h, e'_h, \psi_h \rangle_{h \in H}$ of the transition. However, the transition must be admitted for C_{h_0}, i.e. it must be present in all of the phases $\varphi_{h_0,i}$ of ψ_{h_0} for component C_{h_0}.

For a global transfer based on a consistency rule γ to apply, the current phases $\varphi_{h,i}$ of role $\mathit{\Phi}_{h,i}$ must match the phases of $\mathit{\Phi}_r$, if $C_h = \hat{C}_r$ and $\mathit{\Phi}_{h,i} = \mathit{\Phi}_r$. Also, the extended states of the components $\hat{C}_r$ involved must lie in the traps θ_r, for all $r \in R$. States remain unaffected, but values of variables may change for the components mentioned in the rule, presumably because of the interaction. Phases may change too for the components mentioned, from $\varphi_{h,i} = \varphi_r$ to $\varphi_{h,i} = \varphi'_r$, which are both phases within the role $\mathit{\Phi}_{h,i} = \mathit{\Phi}_r$. Components C_h and phases $\varphi_{h,i}$ not mentioned by consistency rule γ remain the same.

We have the following result.

Theorem 1. *Let* $\mathit{\Pi} = (C_h)_{h \in H}$ *be a Paradigm model, and let* $\mathit{\Gamma}$ *be a closed set of consistency rules for* $\mathit{\Pi}$*. Suppose*

$$\langle s_h, e_h, \psi_h \rangle_{h \in H} \xrightarrow{\gamma} \langle s_h, e'_h, \psi'_h \rangle_{h \in H} \quad \text{and} \quad \langle s_h, e_h, \psi_h \rangle_{h \in H} \xrightarrow{a} \langle s'_h, e''_h, \psi_h \rangle_{h \in H}$$

for configurations $\langle s_h, e_h, \psi_h \rangle_{h \in H}$*,* $\langle s_h, e'_h, \psi'_h \rangle_{h \in H}$*, and* $\langle s'_h, e''_h, \psi_h \rangle_{h \in H}$ *of* $\mathit{\Pi}$*, a consistency rule* γ *in* $\mathit{\Gamma}$*, and a local transition for* a*. Then also*

$$\langle s_h, e_h, \psi_h \rangle_{h \in H} \xrightarrow{\gamma} \langle s_h, \bar{e}'_h, \psi'_h \rangle_{h \in H}$$

for suitable values $\bar{e}'_h$*, for* $h \in H$*.*

The theorem is a direct consequence of the closedness condition for the set of consistency rules. It states that the execution of a local transition cannot disable the execution of a consistency rule. This is the loose coupling in Paradigm between the interaction between components and actions of the components of their own. The reverse obviously doesn't hold. A local transition that was admitted before, may be forbidden by one of the phases put in place by the execution of a consistency rule. Care has to be taken to deal with variables that are set by local transitions as well as by global transfer. To ensure non-interference of the global (manager) and local (worker) level, one may want to restrict reading or updating of variables to happen outside of the traps involved in consistency rules that may change the value of the different variables.

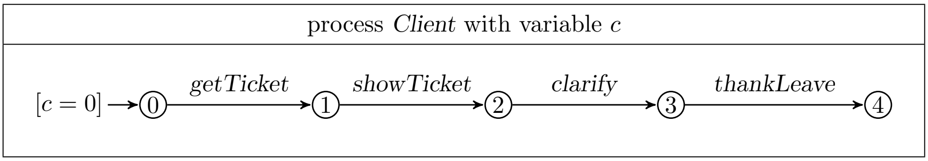
3 An Example Paradigm Model

We illustrate the formal definitions of the previous section by modeling in Paradigm with data the handling of clients in a busy bakery. Clients entering the shop take a ticket from a ticket dispenser and wait for their turn. The client having the ticket displayed is being served. The baker increments the display after having handled a client and next serves the clients holding the ticket with the new number.

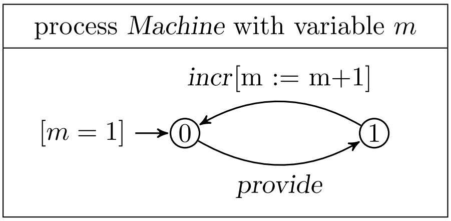
3.1 STDs for the Components

We first model the basic behavior of the components by means of state-transition diagrams (STD).

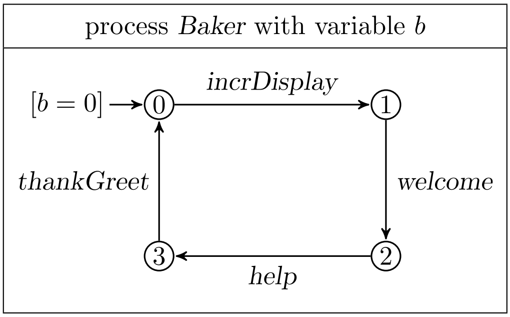
Client processes are introduced by the state-transition diagram below. Each client carries an integer variable c to hold a ticket number. Initially c is set to 0. A client subsequently obtains a ticket from the ticket machine, action *getTicket*, shows the ticket to the baker (action *showTicket*), clarifies his or her wishes (action *clarify*), and finally thanks the baker and leaves (action *thankLeave*). Note, apart from initialization, there is no explicit assignment to variable c in the STD. Also, we don't bother to distinguish multiple instances of the *Client* process. Incorporating another variable, *id* say, holding the identity of a client would cater for this.



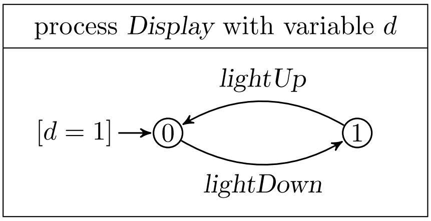
The ticket machine is modeled by the process *Machine* which has an integer variable m for the number of the ticket it dispenses. Starting from initial value 1 for m, the machine may provide a ticket with the current value of m while moving to state 1 (action *provide*) and returns to state 0 on an increment of the value of m (action *incr*). The *incr*-action is decorated with an assignment, viz. the increment $m := m + 1$ of variable m.



The *Baker* process models the workflow for the baker. Starting from the initial state 0, with initial value 0 for the integer variable b of the process, the process cycles through its four states. First the baker aims to increment the display (action *incrDisplay*), next the baker welcomes the client holding the number displayed (action *welcome*) and helps the client (action *help*). The baker closes the cycle by some thanks and greetings (action *thankGreet*). Note, also here no explicit assignments to the variable b are present; changes to b will come from the interaction with the *Display* process described below.



The *Display* process is similar to the *Machine* process. It switches between two states. The *Display* process holds an integer variable d, initially set to 1. However, here we have chosen not to have an update of the variable in the STD as we have for the machine. As variation, the display gets incremented in the interaction with the baker. This is captured by the consistency rules modeling the interplay of these two processes.

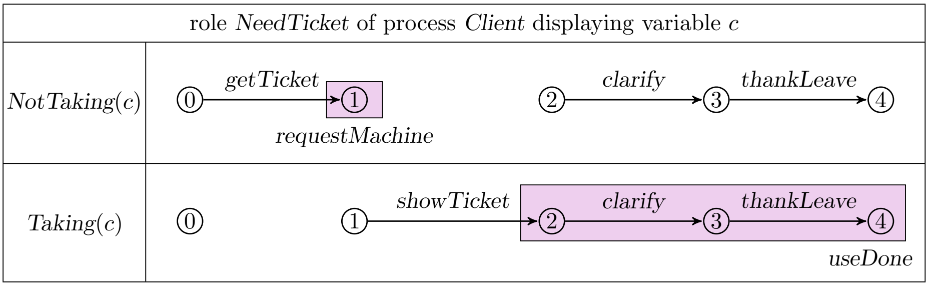


3.2 Roles of the Components

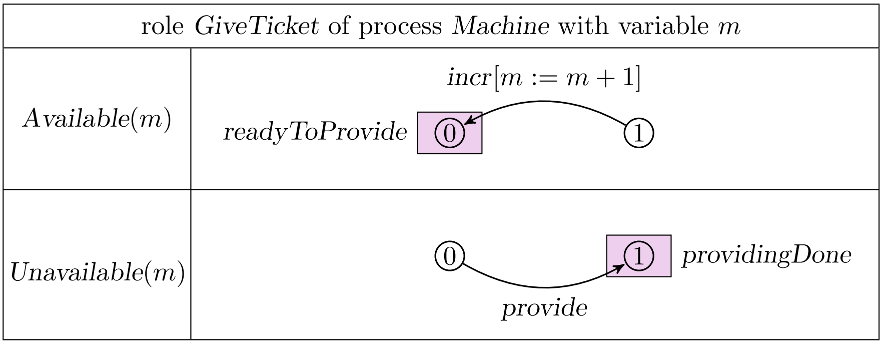
As discussed above, in Paradigm a component can have multiple roles. At the level of the roles the interaction with other components takes place. The phase-and-trap discipline of Paradigm ensures that STD and roles remain aligned during execution: a local transition cannot change the current phase or move out of a trap.

A *Client* process has two roles, *NeedTicket* and *NeedService*, in which it interacts with the *Machine* process and *Baker* process, respectively. The variable c of the *Client* process may be read and/or written during this interaction and is therefore displayed as parameter of the phases involved.

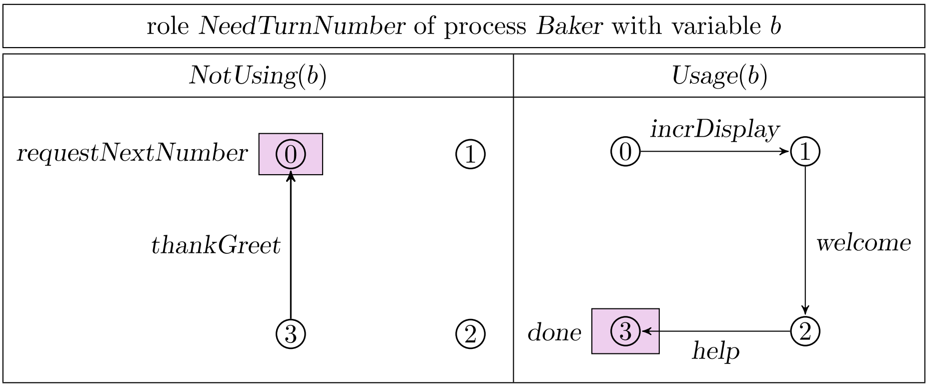
The role *NeedTicket* has two phases, *NotTaking* and *Taking*. The phase *NotTaking* only allows the action *getTicket* modeling that a ticket needs to be obtained first. When state 1 is reached in the STD the trap *requestMachine* has been entered, signaling that in the role *NeedTicket* the component is prepared to leave phase *NotTaking* (and ready to enter phase *Taking*, as we shall see). Phase *Taking* models a client in possession of a ticket. When state 2 has been reached, the trap *useDone* is entered. As required for a trap, the transitions for actions *clarify* and *thankLeave* do not leave trap *useDone*.



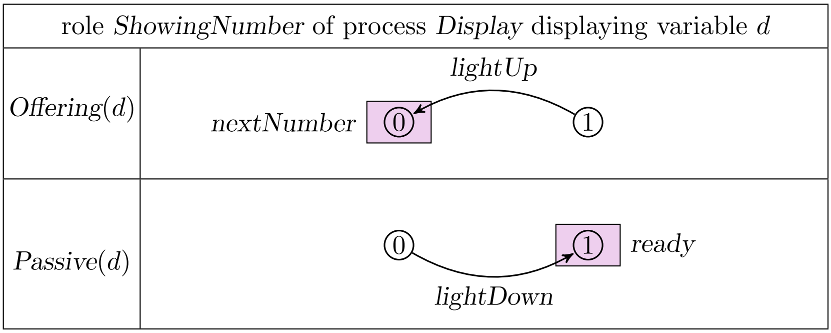
The *Machine* process in its single role *GiveTicket* interacts with the *Client* processes in their roles *NeedTicket*. The role *GiveTicket* has two phases, *Available* and *Unavailable*, that manage the variable m, as indicated. Both phases have a single-state trap, trap *readyToProvide* for phase *Available*, which indicates that a new ticket is available for issue when the trap is reached, and trap *providingDone* of phase *Unavailable*, that indicates that the current ticket number has been issued and the variable m needs to be adapted (as it actually will be in phase *Available*). Note, since variable m is updated when transition *incr*$[m := m + 1]$ is taken, the consistency rule (CM3) presented below doesn't have an increment of its parameter.



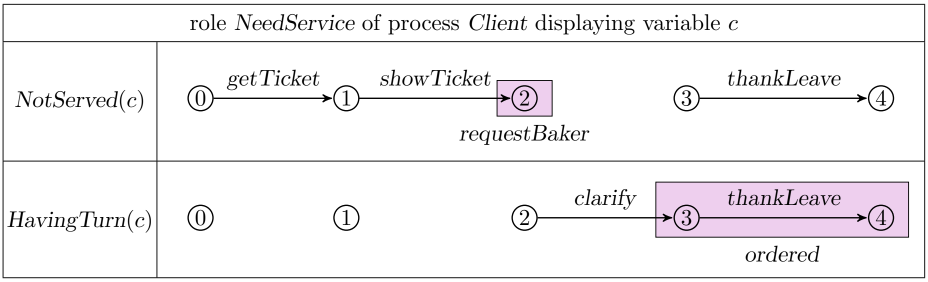
The *Baker* process has two roles, role *NeedTurnNumber* for interaction with the *Display* process, and role *NeedNextClient* for interaction with all *Client* processes. Role *NeedTurnNumber* distinguishes the phases *NotUsing* and *Usage*, that are connected by trap *requestNextNumber* from phase *NotUsing* to phase *Usage* and by trap *done* the other way around. The number of the client at turn is kept in the variable b of process *Baker*.


role *NeedTurnNumber* of process *Baker* with variable b

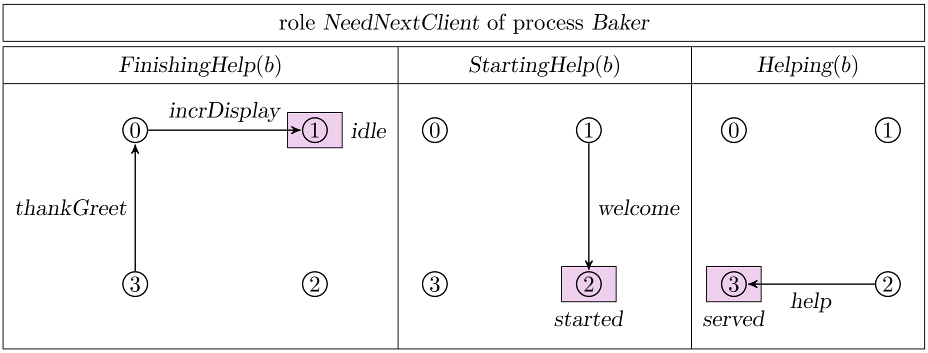
When the *Baker* process needs to know the next ticket number to store it in its variable b, this is provided by the *Display* process, in its single role *ShowingNumber*. In phase *Offering* the value of the variable d of process *Display* is guaranteed to be updated upon reaching trap *nextNumber*. To enforce such an update, phase *Offering* is switched to phase *Passive*, which will move control of *Display* to state 1 from which a next increment is possible once, and which is, via trap *ready*, switched back to phase *Offering*. Note, when changing phase from phase *Passive* to phase *Offering* as prescribed by consistency rule (BD3), given in the next subsection, the variable d will be incremented. Different from the modeling of role *GiveTicket* of the *Machine* process presented above, the role *ShowingNumber* of the *Display* process doesn't update the variable d itself.


role *ShowingNumber* of process *Display* displaying variable d

The role *NeedService* of the *Client* process deals with the client-side in the interaction with the *Baker* process. The role has two phases, *NotServed* and *HavingTurn*, each making use of the variable c of the *Client* process during interaction, viz. to match the ticket number announced by the baker. Only in case of a match, the *Client* process will change phase to *HavingTurn*, based on the trap *requestBaker*. To highlight, be it a bit sketchy, that traps aren't necessarily comprised of a single state, the trap *ordered* of phase *HavingTurn* allows a transfer back to the phase *NotServed* again.



The role *NeedNextClient* of process *Baker* takes care of the baker's part in the interaction with a client. When having reached trap *idle* in phase *FinishingHelp* (finishing helping a previous client), a transfer will take place (by consistency rule (BC1) discussed below) to phase *StartingHelp*. Similarly, in phase *StartingHelp* on reaching trap *started* a transfer will take place (now by consistency rule (BC2)) to phase *Helping*, in which the client is actually served. After trap *ready* has been reached in phase *Helping*, the *Baker* process will switch to phase *FinishingHelp* in role *NeedNextClient*.



3.3 Interactions Among Components

The interaction between the *Client* processes, in their roles *NeedTicket*, and the *Machine* process, in its role *GiveTicket*, arranges that every client entering the bakery is provided with a uniquely numbered ticket. The three consistency rules (CM1), (CM2), and (CM3) below describe how phases change once proper traps have been entered.

$$\otimes\left\{\begin{array}{rrcl} \mathit{Client}(\mathit{NeedTicket}): & \mathit{NotTaking}(0) & \xrightarrow{\mathit{requestMachine}} & \mathit{Taking}(n) \\ \mathit{Machine}(\mathit{GiveTicket}): & \mathit{Available}(n) & \xrightarrow{\mathit{readyToProvide}} & \mathit{Unavailable}(n) \end{array}\right. \qquad \text{(CM1)}$$

$$\mathit{Client}(\mathit{NeedTicket}): \quad \mathit{Taking}(n) \xrightarrow{\mathit{useDone}} \mathit{NotTaking}(n) \qquad \text{(CM2)}$$

$$\mathit{Machine}(\mathit{GiveTicket}): \mathit{Unavailable}(n) \xrightarrow{\mathit{providingDone}} \mathit{Available}(n) \qquad \text{(CM3)}$$

The first consistency rule, rule (CM1), is a synchronous transfer of phases, as indicated by the ⊗-symbol and enclosing braces. Given that (i) the *Client* process, in role *NeedTicket*, has reached trap *requestMachine* of phase *NotTaking*, while (ii) the *Machine* process, in role *GiveTicket* resides in trap *readyToProvide* of phase *Available*, then (i) the *Client* process switches to phase *Taking* of role *NeedTicket*, while (ii) the *Machine* process simultaneously changes to phase *Unavailable* of role *GiveTicket*. Moreover, (i) the *Client* process is assumed to (still) hold the initial value 0, while (ii) the *Machine* process has with a (presumably fresh) ticket number n, then the value n is copied from the *Machine* process to the *Client* process. For consistency rules (CM2) and (CM3) the *Client* and *Machine* process act independently. Based on (CM2), the *Client* process can change phase, from *Taking* to *NotTaking*, via trap *useDone*. Based on (CM3), the *Machine* process can change phase, from *Unavailable* to *Available*, via trap *providingDone*.

The interaction between the *Baker* and *Display* process is governed by the three consistency rules (BD1), (BD2), and (BD3). A similar effect is achieved as for rules (CM1) through (CM3). However, the actual update of variable m is done for the *Machine* process at the STD-level by the transition from state 1 to state 0 executing action $\mathit{incr}[m := m+1]$. Here, for process *Display* the update is accomplished at the level of role *ShowingNumber* by rule (BD3), which passes the parameter value m for phase *Passive* to phase *Offering* as parameter value $m{+}1$.

$$\otimes\left\{\begin{array}{rrcl} \mathit{Baker}(\mathit{NeedTurnNumber}): & \mathit{NotUsing}(n) & \xrightarrow{\mathit{requestNextNumber}} & \mathit{Usage}(m) \\ \mathit{Display}(\mathit{ShowingNumber}): & \mathit{Offering}(m) & \xrightarrow{\mathit{nextNumber}} & \mathit{Passive}(m) \end{array}\right. \qquad \text{(BD1)}$$

$$\mathit{Baker}(\mathit{NeedTurnNumber}): \quad \mathit{Usage}(n) \xrightarrow{\mathit{done}} \mathit{NotUsing}(n) \qquad \text{(BD2)}$$

$$\mathit{Display}(\mathit{ShowingNumber}): \quad \mathit{Passive}(m) \xrightarrow{\mathit{ready}} \mathit{Offering}(m{+}1) \qquad \text{(BD3)}$$

The interaction of the *Baker* and *Client* processes in their respective roles *NeedNextClient* and *NeedService* is more tied up compared to the interactions described above. All three consistency rules (BC1), (BC2), and (BC3) prescribe simultaneous phase transfer for the two processes. Moreover, the value of the variable b of the *Baker* process must be equal to the variable c of the *Client* process; they must both have the value n.

$$\otimes\left\{\begin{array}{rrcl} \mathit{Baker}(\mathit{NeedNextClient}): & \mathit{FinishingHelp}(n) & \xrightarrow{\mathit{idle}} & \mathit{StartingHelp}(n) \\ \mathit{Client}(\mathit{NeedService}): & \mathit{NotServed}(n) & \xrightarrow{\mathit{requestBaker}} & \mathit{NotServed}(n) \end{array}\right. \tag{BC1}$$

$$\otimes\left\{\begin{array}{rrcl} \mathit{Baker}(\mathit{NeedNextClient}): & \mathit{StartingHelp}(n) & \xrightarrow{\mathit{started}} & \mathit{Helping}(n) \\ \mathit{Client}(\mathit{NeedService}): & \mathit{NotServed}(n) & \xrightarrow{\mathit{requestBaker}} & \mathit{HavingTurn}(n) \end{array}\right. \tag{BC2}$$

$$\otimes\left\{\begin{array}{rrcl} \mathit{Baker}(\mathit{NeedNextClient}): & \mathit{Helping}(n) & \xrightarrow{\mathit{served}} & \mathit{FinishingHelp}(n) \\ \mathit{Client}(\mathit{NeedService}): & \mathit{HavingTurn}(n) & \xrightarrow{\mathit{ordered}} & \mathit{NotServed}(n) \end{array}\right. \tag{BC3}$$

4 Model Checking Paradigm Models with Data

As for many modeling languages, formal analysis of Paradigm models supports the modeling itself. In [1,2] it is discussed how to expressing Paradigm models without data in the process language of the mCRL2 toolset [8,10]. In short, for each component the local behavior is modeled as a state machine. For a transition to fire, it is checked if the current phase allows so. For the global behavior of a component a communication intent is issued for each consistency rule that mentions the component. However, the current state and phase should match the relevant trap. Correct interaction can subsequently be enforced by the allow and communication operators of mCRL2, that block single-sided communication intents and synchronize consistent ones, respectively. In this section, we describe by example how the approach extends to deal with data.

```
proc Client(st:Nat,c:Nat,nt_ph:NeedTicketPh,ns_ph:NeedServicePh) =

((st==0) && (nt_ph in [NotTaking]) && (ns_ph in [NotServed])) ->
      getTicket . Client(1,c,nt_ph,ns_ph) +
((st==1) && (nt_ph in [Taking]) && (ns_ph in [NotServed])) ->
    showTicket(c) . Client(2,c,nt_ph,ns_ph) +
((st==2) && (nt_ph in [NotTaking,Taking]) && (ns_ph in [HavingTurn])) ->
     clarify(c) . Client(3,c,nt_ph,ns_ph) +
((st==3) && (nt_ph in [NotTaking,Taking]) && (ns_ph in [HavingTurn])) ->
     thankLeave(c) . Client(4,c,nt_ph,ns_ph) +

%% rule (CM1)
  ((st in [1]) && (nt_ph==NotTaking)) ->
     sum m:Nat . requestMachine(m) . Client(st,m,Taking,ns_ph) +
%% rule (CM2)
   ((st in [2,3,4]) && (nt_ph==Taking)) ->
     useDone . Client(st,c,NotTaking,ns_ph) +

%% rule (BC1)
  ((st in [2]) && (ns_ph==NotServed)) ->
    requestBaker1(c) . Client(st,c,nt_ph,NotServed) +
%% rule (BC2)
  ((st in [2]) && (ns_ph==NotServed)) ->
    requestBaker2(c) . Client(st,c,nt_ph,HavingTurn) +
%% rule (BC3)
  ((st in [3,4]) && (ns_ph==HavingTurn)) ->
    ordered(c) . Client(st,c,nt_ph,NotServed) ;
```

Fig. 1. mCRL2 code for the *Client* process

Figure 1 provides the mCRL2 version of the process *Client* of the bakery example of the previous section (the complete code can be found in the appendix). Here, the Client process has four parameters, viz. the natural number st to hold the state of the underlying STD, the natural number c to hold the ticket number of the client, and the parameters nt_ph and ns_ph to keep track of the phase of the process with respect to their roles *NeedTicket* and *NeedService*, respectively. For this, the definition of the two enumerated types

```
NeedTicketPh = struct NotTaking | Taking;
NeedServicePh = struct NotServed| HavingTurn;
```

are included at the beginning of the specification. The specification of the Client process falls into three parts: (i) lines 3–10, specifying the local transitions (ii) lines 12–17, describing *Client*'s part for the consistency rules (CM1) – (CM3), and similarly (iii) lines 19–27 for the consistency rules (BC1) – (BC3). Each part consists of a number of alternative branches, separated by the non-deterministic choice operation '+', of the form

```
<condition> -> <action> . <continuation process>
```

(with a variation for rule (CM1) to be discussed in a minute). For example, lines 3–4 express that the Client process in state 0, in phase NotTaking for its *NeedTicket* role, as well as in phase NotServed for its *NeedService* role, can perform the action getTicket and continues as Client(1,c,nt_ph,ns_ph) with control now in state 1, but leaving the parameters c, nt_ph, and ns_ph unchanged. The element-of-list construction nt_ph in [NotTaking ,Taking] shows profitable in line 7, for example. Note, the transition ② $\xrightarrow{\mathit{clarify}}$ ③ is admitted both by phase *NotTaking* and by phase *Taking*.

Lines 12–14 embody the contribution of a client process to execution of the (CM2)-rule. If the process resides in trap *useDone* consisting of states 2, 3, and 4 (for all values of variable *c*), and in phase *Taking* regarding its *NeedTicket* role, then the process is willing to execute the action *useDone* and to continue with its *NeedTicket* phase changed from *Taking* to *NotTaking* as consistency rule (CM2) prescribes. Note, no for (CM2) the client process doesn't depend on other processes.

Consistency rule (CM1) in which both a client process and the machine process are involved requires their interaction. Assuming Client is in state 1, hence in trap *requestMachine* as the value of *c* doesn't matter for this, as well as in phase *NotTaking*, then Client is willing to input the value m of the ticket as offered by the machine. But, a priori this value is not known to the client. Therefore, the value is abstracted away by the summation sum m:Nat over all possible values for m. Upon synchronization with the machine process the actual value for m will be handed over. However, this can only happen if the interacting *Machine* process has reached the proper trap in the proper phase regarding the corresponding role.

To enforce synchronization of processes, mCRL2 provides the allow-and-communicate mechanism. Once all processes have been specified (Client, Machine, Baker, and Display) the so-called initial process is given. We have chosen to analyze a typical situation of three clients in combination with one machine, one baker, and one display:

```
allow( {
  getTicket, showTicket, clarify, thankLeave,
    ...
  CM1, useDone, providingDone,
    ... },
comm( {
  requestMachine | readyToProvide -> CM1 ,
    ... },
Client(0,0,NotTaking,NotServed) ||
Client(0,0,NotTaking,NotServed) ||
Client(0,0,NotTaking,NotServed) ||
Mach(0,1,Available) ||
Baker(0,0,NotUsing,FinishingHelp) ||
Display(0,1,Offering) ) )
```

The crucial point is, the synchronized execution of the actions requestMachine(n) by Client and readyToProvide(n) by Machine, for the same value n, will be represented by the execution of the action CM1(n) of the overall system. On top of this, for all n, the action CM1 is allowed to be executed, as mentioned in the list of allowed actions, but the action requestMachine and the action readyToProvide on their own are not, since they are deliberately missing from the list of allowed actions. Thus, a requestMachine or readyToProvide cannot happen alone, but combined into the action CM1 only, provided the actions carry the same value for their parameter. Since, by the sum construction the client is willing to perform requestMachine(m) for each value of m, it can match the specific value for m offered by the machine in readyToProvide(m). This way, for this basic case, passing of parameter values from one process to another is achieved.

In general, a Paradigm model $(C_h)_{h \in H}$ will be encoded in mCRL2 as the parallel composition of $\#H$ processes, with $\#H$ the number of elements of the index set H. For a process C_h we have in its encoding, on the one hand, a parameter st of type Nat enumerating the set of states S_h and parameters $\mathtt{d}_1, \ldots, \mathtt{d}_{n_h}$ of properly chosen built-in or user-defined type to represent the extended state of C_h, and, on the other hand, parameters $\mathtt{ph}_1, \ldots, \mathtt{ph}_{n_h}$ for each of the roles, each of type specifically introduced for the roles. The actions of the processes are either local actions, from the respective action sets A, together with action corresponding to traps of the various roles of the components. With the combined use of allow and communication operators synchronization of traps can be enforced.

A local transition $\langle s_h, e_h, \psi_h \rangle_{h \in H} \xrightarrow{a} \langle s'_h, e'_h, \psi_h \rangle_{h \in H}$, say with $e_h = (e_{h,j})_{j=1}^{m_h}$, is easy to handle. We only need to verify that the transition $\langle s_{h_0}, e_{h_0} \rangle \xrightarrow{a} \langle s'_{h_0}, e'_{h_0} \rangle$ for the active component h_0 is admitted by the phases in ψ_{h_0} (see Definition 3). That other processes remain unchanged is implied by the interleaving of the processes. Thus, for each transition $\langle s_{h_0}, e_{h_0} \rangle \xrightarrow{a} \langle s'_{h_0}, e'_{h_0} \rangle$ in T_h we incorporate for `mCRL2` process `C_h` the non-deterministic branch

```
( ( st == s_h ) &&
  ( d_1 == e_h,1 ) && ... && ( d_m_h == e_h,m(h) ) &&
    ( st in [ list of admitting phases role 1] ) && ... &&
      ( st in [ list of admitting phases role n(h) ] ) ) ->
        a . C_h(s'_h,e'_h,1,...,e'_h,m_h,ph_1,...,ph_n_h)
```

and add the action `a` to the set of actions of the allow operator enclosing the parallel composition of components.

A consistency rule $\gamma = (\,C_r(\Phi_r) : \varphi_r(e_r) \xrightarrow{\theta_r} \varphi'_r(e'_r)\,)_{r \in R}$, say with $e_r = (e_{r,j})_{j=1}^{m_r}$ and $e'_r = (e'_{r,j})_{j=1}^{m_r}$, is distributed over all components and roles involved. For each index r in R, we include a non-deterministic branch for process C_r and role Φ_r in the `mCRL2` process `C_r`.

```
( ( st in [ states for trap theta_r ] ) &&
  ( d_1 == e_h,1 ) && ... && ( d_m_h == e_h,m_h ) &&
    ( ph_i(r) == phase_phi_r_of_Phi_r ) ) ->
      sum w_1:W_1 . ... . sum w_n:W_n .
        theta_r(w_1,...,w_n,expr_1,...,expr_n) .
          C_r(st,e'_r,1,...,e'_r,m_r,...,phase_phi'_r_of_Phi_r,...)
```

The summations `sum w_1:W_1` to `sum w_n:W_n` abstract away the `n-1` groups of variables of the components other the component C_r itself (although this cannot be read off from the notation above). Thus, of the variable groups `w_1` to `w_n` only `w_r` is not bound by a summation. The expressions `expr_1` to `expr_n`, built-up from standard constructs and possibly all of the variables in the `n` groups `w_1` to `w_n`, are the expressions as occurring in the $\#R$ righthand-sides of the consistency rule. For a successful interaction it is required that all parties agree on the values of the parameters and expressions involved. By taking the sum over all possible (potentially infinitely many) values the process `C_r` leaves it totally to the other components to decide on the values of their variables, if occurring at all. Moreover, if $\#R > 1$ we add the communication `theta_1 | ... | theta_#R -> gamma` to the communication operator `com` enclosing the parallel composition of components, but do not include any of the trap actions `theta_1,...,theta_#R`. In case $\#R = 1$, no communication is introduced, but the trap action will be allowed instead.

Some further caution needs to be put in place though, to deal with summations over infinite data types as possibly occurring in the encoding of the consistency rules. In the various analysis steps with the `mCRL2` toolset, in particular statespace generation, the tools may hang because of infinite branching. For Paradigm with data, in concrete situations, simplifications to the coding are applied for variables whose actual value is not used. There are two flavors of this: comparison of an expression involving the variable to an expression involving another (as for the ticket number of the client and the baker in lines 21, 24, and 27), or when the variable doesn't occur at all in the expressions at the righthand-side of the the consistency rule. As illustration of the latter situation, the encoding for the *Machine* process for the consistency rule (CM1) reads

```
( ( st in [ 0 ] ) && ( gt_ph == Available ) ) ->
  readyToProvide(m) . Machine(st,m,Unavailable)
```

where no abstraction of the variable `c` of `Client` is needed. The machine just provides a ticket number, in our modeling, independent of the actual value of `c`.

5 Concluding Remarks

We have shown, with the IWIM model in mind, how the coordination modeling language Paradigm can be extended to deal with data. The present set-up is relatively liberal in the use of variables, although in concrete modeling situations a relatively small number of patterns of data flow among interacting components seem to suffice. Further investigation needs to reveal if this allows for a simplification of the consistency rule format and the associated closedness requirements both for sets of consistency rules as well as for the restriction on updates of variables within a trap.

Currently, for formal analysis using the `mCRL2` toolset, the encoding needs to be tailored to avoid infinite branching during statespace generation. The toolset provides a number of tools, e.g. `lpssumelm` and `lpsconstelm`, that manipulate intermediate artifacts (in so-called linear process specification or lps format) to reduce the specification, leading to smaller and hence more amenable verification problems. It is a topic of further research to develop sum elimination techniques specifically targeting the encoding of consistency rules discussed in this paper.

Application areas for Paradigm with data include the modeling of services, where both coordination and data play a prominent role, as well as the analysis of security protocols. Dissertational work by the second author is underway dealing with the modeling with Paradigm of anonymous networking and internet voting.

Acknowledgments. We are grateful to Farhad Arbab for the many conversations on topics in coordination, computer science, research, and life in general and are looking forward to the undoubtedly many discussions still to come. We thank the anonymous reviewers for their helpful comments.

A Complete `mCRL2` Specification of the Bakery Example

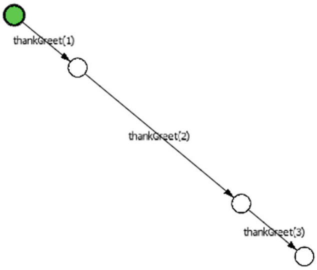
This appendix contains the `mCRL2` code of the bakery example of Sect. 3.

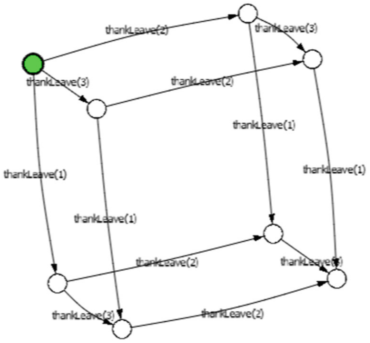
```
sort

  %% role NeedTicket of Client
     NeedTicketPhase = struct NotTaking | Taking ;
  %% role NeedService of Client
     NeedServicePhase = struct NotServed| HavingTurn ;

  %% role GiveTicket of Machine
     GiveTicketPhase = struct Available | Unavailable ;

  %% role ShowNumber of Display
     ShowNumberPhase = struct Offering | Passive ;

  %% role NeedTurnNumber of Baker
     NeedTurnNumberPhase = struct NotUsing | Usage ;
  %% role NeedNextClient of Baker
     NeedNextClientPhase = struct FinishingHelp | StartingHelp | Helping ;

act

  %% Client actions
     getTicket ; showTicket, clarify, thankLeave : Nat ;
  %% Client traps
     requestMachine : Nat ; useDone ;
     requestBaker1, requestBaker2 : Nat ; ordered : Nat ;

  %% Machine actions
     incr, provide ;
  %% Machine traps
     readyToProvide : Nat ; providingDone ;

  %% Display actions
     lightUp, lightDown ;
  %% Display traps
     nextNumber : Nat ; ready ;

  %% Baker actions
     incrDisplay ; welcome, help, thankGreet : Nat ;
  %% Baker traps
     requestNextNumber : Nat ; done ;
     idle, started : Nat ; served : Nat ;


  %% interactions
     CM1 : Nat ;
     BD1 : Nat ;
     BC1, BC2, BC3 : Nat ;

proc
  Client(st:Nat, c:Nat, nt_ph:NeedTicketPhase, ns_ph:NeedServicePhase) =

  %% local STD

    %% 0 -> 1
    ( ( st == 0 ) && ( nt_ph in [ NotTaking ] ) && ( ns_ph in [ NotServed ] ) ) ->
      getTicket . Client(1,c,nt_ph,ns_ph) +

    %% 1 -> 2
    ( ( st == 1 ) && ( nt_ph in [ Taking ] ) && ( ns_ph in [ NotServed ] ) ) ->
      showTicket(c) . Client(2,c,nt_ph,ns_ph) +

    %% 2 -> 3
    ( ( st == 2 ) && ( nt_ph in [ NotTaking, Taking ] ) && ( ns_ph in [ HavingTurn ] ) ) ->
```

```
        clarify(c) . Client(3,c,nt_ph,ns_ph) +

    %% 3 -> 4
    ( ( st == 3 ) && ( nt_ph in [ NotTaking, Taking ] ) && (ns_ph in [ HavingTurn ] ) ) ->
      thankLeave(c) . Client(4,c,nt_ph,ns_ph) +


  %% role NeedTicket

    %% rule (CM1)
       ( ( st in [ 1 ] ) && ( nt_ph == NotTaking ) ) ->
         sum m:Nat . requestMachine(m) . Client(st,m,Taking,ns_ph) +

    %% rule (CM2)
       ( ( st in [ 2, 3, 4 ] ) && ( nt_ph == Taking ) ) ->
         useDone . Client(st,c,NotTaking,ns_ph) +


  %% role NeedService

    %% rule (BC1)
       ( ( st in [ 2 ] ) && ( ns_ph == NotServed ) ) ->
         requestBaker1(c) . Client(st,c,nt_ph,NotServed) +

    %% rule (BC2)
       ( ( st in [ 2 ] ) && ( ns_ph == NotServed ) ) ->
         requestBaker2(c) . Client(st,c,nt_ph,HavingTurn) +

    %% rule (BC3)
       ( ( st in [ 3, 4 ] ) && ( ns_ph == HavingTurn ) ) ->
         ordered(c) . Client(st,c,nt_ph,NotServed) ;

proc
  Machine( st:Nat, m:Nat, gt_ph:GiveTicketPhase ) =

  %% local STD

    %% 0 -> 1
    ( ( st in [ 0 ] ) && ( gt_ph in [ Unavailable ] ) ) ->
      provide . Machine(1,m,gt_ph) +

    %% 1 -> 0
    ( ( st == 1 ) && ( gt_ph in [ Available ] ) ) ->
      incr . Machine(0,m+1,gt_ph) +


  %% role GiveTicket

    %% rule (CM1)
       ( ( st in [ 0 ] ) && ( gt_ph == Available ) ) ->
         readyToProvide(m) . Machine(st,m,Unavailable) +

    %% rule (CM3)
       ( ( st in [ 1 ] ) && ( gt_ph == Unavailable ) ) ->
         providingDone . Machine(st,m,Available) ;

proc
  Baker(st:Nat, b:Nat, ntn_ph:NeedTurnNumberPhase, nnc_ph:NeedNextClientPhase) =

  %% local STD

    %% 0 -> 1
    ( ( st == 0 ) && ( ntn_ph in [ Usage ] ) && (nnc_ph in [ FinishingHelp ] ) ) ->
      incrDisplay . Baker(1,b,ntn_ph,nnc_ph) +

    %% 1 -> 2
    ( ( st == 1 ) && ( ntn_ph in [ Usage ] ) && (nnc_ph in [ StartingHelp ] ) ) ->
      welcome(b) . Baker(2,b,ntn_ph,nnc_ph) +
    %% 2 -> 3
    ( ( st == 2 ) && ( ntn_ph in [ Usage ] ) && (nnc_ph in [ Helping ] ) ) ->
      help(b) . Baker(3,b,ntn_ph,nnc_ph) +

    %% 3 -> 0
    ( ( st == 3 ) && ( ntn_ph in [ NotUsing ] ) && (nnc_ph in [ FinishingHelp ] ) ) ->
      thankGreet(b) . Baker(0,b,ntn_ph,nnc_ph) +

  %% role NeedTurnNumber

    %% rule (BD1)
       ( ( st in [ 0 ] ) && ( ntn_ph == NotUsing ) ) ->
         sum d:Nat . requestNextNumber(d) . Baker(st,d,Usage,nnc_ph) +

    %% rule (BD2)
       ( ( st in [ 3 ] ) && ( ntn_ph == Usage ) ) ->
         done . Baker(st,b,NotUsing,nnc_ph) +

  %% role NeedNextClient

    %% rule (BC1)
       ( ( st in [ 1 ] ) && ( nnc_ph == FinishingHelp ) ) ->
         idle(b) . Baker(st,b,ntn_ph,StartingHelp) +

    %% rule (BC2)
       ( ( st in [ 2 ] ) && ( nnc_ph == StartingHelp ) ) ->
         started(b) . Baker(st,b,ntn_ph,Helping) +

    %% rule (BC3)
       ( ( st in [ 3 ] ) && ( nnc_ph == Helping ) ) ->
         served(b) . Baker(st,b,ntn_ph,FinishingHelp) ;

proc
  Display( st:Nat, d:Nat, sh_ph:ShowNumberPhase ) =

  %% local STD

    %% 0 -> 1
    ( ( st in [ 0 ] ) && ( sh_ph in [ Passive ] ) ) ->
      lightDown . Display(1,d,sh_ph) +

    %% 1 -> 0
    ( ( st == 1 ) && ( sh_ph in [ Offering ] ) ) ->
      lightUp . Display(0,d,sh_ph) +

  %% role ShowingNumber

    %% rule (BD1)
       ( ( st in [ 0 ] ) && ( sh_ph == Offering ) ) ->
         nextNumber(d) . Display(st,d,Passive) +

    %% rule (BD3)
       ( ( st in [ 1 ] ) && ( sh_ph == Passive ) ) ->
         ready . Display(st,d+1,Offering) ;

init
  hide( {
    %% Client actions
       getTicket, showTicket, clarify, thankLeave,

    %% Machine actions
       incr, provide,

    %% Baker actions
       incrDisplay, welcome, help,
       %% thankGreet,
      %% Display actions
         lightUp, lightDown,

      %% interactions
         CM1, useDone, providingDone,
         BD1, done, ready,
         BC1, BC2, BC3 },

    allow( {
      %% Client actions
         getTicket, showTicket, clarify, thankLeave,

      %% Machine actions
         incr, provide,

      %% Baker actions
         incrDisplay, welcome, help, thankGreet,

      %% Display actions
         lightUp, lightDown,


      %% interactions
         CM1, useDone, providingDone,
         BD1, done, ready,
         BC1, BC2, BC3 },

    comm( {
      %% Client-Machine interaction
         requestMachine | readyToProvide -> CM1 ,
      %% Baker-Display interaction
         requestNextNumber | nextNumber -> BD1 ,
      %% Baker-Client interaction
         idle | requestBaker1 -> BC1 ,
         started | requestBaker2 -> BC2 ,
         served | ordered -> BC3 },

    Client(0,0,NotTaking,NotServed) ||
    Client(0,0,NotTaking,NotServed) ||
    Client(0,0,NotTaking,NotServed) ||
    Machine(0,1,Available) ||
    Baker(0,0,NotUsing,FinishingHelp) ||
    Display(0,1,Offering)
      ))) ;
```

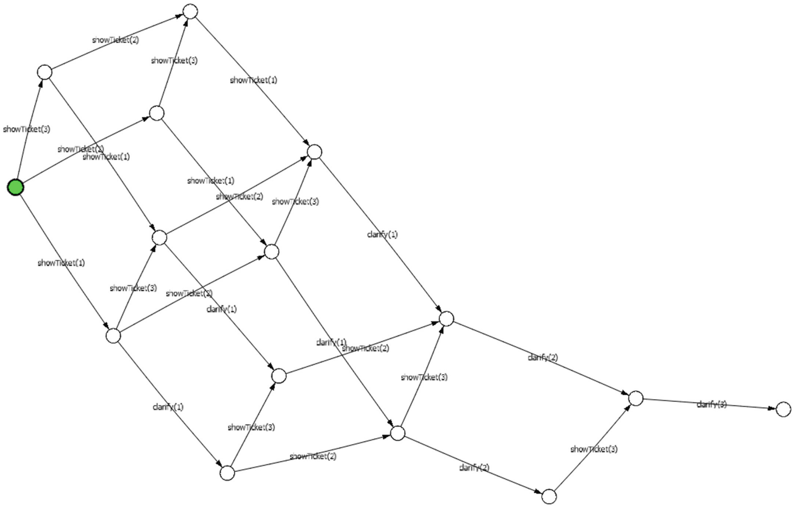
B Reduced LTS

Labeled transition system for the specification of Appendix A with only actions `thankGreet` by the `Baker` process shown on the left. It validates that all three clients are served, assuming an atomic `welcome-help-thankGreet` sequences of actions, and are served in order of their tickets. On the right, the labeled transition system for the specification of Appendix A with only actions `thankLeave` by the `Client` processes shown. It validates that all three clients are served, but they can (and will) leave in any order.





Labeled transition system for the specification of Appendix A with only actions `showTicket` and `clarify` by the `Client` processes shown. It validates that all three clients can raise their ticket independently, since admitted local behavior can be executed asynchronously from other component behavior, but are served in order of their ticket.



References

1. Andova, S., Groenewegen, L.P.J., de Vink, E.P.: Dynamic consistency in process algebra: from Paradigm to ACP. Sci. Comput. Program. **76**(8), 711–735 (2011)
2. Andova, S., Groenewegen, L.P.J., de Vink, E.P.: Dynamic adaptation with distributed control in Paradigm. Sci. Comput. Program. **94**, 333–361 (2014)
3. Arbab, F.: The IWIM model for coordination of concurrent activities. In: Ciancarini, P., Hankin, C. (eds.) COORDINATION 1996. LNCS, vol. 1061, pp. 34–56. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61052-9_38
4. Arbab, F., Herman, I., Spilling, P.: An overview of Manifold and its implementation. Concurr. - Pract. Exp. **5**(1), 23–70 (1993)
5. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. **61**(2), 75–113 (2006)

6. Banach, R., Arbab, F., Papadopoulos, G.A., Glauert, J.R.W.: A multiply hierarchical automaton semantics for the IWIM coordination model. J. Univers. Comput. Sci. **9**(1), 2–33 (2003)
7. Bonsangue, M.M., Arbab, F., de Bakker, J.W., Rutten, J.J.M.M., Secutella, A., Zavattaro, G.: A transition system semantics for the control-driven coordination language Manifold. Theoret. Comput. Sci. **240**(1), 3–47 (2000)
8. Cranen, S., Groote, J.F., Keiren, J.J.A., Stappers, F.P.M., de Vink, E.P., Wesselink, W., Willemse, T.A.C.: An overview of the mCRL2 toolset and its recent advances. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 199–213. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_15
9. Groenewegen, L., de Vink, E.: Operational semantics for coordination in Paradigm. In: Arbab, F., Talcott, C. (eds.) COORDINATION 2002. LNCS, vol. 2315, pp. 191–206. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46000-4_20
10. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014)
11. Jongmans, S.-S.T.Q., Arbab, F.: Global consensus through local synchronization: a formal basis for partially-distributed coordination. Sci. Comput. Program. **115–116**, 199–224 (2016)

Author Index