ICE 2023 Pre-Proceedings

Clément Aubert Cinzia Di Giusto Simon Fowler Larisa Safina

8th July 2023

This document contains the *informal* pre-proceedings of the 16th Interaction and Concurrency Experience (ICE 2023). The post-proceedings will be published on EPTCS.

Some of the slides can be found on the workshop's website.

Contents

Research Challenges in Orchestration Synthesis

Davide Basile Maurice H. ter Beek
Formal Methods and Tools lab, ISTI–CNR, Pisa, Italy
{davide.basile,maurice.terbeek}@isti.cnr.it

Contract automata allow to formally define the behaviour of service contracts in terms of service offers and requests, some of which are moreover optional and some of which are necessary. A composition of contracts is said to be in agreement if all service requests are matched by corresponding offers. Whenever a composition of contracts is not in agreement, it can be refined to reach an agreement using the orchestration synthesis algorithm. This algorithm is a variant of the synthesis algorithm used in supervisory control theory and it is based on the fact that optional transitions are controllable, whereas necessary transitions are at most semi-controllable and cannot always be controlled. In fact, the resulting orchestration is such that as much of the behaviour in agreement is maintained. In this paper, we discuss recent developments of the orchestration synthesis algorithm for contract automata. Notably, we present a refined notion of semi-controllability and compare it with the original notion by means of examples. We then discuss the current limits of the orchestration synthesis algorithm and identify a number of research challenges together with a research roadmap.

1 Introduction

Orchestrations of services describe how control and data exchanges are coordinated in distributed service-based applications and systems. Their principled design is identified in [16] as one of the primary research challenges for the next 10 years, and the Service Computing Manifesto [16] points out that "Service systems have so far been built without an adequate rigorous foundation that would enable reasoning about them" and, moreover, that "The design of service systems should build upon a formal model of services".

The problem of synthesising well-behaving orchestrations of services can be viewed as a specific instance of the more general problem of synthesising strategies in games [9, 7]. This can be solved using refined algorithms from supervisory control for discrete event systems [24, 1], which have well-established relationships with reactive systems synthesis [20], parity games [23], automated behaviour composition [21] and automated planning [17].

Contract automata are a specific type of finite state automata that are used to formally define the behaviour of service contracts. These automata express contracts in terms of both offers and requests [10]. When multiple contracts are composed, they are said to be in agreement if all service requests from one contract are matched by another contract's corresponding offers. A composition of contracts that is not in agreement, can automatically be refined to reach an agreement by means of the orchestration synthesis algorithm, which is a variation of the synthesis algorithm used in supervisory control theory. This orchestration synthesis algorithm for contract automata is described in [8, 9].

The classic algorithm for synthesising a most permissive controller distinguishes transitions whose controllability is invariant [24, 1]. In service contracts, instead, the controllability of certain transitions may vary depending on specific conditions on the orchestration of contracts [9]. The contract automata library `CATLib` [5] implements contract automata and their operations (e.g., composition and synthesis). Orchestrations of contract automata abstract from their underlying realisation; an orchestrator is assumed

Submitted to:
ICE 2023

to interact with the services to realise the overall behaviour as prescribed by the orchestration contract. The contract automata runtime environment CARE [6] implements an orchestrator that interprets the synthesised orchestration to coordinate the services, where each service is implementing a contract. Thus, CARE is explicating the low-level interactions that are abstracted in contract automata orchestrations. Notably, one aspect that is abstracted in contract automata and concretised at the implementation level is that of selecting the next transition to execute in the presence of choice. In [6], different implementations are proposed based on whether services may participate externally or internally in a choice.

This paper delves into challenges and research issues for orchestration synthesis of contract automata, given the latest developments in this field. In particular, we start by refining the current definition of semi-controllability to consider the aforementioned possible realisations of choices defined in [6]. We provide several examples to illustrate the differences between the refined definition and the original definition. The various definitions of semi-controllability lead to different sets of contract automata orchestrations, which we present in Figure 3 together with an example for each level of the orchestration hierarchy depicted. This allows us to highlight the unique characteristics of each level and to identify current issues in synthesising orchestrations of contract automata using these examples. Based on the issues presented, we then outline future research challenges in the orchestration synthesis of contract automata and a research roadmap to address them.

Related Work At last year's ICE 2022 workshop, the compositionality of communicating finite state machines (CFSM) with asynchronous semantics was discussed in [3]. Also contract automata are composable, enabling the modelling of systems of systems. Moreover, under certain specific conditions that were presented at the 2014 edition of ICE [11, 12], an orchestration of contract automata can be translated into a choreography of synchronous or asynchronous CFSM. The relation between multiparty session types and CFSM is discussed in [27]. Therefore, contract automata can be related to multiparty session types by exploiting their common relation with CFSM [11, 12, 27].

The contract automata approach is closer to [22], in which behavioural types are expressed as finite state automata of Mungo, called typestates [25]. Similarly to CARE, the runtime environment for contract automata [6], in Mungo finite state automata are used as behaviour assigned to Java classes (one automaton per class), with transition labels corresponding to methods of the classes. A tool to translate typestates into automata was presented at ICE 2020 [26]. CATApp, a graphical front-end tool for designing contract automata, is available in [19]. A tool similar to Mungo is JaTyC (Java Typestate Checker) [2].

The refined definition of semi-controllability presented in this paper closely aligns with the notion of weak receptiveness in team automata [14, 15]. However, the challenges addressed in this paper are primarily related to the problem of synthesising an orchestration of services and as such are not directly relevant to team automata.

Differently from the semi-controllability for orchestrations, a distinct notion of semi-controllability has been studied in [9, 4] for choreographies of services. Finally, while a runtime environment for the orchestration of services has been proposed in [6], this has yet to be realised for the case of choreographies, which could result in improvements in the notion of semi-controllability for choreographies.

Outline We start by providing some background on contract automata and orchestration synthesis in Section 2. We introduce a refined notion of semi-controllability in Section 3. In Section 4, we present several research challenges for orchestration synthesis of contract automata. We conclude in Section 5.

2 Background

We will begin by formally introducing contract automata and their synthesis operation. Contract automata are a type of finite state automata that use a partitioned alphabet of actions. A Contract Automaton (CA) can model either a single service or a composition of multiple services that perform actions. The number of services in a CA is known as its rank. If the rank of a CA is 1, then the contract is referred to as a principal (i.e., a single service).

The labels of a CA are vectors of atomic elements known as actions. Actions are categorised as either requests (prefixed by ?), offers (prefixed by !), or idle actions (represented by a distinguished symbol $-$). Requests and offers belong to the sets R and O, respectively, and they are pairwise disjoint. The states of a CA are vectors of atomic elements known as basic states. Labels are restricted to requests, offers or matches. In a request (resp. offer) label there is a single request (resp. offer) action and all other actions are idle. In a match label there is a single pair of request and offer actions that match, and all other actions are idle. The length of the vectors of states and labels is equal to the rank of the CA. For example, the label $[!a, ?a, -, -]$ is a match where the request action $?a$ is matched by the offer action $!a$, and all other actions are idle. Note the difference between a request label (e.g., $[?a, -]$) and a request action (e.g., $?a$). A transition may also be called a request, offer or match according to its label. Figure 4 depicts three principal contracts, whilst Figure 5 depicts a contract of rank 3.

The goal of each service is to reach an accepting (*final*) state such that all its request (and possibly offer) actions are matched. Transitions are equipped with *modalities*, i.e., *necessary* ($\Box$) and *optional* ($\circ$) transitions, respectively [1]. Optional transitions are controllable, whereas necessary transitions can be uncontrollable (called *urgent* necessary transitions) or semi-controllable (called *lazy* necessary transitions). The resulting formalism is called *Modal Service Contract Automata* (MSCA). In the following definition, given a vector $\vec{a}$, its ith element is denoted by $\vec{a}_{(i)}$.

Definition 1 (MSCA)**.** *Given a finite set of states $\mathcal{Q} = \{q_1, q_2, \ldots\}$, an MSCA $\mathcal{A}$ of rank n is a tuple $\langle Q, \vec{q}_0, A^r, A^o, T, F\rangle$, with set of states $Q = Q_1 \times \ldots \times Q_n \subseteq \mathcal{Q}^n$, initial state $\vec{q}_0 \in Q$, set of requests $A^r \subseteq \mathsf{R}$, set of offers $A^o \subseteq \mathsf{O}$, set of final states $F \subseteq Q$, set of transitions $T \subseteq Q \times A \times Q$, where $A \subseteq (A^r \cup A^o \cup \{\bullet\})^n$, partitioned into* optional *transitions T° and* necessary *transitions $T^{\Box}$, with $T^{\Box}$ further partitioned into* urgent *necessary transitions $T^{\Box_u}$ and* lazy *necessary transitions $T^{\Box_l}$, and such that given $t = (\vec{q}, \vec{a}, \vec{q}\,') \in T$: i) $\vec{a}$ is either a request, an offer or a match; ii) if $\vec{a}$ is an offer, then $t \in T^{\circ}$; and iii) $\forall i \in 1 \ldots n$, $\vec{a}_{(i)} = \bullet$ implies $\vec{q}_{(i)} = \vec{q}\,'_{(i)}$.*

Composition of services is rendered through the composition of their MSCA models by means of the *composition operator* $\otimes$, which is a variant of a synchronous product. This operator basically interleaves or matches the transitions of the component MSCA, but, whenever two component MSCA are enabled to execute their respective request/offer action, then the match is forced to happen. Moreover, a match involving a necessary transition of an operand is itself necessary. The rank of the composed MSCA is the sum of the ranks of its operands. The vectors of states and actions of the composed MSCA are built from the vectors of states and actions of the component MSCA, respectively. In this paper, we will only consider principal contracts and compositions of principals, which will be automatically refined into orchestrations (as shown in Figure 2). However, it is important to note that contracts can be created by composing contracts with a rank of one or higher.

In a composition of MSCA, typically various properties are analysed. We are especially interested in *agreement*. The property of agreement requires to match all requests, whereas offers can go unmatched.

[1] Originally, in [8], the optional modality was called permitted and denoted with $\diamond$. Since in contract automata the two modalities are a partition, the terminology has been updated to avoid confusion with modal transition systems, where $\Box \subseteq \diamond$.

CA support the synthesis of the most permissive controller (mpc) known from the theory of supervisory control of discrete event systems [24, 18], where a finite state automaton model of a *supervisory controller* is synthesised from given (component) finite state automata that are composed. The synthesised automaton, if successfully generated (i.e., non-empty), is such that it is *non-blocking*, *controllable*, and *maximally permissive*. An automaton is said to be *non-blocking* if, from each state, at least one of the *final states* (distinguished stable states that represent completed 'tasks' [24]) can be reached without passing through so-called *forbidden states*, meaning that there is always a possibility to return to an accepted stable state (e.g., a final state).

The synthesised automaton is said to be *controllable* when only controllable transitions are disabled. Indeed, the supervisory controller is not permitted to directly block uncontrollable transitions from occurring; the controller is only allowed to disable them by preventing controllable actions from occurring. Finally, the fact that the resulting supervisory controller is said to be *maximally permissive* (or least restrictive) means that as much behaviour of the uncontrolled system as possible is present in the controlled system without violating neither the requirements, nor controllability nor the non-blocking condition.

Orchestration Synthesis As stated previously, optional transitions are controllable, whereas necessary transitions can be either uncontrollable (called *urgent*) or semi-controllable (called *lazy*). In the mpc synthesis (implemented in `CATLib` [9, 5]), all necessary transitions are *urgent*, i.e., they are always uncontrollable. This stems from the fact that traditionally uncontrollable transitions relate to an unpredictable environment.

When synthesising an orchestration of services, all necessary transitions are instead *lazy*, i.e., they are *semi-controllable* [8, 9]. A semi-controllable transition t is a transition that is either uncontrollable or controllable according to given conditions. In [9], different conditions are given according to whether the synthesis of an orchestration or a choreography is computed. In this paper, we only consider orchestrations. Below, we denote with $Dangling(\mathcal{A})$ the set of states that are not reachable from the initial state or cannot reach any final state. More in detail, a semi-controllable transition t is controllable if in a given portion $\mathcal{A}'$ of $\mathcal{A}$ there exists a semi-controllable match transition t', with source and target states not dangling, such that in both t and t' the *same* service, in the *same* local state, does the *same* request. Otherwise, t is uncontrollable.

Definition 2 (Controllability)**.** *Let $\mathcal{A}$ be an MSCA and let $t = (\vec{q}_1, \vec{a}_1, \vec{q}_1{}') \in T_{\mathcal{A}}$. Then:*

- *if $t \in T_{\mathcal{A}}^{\circ}$, then t is* controllable *(in $\mathcal{A}$);*
- *if $t \in T_{\mathcal{A}}^{\square u}$, then t is* uncontrollable *(in $\mathcal{A}$);*
- *if $t \in T_{\mathcal{A}}^{\square l}$, then t is* semi-controllable *(in $\mathcal{A}$).*

Moreover, given $\mathcal{A}' \subseteq \mathcal{A}$, if t is semi-controllable and $\exists t' = (\vec{q}_2, \vec{a}_2, \vec{q}_2{}') \in T_{\mathcal{A}'}^{\square}$ in $\mathcal{A}'$ such that $\vec{a}_2$ is a match, $\vec{q}_2, \vec{q}_2{}' \notin Dangling(\mathcal{A}')$, $\vec{q}_{1(i)} = \vec{q}_{2(i)}$, and $\vec{a}_{1(i)} = \vec{a}_{2(i)} = ?a$ for some $i \in 0 \ldots rank(\mathcal{A})$, then t is controllable *in $\mathcal{A}'$ (via t'). Otherwise, t is* uncontrollable *in $\mathcal{A}'$.*

The interpretation of optional/controllable and urgent/uncontrollable transitions is standard [24, 18]. In the upcoming section, we will delve into different understandings and interpretations of the concept of semi-controllability. We remark that the orchestration synthesis defined below does not support urgent transitions. The orchestration synthesis, as defined below, involves an iterative refinement of the initial automaton $\mathcal{A}$ (i.e., the composition of contracts). In each iteration, transitions are selectively pruned, and a set R of forbidden states is updated accordingly. A transition t is pruned under one of two conditions: if it is a request (thus violating the agreement property enforced by the orchestration), or if the target

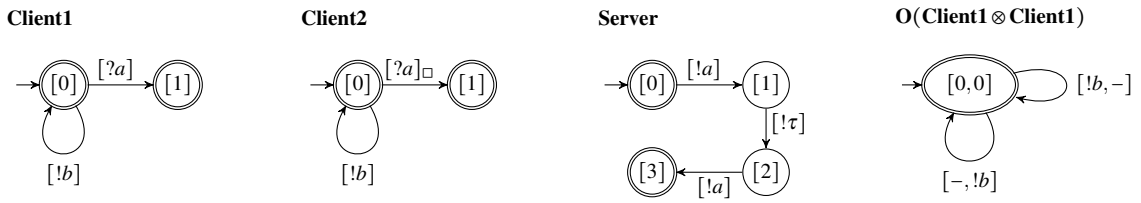

Figure 1: Contracts of **Client1**, **Client2** and **Server**, and orchestration $\mathbf{O}(\mathbf{Client1} \otimes \mathbf{Client1})$

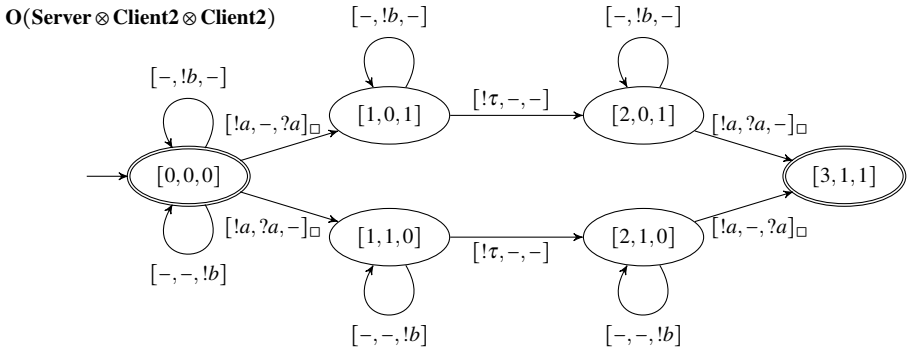

Figure 2: Orchestration $\mathbf{O}(\mathbf{Server} \otimes \mathbf{Client2} \otimes \mathbf{Client2})$

state of t belongs to the set R computed up to that point. During the first iteration, all request transitions, including both lazy and optional ones, are pruned.

In Definition 2, the automaton $\mathcal{A}'$ represents an intermediate refinement of $\mathcal{A}$ (the starting composition) which occurs during an iteration of the synthesis process. Intuitively, the semi-controllable transition t of $\mathcal{A}$ is controllable in $\mathcal{A}'$ because there is another transition t' in $\mathcal{A}'$ matching the same request from the same service in the same state. Otherwise, if there is no such transition t' in $\mathcal{A}'$, then t is uncontrollable. Put differently, the controllability of t in $\mathcal{A}'$ relies on the presence of a corresponding transition t' within $\mathcal{A}'$ itself. If such a matching transition t' does not exist in $\mathcal{A}'$, then t is deemed uncontrollable.

Note that in Definition 2, it is not required for t and t' to be distinct. This implies that during the synthesis process, a semi-controllable match transition t can switch from being controllable to uncontrollable only after it has been pruned in a previous iteration. To clarify further, a semi-controllable match transition t can switch its controllability status from controllable to uncontrollable only when t is absent in the sub-automaton $\mathcal{A}'$ during the current iteration. If t is present in $\mathcal{A}'$ (i.e., it has not been pruned thus far), then, according to Definition 2, t is considered semi-controllable and controllable within $\mathcal{A}'$ via t itself. It is important to note that these considerations are applicable only if t is a match. Additionally, it is never the case that a semi-controllable transition t switches from uncontrollable to controllable since transitions are only removed during the synthesis process and are never added back.

The set R of forbidden states is updated at each iteration by adding source states of uncontrollable transitions and dangling states of the refined automaton in the current iteration. Specifically, when the synthesis process eliminates all transitions t' that satisfy the conditions for rendering the semi-controllable transition t controllable via t', then t becomes uncontrollable within the sub-automaton in the current iteration. It is worth noting that even if t was previously pruned in an earlier iteration, its source state $\vec{q}_1$ might still be reachable in the sub-automaton of the current iteration. Consequently, $\vec{q}_1$ is added

to the set R. In the subsequent iteration, all transitions with target state $\vec{q}_1$ will be pruned. This pruning of transitions whose target is $\vec{q}_1$ can potentially render another previously pruned semi-controllable transition as uncontrollable, thereby adding its source state to the updated set R. This refinement process continues until no further transitions are pruned, and no additional states are added to R. The resulting refined automaton obtained at the end of the synthesis process represents the orchestration automaton.

The algorithm for synthesising an orchestration enforcing agreement of MSCA is defined below.

Definition 3 (MSCA orchestration synthesis)**.** *Let $\mathcal{A}$ be an MSCA and let $\mathcal{K}_0 = \mathcal{A}$ and $R_0 = Dangling(\mathcal{K}_0)$. We let the* orchestration synthesis function *$f_o : MSCA \times 2^Q \to MSCA \times 2^Q$ be defined as follows:*

$$f_o(\mathcal{K}_{i-1}, R_{i-1}) = (\mathcal{K}_i, R_i), \text{ with}$$
$$T_{\mathcal{K}_i} = T_{\mathcal{K}_{i-1}} \setminus \{ (\vec{q} \to \vec{q}') = t \in T_{\mathcal{K}_{i-1}} \mid (\vec{q}' \in R_{i-1} \vee t \text{ is a request}) \}$$
$$R_i = R_{i-1} \cup \{ \vec{q} \mid (\vec{q} \to) \in T_{\mathcal{A}}^{\Box_l} \text{ is uncontrollable in } \mathcal{K}_i \} \cup Dangling(\mathcal{K}_i)$$

The orchestration automaton is obtained from the fixpoint of the function f_o. In the rest of the paper, if not stated otherwise, all necessary transitions in the examples are lazy (cf. Definition 1); for brevity and less cluttering in the figures, we denote them by $\Box$ rather than $\Box_l$.

Example 1. We provide an illustrative example to underline the differences between optional transitions, urgent necessary transitions and lazy necessary transitions. Figure 1 shows two client contracts and a server contract. Firstly, we discuss the difference between optional and necessary transitions. When all actions of the client contract are optional (**Client1**), there exists an orchestration of the composition of two **Client1** contracts, also depicted in Figure 1 ($\mathbf{O}(\mathbf{Client1} \otimes \mathbf{Client1})$). Indeed the (transition labelled with the) request $?a$ is optional and can be removed to obtain the orchestration. If instead the request $?a$ was necessary (**Client2**), then there would be no orchestration for the composition of two **Client2** contracts, because the necessary request is never matched by a corresponding offer.

To illustrate the distinction between urgent and lazy necessary transitions, we consider also the server contract shown in Figure 1. If we were to employ the traditional mpc synthesis, the clients' necessary requests ($?a$) would be treated as urgent. In such a scenario, the orchestration of the composition between two clients and the server (generated using the mpc synthesis algorithm) would be empty, indicating that no feasible orchestration exists.

However, if the clients' necessary requests ($?a$) are considered lazy instead, an orchestration of the composition between the server and the two clients can be achieved (computed using the orchestration synthesis). This orchestration is depicted in Figure 2. In this case, the clients take turns fulfilling their lazy necessary requests. This alternating behaviour is not possible when the necessary requests are urgent.

The orchestration in Figure 2 is obtained after three iterations of the algorithm specified in Definition 3. Initially, $\mathcal{K}_0 = \mathcal{A} = \mathbf{Server} \otimes \mathbf{Client2} \otimes \mathbf{Client2}$ and $R_0 = Dangling(\mathcal{A}) = \varnothing$.

With respect to the orchestration in Figure 2, the automaton $\mathcal{A}$ contains four additional transitions that are $t_1 = [1,0,1] \xrightarrow{[-,?a,-]_\Box} [1,1,1]$, $t_2 = [1,1,0] \xrightarrow{[-,-,?a]_\Box} [1,1,1]$, $t_3 = [1,1,1] \xrightarrow{[!\tau,-,-]} [2,1,1]$ and $t_4 = [2,1,1] \xrightarrow{[!a,-,-]} [3,1,1]$. In the first iteration, t_1 and t_2 are removed from $\mathcal{K}_1$ because they are request transitions. We have $T_{\mathcal{K}_1} = T_{\mathcal{K}_0} \setminus \{t_1, t_2\}$. Since there are no forbidden states, these are the only two transitions that are removed during the first iteration.

Concerning the set of forbidden states R_1, we have that $t_1 \in T_{\mathcal{A}}^{\Box_l}$ is controllable in $\mathcal{K}_1$ via transition $[0,0,0] \xrightarrow{[a!,a?,-]_\Box} [1,1,0]$. Similarly, $t_2 \in T_{\mathcal{A}}^{\Box_l}$ is controllable in $\mathcal{K}_1$ via $[0,0,0] \xrightarrow{[a!,-,a?]_\Box} [1,0,1]$. Hence, the source states of t_1 and t_2 will not be added to R_1. Concerning the set $Dangling(\mathcal{K}_1)$, state $[1,1,1]$ was the target of only t_1 and t_2. Moreover, state $[2,1,1]$ was the target of only t_3. Therefore, states $[1,1,1]$ and $[2,1,1]$ are unreachable in $\mathcal{K}_1$. We have that $R_1 = Dangling(\mathcal{K}_1) = \{[1,1,1],[2,1,1]\}$. In the subsequent iteration $i = 2$, since transition t_3 has target in R_1, we have $T_{\mathcal{K}_2} = T_{\mathcal{K}_1} \setminus \{t_3\}$, whilst $R_2 = R_1$.

Finally, we reach the fixpoint at iteration $i = 3$, where $T_{\mathcal{K}_3} = T_{\mathcal{K}_2}$ and $R_3 = R_2$. The finalising operations for obtaining the orchestration $\mathbf{O}$ in Figure 2 from the fixpoint $\mathcal{K}_3$ consist in removing the states in R_3, i.e., $Q_{\mathbf{O}} = Q_{\mathcal{K}_3} \setminus R_3$, and removing the remaining unreachable transitions in $\mathcal{K}_3$. In this case, transition $t_4 \in T_{\mathcal{K}_3}$ is removed from the orchestration, i.e., $T_{\mathbf{O}} = T_{\mathcal{K}_3} \setminus \{t_4\}$.

In the subsequent section, we will delve deeper into additional details and interpretations regarding the semi-controllable transitions of contract automata.

3 Refined Semi-Controllability

We start by introducing a refined notion of semi-controllability to be used in the orchestration synthesis, formalised below. After that, we discuss how this refined notion may assist to discard some counter-intuitive orchestrations.

Definition 4 (Refined Semi-Controllability)**.** *Let $\mathcal{A}$ be an MSCA and let $t = (\vec{q}_t, \vec{a}_t, \vec{q}_t{}') \in T^{\Box_l}_{\mathcal{A}}$. Moreover, given $\mathcal{A}' \subseteq \mathcal{A}$, if $\exists t' = (\vec{q}_{t'}, \vec{a}_{t'}, \vec{q}_{t'}{}') \in T^{\Box_l}_{\mathcal{A}'}$ in $\mathcal{A}'$ such that the following hold:*

1. *$\vec{a}_{t'}$ is a match, $\vec{q}_{t'}, \vec{q}_{t'}{}' \notin Dangling(\mathcal{A}')$, $\vec{q}_{t\,(j)} = \vec{q}_{t'\,(j)}$, $\vec{a}_{t\,(j)} = \vec{a}_{t'\,(j)} = ?a$, for some $j \in 0 \ldots rank(\mathcal{A})$; and*
2. *there exists a sequence of transitions $t_0, \ldots, t_n$ of $\mathcal{A}'$ such that $\forall i \in 0 \ldots n$, $t_i = (\vec{q}_i, \vec{a}_i, \vec{q}_i{}')$ and the following hold:*
   - *$\vec{q}_0 = \vec{q}_t$;*
   - *$t_n = t'$;*
   - *$\vec{q}_i, \vec{q}_i{}' \notin Dangling(\mathcal{A}')$; and*
   - *if $i < n$, then $\vec{a}_{i\,(j)} = -$ and $\vec{q}_i{}' = \vec{q}_{i+1}$;*

then t is controllable *in $\mathcal{A}'$ (via t'). Otherwise, t is* uncontrollable *in $\mathcal{A}'$.*

By comparing Definition 2 and Definition 4, we note that only the semi-controllable transitions have been refined, whilst the others are unaltered. Conditions 1 and 2 contain the constraints that are used to decide when a semi-controllable transition is controllable or uncontrollable. The constraints of Condition 1 are also present in Definition 2. The intuition is that a (refined) semi-controllable transition t becomes controllable if (similarly to Definition 2) in a given portion of $\mathcal{A}$, there exists a semi-controllable match transition t', with source and target states not dangling, such that in both t and t' the *same* service, in the *same* local state, does the *same* request. Condition 2 of Definition 4 imposes new further constraints. It requires that t' is *reachable* from the source state of t through a sequence of transitions where the service performing the request is idle.

Consider the Venn diagram in Figure 3. The outermost set *Orchestrations* contains all orchestrations of contract automata that are computed using the notion of semi-controllability of Definition 2. The innermost set *Refined* contains only those orchestrations that are computed using the refined notion of semi-controllability in Definition 4. Intuitively, the refined notion imposes a further constraint on *when* a semi-controllable transition is controllable. As a result, more semi-controllable transitions are uncontrollable than in the previous definition. This explains why *Refined* is contained in *Orchestrations*.

All the examples of semi-controllability available in the literature [13, 8, 9, 6] (e.g., Hotel service) and Figure 2 are orchestrations belonging to the set *Refined* in Figure 3. This means that by updating the notion of semi-controllability, all orchestrations of these examples remain unaltered.

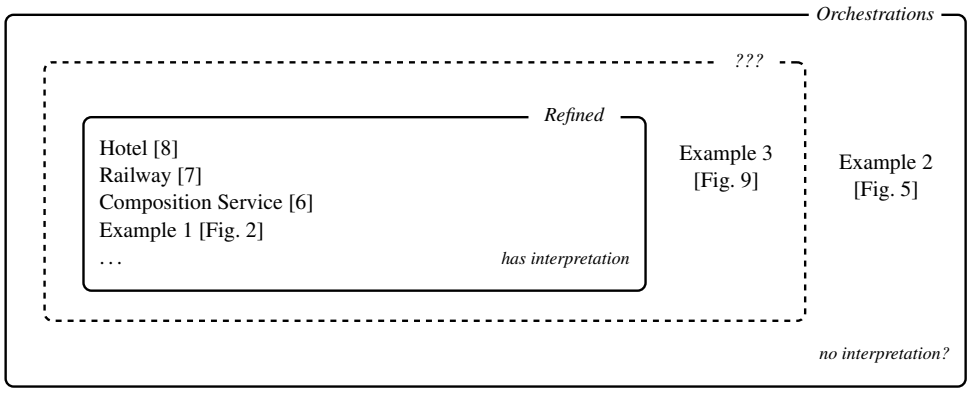

Figure 3: A Venn diagram showing the set of orchestrations of contract automata

Example 2. We now provide an example of an orchestration belonging to *Orchestrations* $\setminus$ *Refined* (cf. Figure 3). We have three principal contracts, namely **A**lice, **B**ob and **C**arl, depicted in Figure 4. The contracts of **B**ob and **C**arl perform two alternative necessary requests. The contract of **A**lice has two branches. In each branch, a request of **B**ob and a request of **C**arl are fulfilled by corresponding offers.

Using the notion of semi-controllability from Definition 2, the synthesis algorithm of Definition 3 takes as input the composed automaton and returns the orchestration of the composition, depicted in Figure 5, which is a contract of rank 3. Indeed, for each necessary request of each service, there *exists* a match transition in the composition where the necessary request is fulfilled by a corresponding offer. In other words, for each necessary request of **B**ob and **C**arl, there exists an execution where the request is matched by a corresponding offer. For example, the composition **A**lice $\otimes$ **B**ob $\otimes$ **C**arl contains the transition $t = [\mathrm{a}_1, \mathrm{b}_0, \mathrm{c}_0] \xrightarrow{[-,?d,-]_\square} [\mathrm{a}_1, \mathrm{b}_2, \mathrm{c}_0]$, which is semi-controllable. According to Definition 2, t is controllable (in **A**lice $\otimes$ **B**ob $\otimes$ **C**arl) via $t' = [\mathrm{a}_2, \mathrm{b}_0, \mathrm{c}_0] \xrightarrow{[!d,?d,-]_\square} [\mathrm{a}_4, \mathrm{b}_2, \mathrm{c}_0]$. Since t is controllable and it is not in agreement (i.e., the label of t is a request), this transition is pruned during the synthesis of the orchestration. We note that t is controllable in t' also in all sub-automaton of the composition computed in the various iterations of the synthesis algorithm, and in the final orchestration depicted in Figure 5.

Using the refined notion of semi-controllability of Definition 4, the orchestration of **A**lice $\otimes$ **B**ob $\otimes$ **C**arl is empty (i.e., there is no orchestration). Consider again transition t. From state $[\mathrm{a}_1, \mathrm{b}_0, \mathrm{c}_0]$, it is not possible to reach any transition labelled by $[!d, ?d, -]_\square$. It follows that t is uncontrollable. Hence, at some iteration i of the orchestration synthesis algorithm in Definition 3, state $[\mathrm{a}_1, \mathrm{b}_0, \mathrm{c}_0]$ becomes forbidden and it is added to the set R_i. At iteration $i+1$, the controllable transition $[\mathrm{a}_0, \mathrm{b}_0, \mathrm{c}_0] \xrightarrow{[!a,-,-]_\square} [\mathrm{a}_1, \mathrm{b}_0, \mathrm{c}_0]$ is pruned because its target state is forbidden. At the next iteration $(i+2)$, the initial state $[\mathrm{a}_0, \mathrm{b}_0, \mathrm{c}_0]$ becomes forbidden, because there are semi-controllable transitions not in agreement exiting the initial state (e.g., $[\mathrm{a}_0, \mathrm{b}_0, \mathrm{c}_0] \xrightarrow{[-,?c,-]_\square} [\mathrm{a}_0, \mathrm{b}_1, \mathrm{c}_0]$) that are uncontrollable in the sub-automaton whose transitions are T_{i+2}. Since the initial state is forbidden, it follows that there is no orchestration for **A**lice $\otimes$ **B**ob $\otimes$ **C**arl.

Indeed, whenever the state $[\mathrm{a}_1, \mathrm{b}_0, \mathrm{c}_0]$ is reached, although **B**ob and **C**arl are still in their initial state, **B**ob can no longer perform the necessary request $?d$ and **C**arl can no longer perform the request $?f$. In fact, neither **B**ob nor **C**arl can decide internally which necessary request to execute from their current state. For example, there is no trace where the request $?c$ of **B**ob and the request $?f$ of **C**arl are matched.

The orchestrations belonging to *Refined* (i.e., orchestrations computed using the refined notion of semi-controllability given in Definition 4) have an intuitive interpretation when compared to the classic notion of uncontrollability. We recall that uncontrollable transitions are called *urgent* necessary transitions in MSCA, while semi-controllable transitions are called *lazy* necessary transitions.

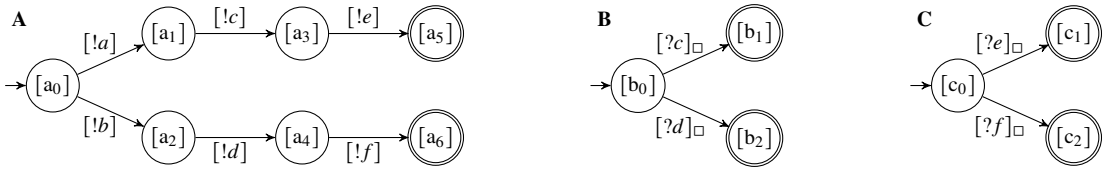

Figure 4: Contracts of **A**lice, **B**ob and **C**arl

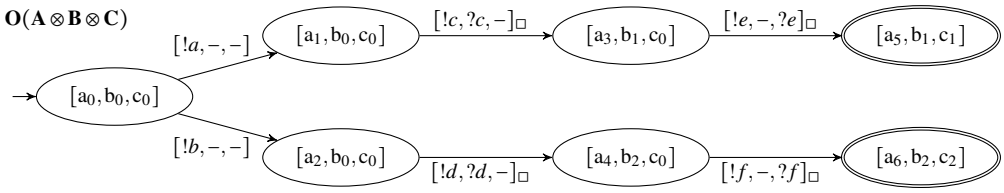

Figure 5: Orchestration $\mathbf{O}(\mathbf{A}\otimes\mathbf{B}\otimes\mathbf{C})$ of $\mathbf{A}\text{lice}\otimes\mathbf{B}\text{ob}\otimes\mathbf{C}\text{arl}$

Intuitively, an urgent transition cannot be delayed, whereas this is the case for a lazy one. In a concurrent composition of agents, the scheduling of concurrent urgent necessary transitions is *uncontrollable*. Instead, concerning concurrent lazy necessary transitions, each agent *internally* decides its next lazy necessary transition to execute, but the orchestrator schedules when this transition will be executed, i.e., the scheduling is *controllable*. In Example 2, there is no orchestration because, for example, from state $[a_1,b_0,c_0]$ there is no possible scheduling that allows the services to match all their necessary requests. Continuing Example 1, the orchestration in Figure 2 is non-empty because the scheduling of the actions in the orchestration is *controlled* by the orchestrator: one of the two necessary requests is scheduled to be matched only when the server has reached its internal state $[2]$. If instead the clients' necessary request $?a$ is urgent, then there exists no orchestration of the composition of two clients and the server. This is because in this case the scheduling is *uncontrollable*: it is not possible to schedule one of the two clients to have its necessary urgent request to be matched only when the server reaches the state $[2]$. In this case, the server should be ready to match the requests whenever they can be executed, without delaying them.

4 Research Challenges

In this section, we describe the currently known limits of the synthesis of orchestrations adopting either Definition 2 or Definition 4, we identify a number of research challenges to overcome these limits, and we propose a research roadmap aimed to tackle these challenges effectively.

First, the notion of semi-controllability introduced in [8, 5] and recalled in Definition 2 allows to synthesise orchestrations that may sometimes limit the capability of each service to perform internal choices. The contract automata formalism abstracts from the way that choices are made. Different implementations are possible in which each service may or may not decide the next step in an orchestration [6].

Consider again Example 2. Both **B**ob and **C**arl are able to perform two alternative necessary requests from their initial state. However, as shown in Figure 5, they are forbidden from internally deciding which necessary request is to be executed at runtime. If, for example, **B**ob selects the request $?d$ and **C**arl selects the request $?e$, then it is not possible for **A**lice to match both requests.

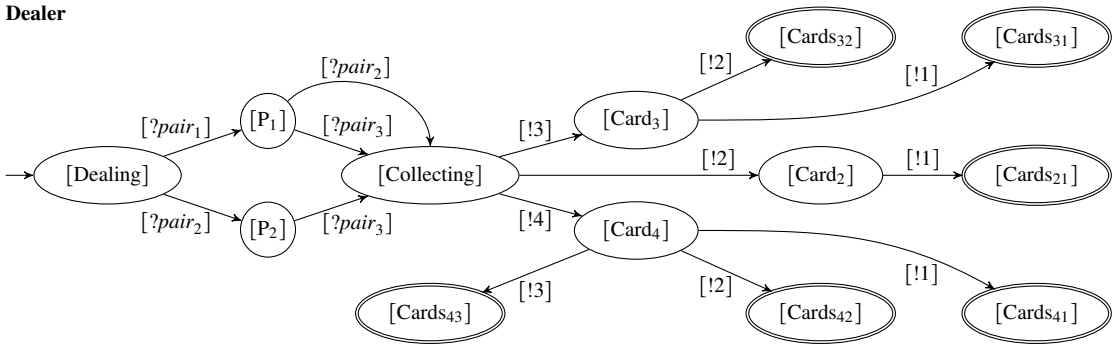

Figure 6: Contract of the **Dealer**

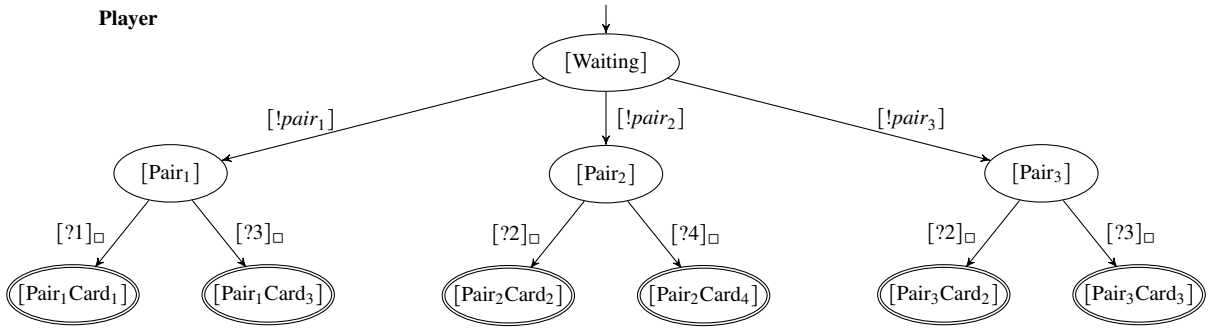

Figure 7: Contract of the **Player**

If we adopt the interpretation given previously (i.e., agents internally choose their necessary transitions and their scheduling is controllable) then we argue that the orchestration computed using Definition 2 is too abstract and should in fact be empty. This is indeed the case if Definition 4 were used instead of Definition 2.

The first research challenge is to identify a concrete application of services that perform necessary requests and whose orchestration belongs to the set $\mathit{Orchestrations} \setminus \mathit{Refined}$.

Solving this challenge could help provide an intuitive interpretation of these types of orchestrations. An application should be identified in which each service statically requires that for each necessary request there must exist an execution where this is eventually matched (cf. Definition 2). However, during execution, the choice of which necessary request is to be matched could be external to the service performing the necessary request. Even if the execution of different branches is determined externally, a service contract may still require all branches to be available in the composition. This could be due to the contract's need to enforce certain hyperproperties, such as non-interference or opacity.

Next, we illustrate the second research challenge. All examples of orchestrations currently available in the literature [7, 6, 5, 9, 8] reside inside the set *Refined* (cf. Figure 3). We showed in Example 2 an orchestration **O** not belonging to the set *Refined* and we argued that **O** is too abstract and should in fact be empty. We now provide another example of an orchestration not belonging to the set *Refined*. However, differently from Example 2, in this case the orchestration should not be empty.

Example 3. This example involves a simple card game with two players and a dealer. At the beginning of each round, the dealer chooses a pair of cards to deal to each player (i.e., each player receives a pair of cards). The dealer can select two out of three different pairs of cards:

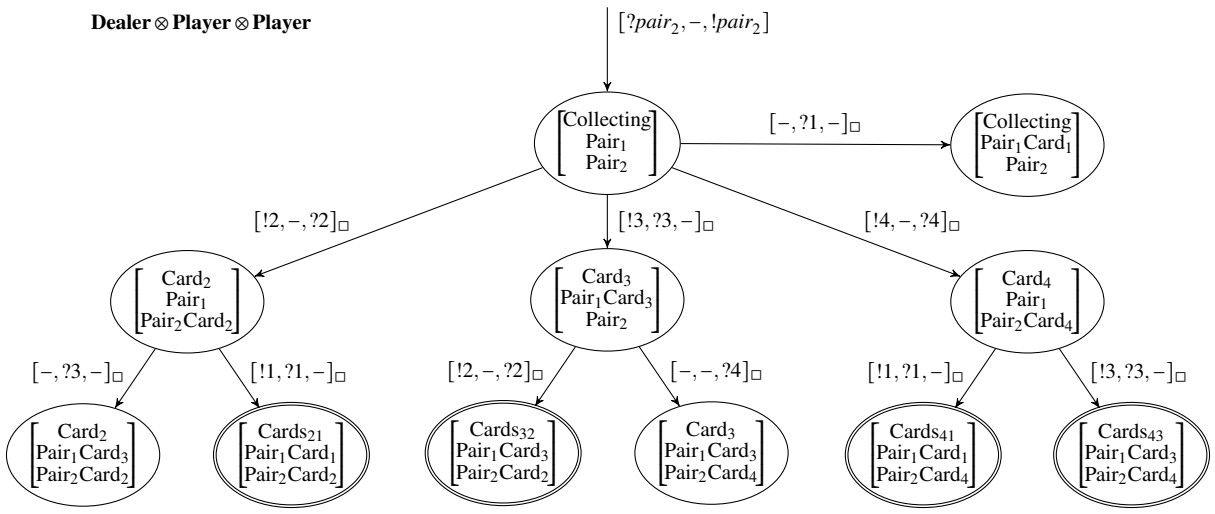

Figure 8: A fragment of the composition of **Dealer** ⊗ **Player** ⊗ **Player**

- Pair 1: card 1 and card 3;
- Pair 2: card 2 and card 4;
- Pair 3: card 2 and card 3.

After the dealer has dealt the pairs of cards, each player selects one of the two cards that was received. Once the players have selected their cards, the dealer collects the selected cards from each player. The goal of the game is for the dealer to avoid picking up two cards in ascending or equal order, which would result in the dealer losing. In other words, if the dealer picks up a card that is higher than the other card that was picked up or if two cards of the same value are picked up, the dealer loses. To ensure that the dealer never loses, the dealer has to choose the correct pairs of cards to deal. There are six possible ways to choose the pairs of cards, but only two of them guarantee a strategy for the dealer to collect the cards selected by the players in descending order. The strategy for the dealer consists of dealing to the players (in no particular order) Pair 1 and Pair 2. Indeed, in the remaining cases there exists the possibility that the players *internally* select the same card. In this case, there is no way of rearranging the transitions to avoid the same cards being picked by the dealer.

We modelled this above-mentioned problem as an orchestration of contracts, using the refined notion of semi-controllability. We only model one round of the game. The CA in Figure 6 models the dealer. Note that each request can be matched by either of the two players. Once the dealer has dealt the pairs of cards, the cards selected by the players are collected. Note that the two cards can only be collected in descending order. The CA in Figure 7 models a player. Once the player has received a card, the player decides internally which card to select. This internal decision is modelled as a choice among lazy necessary transitions.

The synthesis algorithm adopting the refined notion of semi-controllability from Definition 4 takes as input the composition of the dealer CA and two players CA and returns an empty orchestration. To explain why the resulting orchestration is empty, consider Figure 8 depicting a portion of the composition of the dealer with two players.

The state $[\mathrm{Collecting}, \mathrm{Pair}_1, \mathrm{Pair}_2]$ is reached when the first player receives $pair_1$ and the second player receives $pair_2$. A symmetric argument holds for state $[\mathrm{Collecting}, \mathrm{Pair}_2, \mathrm{Pair}_1]$, not depicted here.

The transition

$$[\text{Card}_2, \text{Pair}_1, \text{Pair}_2\text{Card}_2] \xrightarrow{[-,?3,-]_{\square}} [\text{Card}_2, \text{Pair}_1\text{Card}_3, \text{Pair}_2\text{Card}_2]$$

is uncontrollable according to Definition 4. Indeed, from state $[\text{Card}_2, \text{Pair}_1, \text{Pair}_2\text{Card}_2]$ it is not possible to reach state $[\text{Collecting}, \text{Pair}_1, \text{Pair}_2]$. This makes the state $[\text{Card}_2, \text{Pair}_1, \text{Pair}_2\text{Card}_2]$ forbidden. Hence, to avoid reaching a forbidden state, the algorithm prunes the transition

$$[\text{Collecting}, \text{Pair}_1, \text{Pair}_2] \xrightarrow{[!2,-,?2]_{\square}} [\text{Card}_2, \text{Pair}_1, \text{Pair}_2\text{Card}_2]$$

which is in fact controllable according to Definition 4. Indeed, from state $[\text{Collecting}, \text{Pair}_1, \text{Pair}_2]$ it is possible to reach the transition

$$[\text{Card}_3, \text{Pair}_1\text{Card}_3, \text{Pair}_2] \xrightarrow{[!2,-,?2]_{\square}} [\text{Cards}_{32}, \text{Pair}_1\text{Card}_3, \text{Pair}_2\text{Card}_2]$$

via a transition in which the second player is idle. However, during the synthesis algorithm also the state $[\text{Card}_3, \text{Pair}_1\text{Card}_3, \text{Pair}_2]$ becomes forbidden due to its outgoing necessary transition, which is uncontrollable according to Definition 2. This in turn causes the pruning of transition

$$[\text{Collecting}, \text{Pair}_1, \text{Pair}_2] \xrightarrow{[!3,?3,-]_{\square}} [\text{Card}_3, \text{Pair}_1\text{Card}_3, \text{Pair}_2]$$

which is controllable. Once the transition has been pruned, the transition

$$[\text{Collecting}, \text{Pair}_1, \text{Pair}_2] \xrightarrow{[!2,-,?2]_{\square}} [\text{Card}_2, \text{Pair}_1, \text{Pair}_2\text{Card}_2]$$

which was previously controllable becomes uncontrollable. This makes the state $[\text{Collecting}, \text{Pair}_1, \text{Pair}_2]$ forbidden. Note, however, that $[\text{Collecting}, \text{Pair}_1, \text{Pair}_2]$ should *not* be forbidden. Indeed, from that state, for each pair of cards selected by the players, the dealer has a strategy to pick them in the correct order:

- if player 1 selects card 1 and player 2 selects card 2, then execute $[!2,-,?2], [!1,?1,-]$;
- if player 1 selects card 1 and player 2 selects card 4, then execute $[!4,-,?4], [!1,?1,-]$;
- if player 1 selects card 3 and player 2 selects card 2, then execute $[!3,?3,-], [!2,-,?2]$;
- if player 1 selects card 3 and player 2 selects card 4, then execute $[!4,-,?4], [!3,?3,-]$.

This example shows that there are cases for which Definition 4 is too restrictive. In this case, the orchestration can be computed using Definition 2, and it is displayed in Figure 9.

To better understand the underlying assumption of Definition 4, we need to decouple the moment in which a service *selects* which transition it will execute from the moment in which a service *executes* that transition. The underlying assumption of Definition 4 is that these two moments are not decoupled.

For example, the first player whose internal state is Pair_1 could select and execute ?3 also from state $[\text{Card}_2, \text{Pair}_1, \text{Pair}_2\text{Card}_2]$, while the strategy described above assumes that the player selects a card in state $[\text{Collecting}, \text{Pair}_1, \text{Pair}_2]$. In fact, the current implementation of the contract automata runtime environment `CARE` [6] allows the decoupling of these two moments. Once state $[\text{Collecting}, \text{Pair}_1, \text{Pair}_2]$ is reached, the orchestrator interacts with both players and, based on their choices, correctly schedules the transitions of the dealer and the players. This means that the players select their next action in state $[\text{Collecting}, \text{Pair}_1, \text{Pair}_2]$ and afterwards their execution is bounded to the transition they have selected. Summarising, Example 2 has showed that in some cases Definition 2 is too abstract, whereas Example 3 has showed that in some cases Definition 4 is too restrictive.

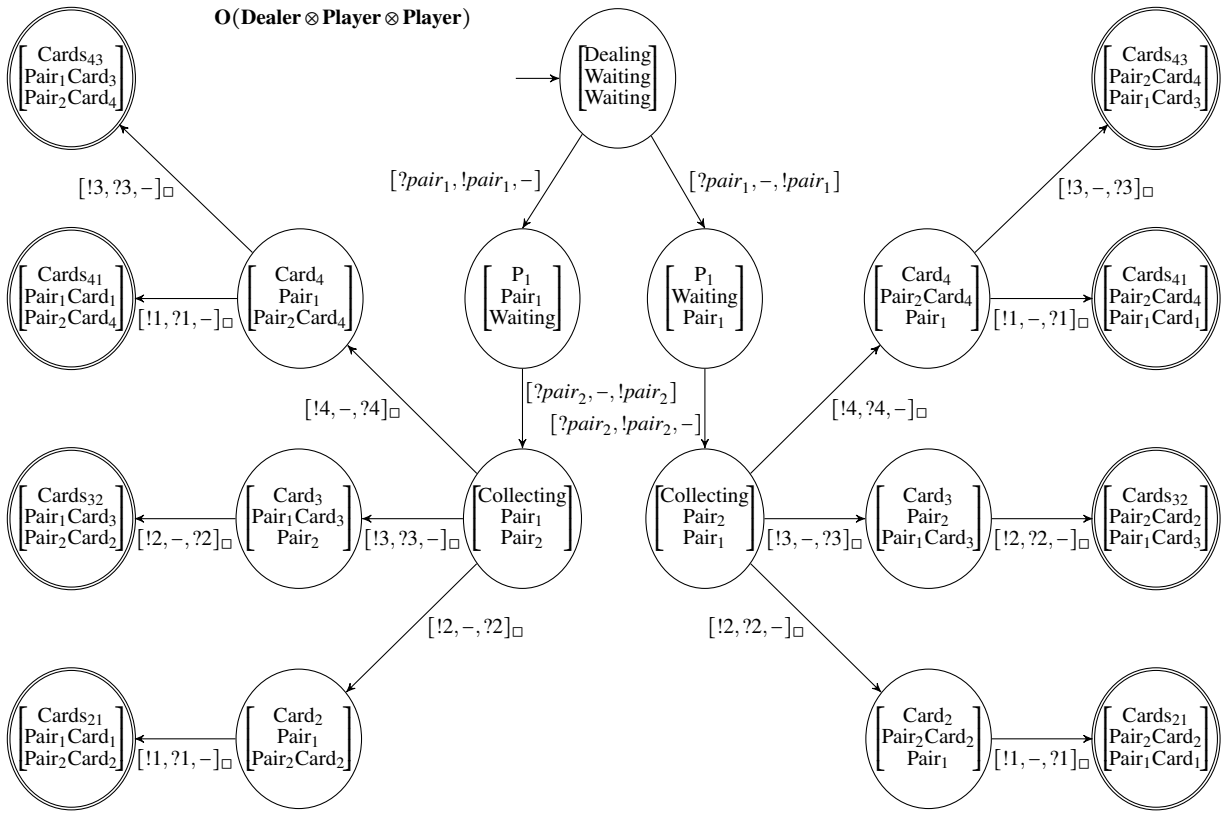

Figure 9: Orchestration $\mathbf{O}(\mathbf{Dealer} \otimes \mathbf{Player} \otimes \mathbf{Player})$

The second research challenge is to identify a notion of semi-controllability capable of discarding orchestrations such as the one in Example 2 and providing non-empty orchestrations in scenarios such as the one described in Example 3.

The resulting, currently unknown set of orchestrations that would be identified by the notion of semi-controllability that solves this challenge is depicted in Figure 3 with dashed lines.

We continue by discussing further research challenges for the orchestration synthesis of contract automata. An important aspect is the ability to scale to large orchestrations when many service contracts are composed. We note that computing Definition 4 is harder than computing Definition 2, due to the additional constraint of reachability which requires a visit of the automaton. Decoupling the moment in which a service selects a choice from the moment in which the selected choice is executed, could further increase the hardness of deciding when a lazy necessary transition is controllable.

Consider again the CA in Figure 1. From their initial state, both **B**ob and **C**arl have two choices. If, instead of two principals, we had ten principals whose behaviour is similar to that of **B**ob and **C**arl, then there would be 2^{10} possible combinations of (internal) choices the services could make.

The third research challenge is to provide scalable solutions for synthesising orchestrations.

Generally speaking, the behaviour of an orchestration that belongs to the unknown dotted set of Figure 3 must be a sub-automaton of an orchestration computed using Definition 2 and a super-automaton of an orchestration computed using Definition 4. Indeed, Definition 2 can be used as an upper bound and Definition 4 as a lower bound to approximate the behaviour of such an orchestration.

Finally, we discuss the last research challenge identified in this paper. We previously formalised the notion of lazy necessary request that is semi-controllable according to either Definition 2 or Definition 4. We noted that Definition 2 may exclude the case in which, in the presence of a choice, a service may internally select its necessary transition. Instead, Definition 4 may exclude the case in which, in the presence of a choice, the moment in which the service internally selects its necessary transition is decoupled from the moment in which the selected necessary transition is executed. In other words, we identified two *requirements* that an orchestration of services should satisfy: *independence* and *decoupling* of choices.

The fourth research challenge is to consolidate a set of requirements that a desirable orchestration of service contracts must satisfy.

The requirements that would solve this challenge should be established incrementally, as discussed in this paper. Formal definitions of necessary service transitions and practical examples are useful to identify the ideal set of requirements that an orchestration of services should satisfy. Of course, these requirements are entangled with the underlying execution support of an orchestration of services, which was recently proposed in [6].

4.1 Research Roadmap

We have presented a series of research challenges associated with the orchestration of contract automata. We now propose a potential research roadmap aimed at tackling these challenges effectively. However, it is necessary to further examine the concepts described below to determine their validity.

Specifying Choices We propose to concretise the selection of the next transition to execute at contract automata level, distinguishing between internal and external selections. Presently, this distinction is abstracted away within contracts and handled by the underlying execution support. Our rationale is that abstracting from the selection process may lead to scalability challenges. Specifically, if a transition is selected internally, it must always be available, whereas an externally selected transition can be removed from the orchestration. In essence, internal selection imposes stricter requirements than external selection. Consequently, treating all selections as internal to ensure independence of choice leads to larger state spaces. For instance, the issue highlighted in Example 2 arises due to the presence of externally selected transitions. By allowing contracts to specify which transitions are internally or externally selected, we can potentially reduce the state space, as compared to considering all choices as internal.

Pursuing the above has important implications. Firstly, it necessitates updating accordingly the underlying execution support, `CARE`, to align it with the contract automata specifications. This entails reducing the implementation freedom for each choice to adhere to the contract's explicit selection of the next transition. By explicating choices within contracts, we establish the interpretation of necessary requests discussed in this paper. In this interpretation, a service *internally* decides to perform a necessary request, but the scheduling of the execution of the request is controlled by the orchestrator. In other words, optional actions are externally selected, whereas necessary actions are internally selected. Consequently, by explicitly stating choices in contracts, we can address the first research challenge. Indeed, all necessary requests would be internally selected. Scenarios like the one outlined in the contect of the first research challenge (i.e., external necessary requests) would be practically ruled out.

Another implication involves associating optional actions with offers and necessary actions with requests, which helps eliminate undefined choices. For instance, a branch where all actions (both offers and requests) are optional would require runtime support to determine which service is responsible for choosing the next step.

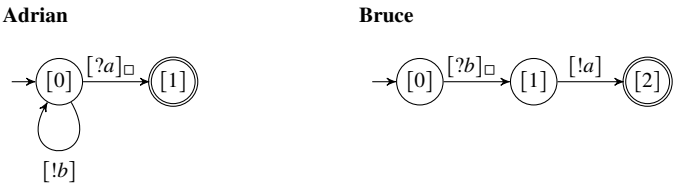

Figure 10: Two contracts whose orchestration requires further investigation

A third and final implication relates to the fourth research challenge, which entails consolidating a set of requirements for effective orchestrations. Notably, if choices are explicitly specified in contracts, the requirement of independence of choice can be removed.

Implementing the Decoupling of Choices The second research challenge, as mentioned previously, revolves around the absence of the decoupling of choice requirement in both Definitions 2 and 4. This requirement suggests a potential implementation of semi-controllable transitions and may help identify the currently unknown set of orchestrations in Figure 3. Currently, a semi-controllable transition is defined as a transition that can be either controllable or uncontrollable based on a global condition of the automaton. However, decoupling the moment when a service internally selects a transition from the moment when the transition is executed might require splitting a semi-controllable transition into two distinct transitions.

Reasoning in this way suggests that a semi-controllable transition could potentially be represented as two consecutive transitions. The first transition would be uncontrollable, capturing the internal selection, while the subsequent transition would be controllable and responsible for executing the action. For example, consider a semi-controllable transition $[\mathtt{q}] \xrightarrow{[?a]_{\Box/u}} [\mathtt{q}']$, which would be split into two transitions: $t_1 = [\mathtt{q}] \xrightarrow{[\tau]_{\Box u}} [\mathtt{i}]$ and $t_2 = [\mathtt{i}] \xrightarrow{[?a]} [\mathtt{q}']$. Here, t_1 represents an uncontrollable silent transition to an intermediate, non-final state, while t_2 is controllable and executes the action. This approach suggests that the orchestrator cannot control the internal selection made with t_1, but it can control and schedule the execution of the action indicated by t_2. Moreover, an important consequence of the fact that the intermediate state is non-final, is that t_2 must eventually be executed.

Further exploration is required to determine whether this interpretation of semi-controllability solves the third research challenge. In particular, there are still corner cases that require further investigation. For instance, consider the contracts in Figure 10. Although an orchestration could be obtained by matching the necessary request `?b` of Bruce first and only afterwards the necessary request `?a` of Adrian, this orchestration is not supported by the notion of semi-controllability outlined above. In this orchestration, Adrian internally selects the request `?a` and the orchestrator schedules the request of Adrian to be matched later after Adrian matches the request of Bruce.

Furthermore, we envision the establishment of a clear separation between optional and necessary transitions on the one hand and controllable and uncontrollable transitions on the other. Modellers should only define contracts with requests and offers, in which all requests are considered necessary, and all offers are considered optional. Additionally, all necessary requests should be categorised as lazy/semi-controllable, thus effectively excluding urgent necessary requests from contracts. This implies that contract automata with optional and necessary transitions should be transformed into automata with solely controllable and uncontrollable transitions, which are known as plant automata in supervisory control theory. It is worth noting that all uncontrollable transitions will serve as silent moves to represent the internal selection of a necessary transition.

Experimental Validation of Performance The third research challenge highlights the issue of scalability and proposes the adoption of Definition 2 as an upper bound for the set of orchestrations. However, it remains unclear whether the synthesis process using Definition 2 is faster compared to synthesising using the mapped plant automaton as suggested earlier. Definition 2 necessitates a visit of the automaton at each iteration of the synthesis process to determine whether a semi-controllable transition is controllable or uncontrollable. This requirement is not present in a plant automaton consisting solely of controllable and uncontrollable transitions. On the other hand, the suggested mapping approach increases the state space of the automata by introducing an additional state for each necessary transition. As a result, it is essential to conduct further experimental research to assess the effectiveness of utilising Definition 2 as an upper bound for the set of orchestrations. This research should involve measuring the performance and efficiency of the synthesis process when employing Definition 2 and comparing it with the approach based on the mapped plant automaton.

5 Conclusion

We have presented a number of research challenges related to the orchestration synthesis of contract automata. Initially, we proposed a novel refined definition of semi-controllability and compared it to the current definition through illustrative examples. We identified various sets of orchestrations, as showed in Figure 3. Additionally, we informally discussed two prerequisites that the orchestration of contracts should satisfy: independence and decoupling of choices. Furthermore, we evaluated the current formal definitions of semi-controllability based on these requirements, which generated a series of research questions regarding the orchestration synthesis of contract automata, to be addressed in future work, possibly by following the proposed research roadmap.

Acknowledgements We would like to thank the reviewers for their useful comments. This work has been partially supported by the Italian MIUR PRIN 2017FTXR7S project IT MaTTerS (Methods and Tools for Trustworthy Smart Systems) and the Italian MUR PRIN 2020TL3X8X project T-LADIES (Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems).

References

[1] Eugene Asarin, Oded Maler, Amir Pnueli & Joseph Sifakis (1998): *Controller Synthesis for Timed Automata*. *IFAC Proc. Vol.* 31(18), pp. 447–452, doi:10.1016/S1474-6670(17)42032-5.

[2] Lorenzo Bacchiani, Mario Bravetti, Marco Giunti, João Mota & António Ravara (2022): *A Java typestate checker supporting inheritance*. *Sci. Comput. Program.* 221, doi:10.1016/j.scico.2022.102844.

[3] Franco Barbanera, Ivan Lanese & Emilio Tuosto (2022): *On Composing Communicating Systems*. In Clément Aubert, Cinzia Di Giusto, Larisa Safina & Alceste Scalas, editors: *Proceedings of the 15th Interaction and Concurrency Experience (ICE'22)*, *EPTCS* 365, pp. 53–68, doi:10.4204/EPTCS.365.4.

[4] Davide Basile & Maurice H. ter Beek (2021): *A Clean and Efficient Implementation of Choreography Synthesis for Behavioural Contracts*. In Ferruccio Damiani & Ornela Dardha, editors: *Proceedings of the 23rd IFIP WG 6.1 International Conference on Coordination Models and Languages (COORDINATION'21)*, *LNCS* 12717, Springer, pp. 225–238, doi:10.1007/978-3-030-78142-2_14.

[5] Davide Basile & Maurice H. ter Beek (2022): *Contract Automata Library*. *Sci. Comput. Program.* 221, doi:10.1016/j.scico.2022.102841. Available at `https://github.com/contractautomataproject/ContractAutomataLib`.

[6] Davide Basile & Maurice H. ter Beek (2023): *A Runtime Environment for Contract Automata*. In Marsha Chechik, Joost-Pieter Katoen & Martin Leucker, editors: *Proceedings of the 25th International Symposium on Formal Methods (FM'23)*, *LNCS* 14000, Springer, pp. 550–567, doi:10.1007/978-3-031-27481-7_31.

[7] Davide Basile, Maurice H. ter Beek, Laura Bussi & Vincenzo Ciancia (2023): *A Toolchain for Strategy Synthesis with Spatial Properties*. *Int. J. Softw. Tools Technol. Transf.*

[8] Davide Basile, Maurice H. ter Beek, Pierpaolo Degano, Axel Legay, Gian Luigi Ferrari, Stefania Gnesi & Felicita Di Giandomenico (2020): *Controller synthesis of service contracts with variability*. *Sci. Comput. Program.* 187, doi:10.1016/j.scico.2019.102344.

[9] Davide Basile, Maurice H. ter Beek & Rosario Pugliese (2020): *Synthesis of Orchestrations and Choreographies: Bridging the Gap between Supervisory Control and Coordination of Services*. *Log. Methods Comput. Sci.* 16(2), pp. 9:1–9:29, doi:10.23638/LMCS-16(2:9)2020.

[10] Davide Basile, Pierpaolo Degano & Gian Luigi Ferrari (2016): *Automata for Specifying and Orchestrating Service Contracts*. *Log. Methods Comput. Sci.* 12(4), pp. 6:1–6:51, doi:10.2168/LMCS-12(4:6)2016.

[11] Davide Basile, Pierpaolo Degano, Gian Luigi Ferrari & Emilio Tuosto (2014): *From Orchestration to Choreography through Contract Automata*. In Ivan Lanese, Alberto Lluch Lafuente, Ana Sokolova & Hugo Torres Vieira, editors: *Proceedings of the 7th Interaction and Concurrency Experience (ICE'14)*, *EPTCS* 166, pp. 67–85, doi:10.4204/EPTCS.166.8.

[12] Davide Basile, Pierpaolo Degano, Gian Luigi Ferrari & Emilio Tuosto (2016): *Relating two automata-based models of orchestration and choreography*. *J. Log. Algebr. Methods Program.* 85(3), pp. 425–446, doi:10.1016/j.jlamp.2015.09.011.

[13] Davide Basile, Felicita Di Giandomenico, Stefania Gnesi, Pierpaolo Degano & Gian Luigi Ferrari (2017): *Specifying Variability in Service Contracts*. In: *Proceedings of the 11th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'17)*, ACM, pp. 20–27, doi:10.1145/3023956.3023965.

[14] Maurice H. ter Beek, Josep Carmona, Rolf Hennicker & Jetty Kleijn (2017): *Communication Requirements for Team Automata*. In Jean-Marie Jacquet & Mieke Massink, editors: *Proceedings of the 19th International Conference on Coordination Models and Languages (COORDINATION'17)*, *LNCS* 10319, Springer, pp. 256–277, doi:10.1007/978-3-319-59746-1_14.

[15] Maurice H. ter Beek, G. Cledou, R. Hennicker & J. Proença (2023): *Can we Communicate? Using Dynamic Logic to Verify Team Automata*. In M. Chechik, J.-P. Katoen & M. Leucker, editors: *Proceedings of the 25th International Symposium on Formal Methods (FM'23)*, *LNCS* 14000, Springer, pp. 122–141, doi:10.1007/978-3-031-27481-7_9.

[16] Athman Bouguettaya, Munindar P. Singh, Michael N. Huhns, Quan Z. Sheng, Hai Dong, Qi Yu, Azadeh Ghari Neiat, Sajib Mistry, Boualem Benatallah, Brahim Medjahed, Mourad Ouzzani, Fabio Casati, Xumin Liu, Hongbing Wang, Dimitrios Georgakopoulos, Liang Chen, Surya Nepal, Zaki Malik, Abdelkarim Erradi, Yan Wang, M. Brian Blake, Schahram Dustdar, Frank Leymann & Michael P. Papazoglou (2017): *A Service Computing Manifesto: The Next 10 Years*. *Commun. ACM* 60(4), pp. 64–72, doi:10.1145/2983528.

[17] Alberto Camacho, Meghyn Bienvenu & Sheila A. McIlraith (2019): *Towards a Unified View of AI Planning and Reactive Synthesis*. In: *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS'18)*, AAAI, pp. 58–67, doi:10.1609/icaps.v29i1.3460.

[18] Christos G. Cassandras & Stéphane Lafortune (2006): *Introduction to Discrete Event Systems*. Springer, doi:10.1007/978-0-387-68612-7.

[19] `CATApp`. Available at `https://github.com/contractautomataproject/ContractAutomataApp`.

[20] Rüdiger Ehlers, Stéphane Lafortune, Stavros Tripakis & Moshe Y. Vardi (2017): *Supervisory control and reactive synthesis: a comparative introduction*. *Discret. Event Dyn. Syst.* 27(2), pp. 209–260, doi:10.1007/s10626-015-0223-0.

[21] Paolo Felli, Nitin Yadav & Sebastian Sardina (2017): *Supervisory Control for Behavior Composition*. *IEEE Trans. Autom. Control* 62(2), pp. 986–991, doi:10.1109/TAC.2016.2570748.

[22] Dimitrios Kouzapas, Ornela Dardha, Roly Perera & Simon J. Gay (2018): *Typechecking protocols with Mungo and StMungo: A session type toolchain for Java*. *Sci. Comput. Program.* 155, pp. 52–75, doi:10.1016/j.scico.2017.10.006.

[23] Michael Luttenberger, Philipp J. Meyer & Salomon Sickert (2020): *Practical synthesis of reactive systems from LTL specifications via parity games*. *Acta Inform.* 57(1-2), pp. 3–36, doi:10.1007/s00236-019-00349-3.

[24] Peter J. Ramadge & Walter M. Wonham (1987): *Supervisory Control of a Class of Discrete Event Processes*. *SIAM J. Control Optim.* 25(1), pp. 206–230, doi:10.1137/0325013.

[25] Robert E. Strom & Shaula Yemini (1986): *Typestate: A Programming Language Concept for Enhancing Software Reliability*. *IEEE Trans. Softw. Eng.* 12(1), pp. 157–171, doi:10.1109/TSE.1986.6312929.

[26] André Trindade, João Mota & António Ravara (2020): *Typestates to Automata and back: a tool*. In Julien Lange, Anastasia Mavridou, Larisa Safina & Alceste Scalas, editors: *Proceedings of the 13th Interaction and Concurrency Experience (ICE'20)*, *EPTCS* 324, pp. 25–42, doi:10.4204/EPTCS.324.4.

[27] Nobuko Yoshida, Fangyi Zhou & Francisco Ferreira (2021): *Communicating Finite State Machines and an Extensible Toolchain for Multiparty Session Types*. In Evripidis Bampis & Aris Pagourtzis, editors: *Proceedings of the 23rd International Symposium on Fundamentals of Computation Theory (FCT'21)*, *LNCS* 12867, Springer, pp. 18–35, doi:10.1007/978-3-030-86593-1_2.

Partially Typed Multiparty Sessions

Franco Barbanera*
Dipartimento di Matematica e Informatica, Università di Catania, Catania, Italy
barba@dmi.unict.it

Mariangiola Dezani-Ciancaglini
Dipartimento di Informatica, Università di Torino, Torino, Italy
dezani@di.unito.it

A multiparty session formalises a set of concurrent communicating participants. We propose a type system for multiparty sessions where some communications between participants can be ignored. This allows us to type some sessions with global types representing interesting protocols, which have no type in the standard type systems. Our type system enjoys Subject Reduction, Session Fidelity and "partial" Lock-freedom. The last property ensures the absence of *locks* for participants with non ignored communications. A sound and complete type inference algorithm is also discussed.

1 Introduction

The key issue in multiparty distributed systems is the composition of independent entities such that a sensible behaviour of the whole emerges from those of the components, while avoiding type errors of exchanged messages and ensuring good communication properties like Lock-freedom. MultiParty Session Types (MPST), introduced in [16, 17], are a class of choreographic formalisms for the description and analysis of such systems. Choreographic formalism are characterised by the coexistence of two distinct but related views of distributed systems: the *global* and the *local* views. The former describes the behaviour of a system as a whole, whereas the local views specify the behaviour of the single components in "isolation". Systems described by means of MPST formalisms are usually ensured (i) their overall behaviour to adhere to a given communication protocol (represented as a global type) and (ii) to enjoy particular communication properties like Lock-freedom (the specific property we focus on in the present paper).

In [3] a MPST formalism was developed for systems using synchronous communications, where global types can be assigned to multiparty sessions (parallel composition of named processes) via a type system. Typability of a multiparty session $\mathbb{M}$ by a global type G ensures that $\mathbb{M}$ behaves as described by G and is lock-free.

The property of Lock-freedom ensures that no lock is ever reached in the evolution of a system. A lock is a system's reachable configuration where a participant, which is able to perform an action, is forever prevented to do so in any possible continuation. In particular, such a configuration is called a p-lock in case the stuck participant be p. Lock-freedom – which entails Deadlock-freedom – could be however too strong to be proved in some settings, and actually useless sometimes. As a matter of fact, for particular systems, the presence of p-locks for some participants would not be problematic and would not break their specifications. Let us assume, for instance, to have a social medium where participants can

*Partially supported by Project "National Center for "HPC, Big Data e Quantum Computing", Programma M4C2 – dalla ricerca all'impresa – Investimento 1.3: Creazione di "Partenariati estesi alle università, ai centri di ricerca, alle aziende per il finanziamento di progetti di ricerca di base" – Next Generation EU; and by the Piano Triennale Ricerca Pia.Ce.Ri UniCT.

To appear in EPTCS.

ask for upgrades of their communication level (i.e. the capability describing which participants they can communicate with and which sort of messages can be sent). The upgrades are granted by some particular participant u according to the particular policy of the social medium. In case u be implemented so to reply to an unbounded number of requests, it is immediate to realise that, in case all the participants get to the highest communication level, we would be in presence of a u-lock since no more level upgrade will be requested. This would not be a problem, since what we are interested in is the possibility for all the participants to progress until no communication with u is possible. From that moment on the participants other than u must be ensured to progress, but not u. This sort of circumstance is typical in clients/servers scenarios. Given a set of participants $\mathscr{P}$, we dub a system to be *$\mathscr{P}$-excluded lock-free* whenever it is p-lock free for each participant p not belonging to $\mathscr{P}$.

In this paper we present a MPST type system in the style of [3] where it is possible to derive judgments of the new shape

$$\mathsf{G} \vdash_{\mathscr{P}} \mathbb{M}$$

We say that our typing is *partial* since some communications between participants in $\mathscr{P}$ do not appear in the global type. Our type system ensures that (a) the communications of the participants in $\mathbb{M}$ not belonging to $\mathscr{P}$ comply with the interaction scenario represented by G and (b) $\mathbb{M}$ is $\mathscr{P}$-excluded lock-free.

Contributions and structure of the paper. In Section 2 we recall the calculus of multiparty sessions from [3], together with the global types. Also, we introduce the novel notion of $\mathscr{P}$-excluded Lock-freedom, that we clarify by means of an example. Section 3 is devoted to the presentation of our "partial" type system, assigning global types to multiparty sessions where some communications can be ignored. Besides, we prove the relevant properties of partially typable sessions: Subject Reduction, Session Fidelity and $\mathscr{P}$-excluded Lock-freedom. In Section 4 we discuss a sound and complete type inference algorithm for our partial type system. A section summing up our results, discussing related works and possible directions for future work concludes the paper.

2 Multiparty Sessions and Global Types

In this section we recall the calculus of multiparty sessions and the global types defined in [3]. This calculus is simpler than the original MPST calculus [16] and many of the subsequent ones. Lack of explicit channels – even if preventing the representation of session interleaving and delegation – enables us to focus on our main concerns and allows for a clear explanation of the type system we will introduce in the next section.

We use the following base sets and notation: *messages*, ranged over by $\lambda, \lambda', \ldots$; *session participants*, ranged over by $\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}, \mathsf{u}, \ldots$; *processes*, ranged over by $P, Q, R, S, U, \ldots$; *multiparty sessions*, ranged over by $\mathbb{M}, \mathbb{M}', \ldots$; *integers*, ranged over by $i, j, l, h, k, \ldots$; *integer sets*, ranged over by $I, J, L, H, K, \ldots$.

Definition 2.1 (Processes) Processes *are defined by:*

$$P ::=^{coind} \mathbf{0} \mid \mathsf{p}!\{\lambda_i.P_i\}_{i\in I} \mid \mathsf{p}?\{\lambda_i.P_i\}_{i\in I}$$

where $I \neq \emptyset$ and $\lambda_j \neq \lambda_h$ for $j, h \in I$ and $j \neq h$.

The symbol $::=^{coind}$ in Definition 2.1 and in later definitions indicates that the productions are interpreted *coinductively*. That is, processes are possibly infinite terms. However, we assume such processes to be *regular*, i.e., with finitely many distinct sub-processes. This is done also in [7] and it allows us to adopt

in proofs the coinduction style advocated in [21] which, without any loss of formal rigour, promotes readability and conciseness.

Processes implement the communication behaviour of participants. The output process $\mathsf{p}!\{\lambda_i.P_i\}_{i\in I}$ non-deterministically chooses one message λ_i for some $i \in I$, and sends it to the participant p, thereafter continuing as P_i. Symmetrically, the input process $\mathsf{p}?\{\lambda_i.P_i\}_{i\in I}$ waits for one of the messages λ_i from the participant p, then continues as P_i after receiving it. When there is only one output we write $\mathsf{p}!\lambda.P$ and similarly for one input. We use $\mathbf{0}$ to denote the terminated process. We shall omit writing trailing $\mathbf{0}$s in processes. We denote by $\mathsf{p}\,\dagger\,\{\lambda_i.P_i\}_{i\in I}$ either $\mathsf{p}!\{\lambda_i.P_i\}_{i\in I}$ or $\mathsf{p}?\{\lambda_i.P_i\}_{i\in I}$.

In a full-fledged calculus, messages would carry values, that we avoid for the sake of simplicity; hence no selection operation over values is included in the syntax.

Definition 2.2 (Multiparty sessions) Multiparty sessions *are expressions of the shape:*

$$\mathsf{p}_1[P_1] \parallel \cdots \parallel \mathsf{p}_n[P_n]$$

where $\mathsf{p}_j \neq \mathsf{p}_h$ *for* $1 \leq j,h \leq n$ *and* $j \neq h$*. We use* $\mathbb{M}$ *to range over multiparty sessions.*

Multiparty sessions (sessions, for short) are parallel compositions of located processes of the form $\mathsf{p}[P]$, each enclosed within a different participant p. We assume the standard structural congruence $\equiv$ on multiparty sessions, stating that parallel composition is associative and commutative and has neutral elements $\mathsf{p}[\mathbf{0}]$ for any p. If $P \neq \mathbf{0}$ we write $\mathsf{p}[P] \in \mathbb{M}$ as short for $\mathbb{M} \equiv \mathsf{p}[P] \parallel \mathbb{M}'$ for some $\mathbb{M}'$. This abbreviation is justified by the associativity and commutativity of parallel composition.

The *set of active participants* (participants for short) of a session $\mathbb{M}$, notation $\mathsf{prt}(\mathbb{M})$, is as expected:

$$\mathsf{prt}(\mathbb{M}) = \{\mathsf{p} \mid \mathsf{p}[P] \in \mathbb{M}\}$$

It is easy to verify that the sets of participants of structurally congruent sessions coincide.

To define the *synchronous operational semantics* of sessions we use an LTS, whose transitions are decorated by labels denoting message exchanges.

Definition 2.3 (LTS for Multiparty Sessions) *The* labelled transition system (LTS) for multiparty sessions *is the closure under structural congruence of the reduction specified by the unique rule:*

$$[\textsc{Comm-T}]\ \frac{h \in I \subseteq J}{\mathsf{p}[\mathsf{q}!\{\lambda_i.P_i\}_{i\in I}] \parallel \mathsf{q}[\mathsf{p}?\{\lambda_j.Q_j\}_{j\in J}] \parallel \mathbb{M} \xrightarrow{\mathsf{p}\lambda_h\mathsf{q}} \mathsf{p}[P_h] \parallel \mathsf{q}[Q_h] \parallel \mathbb{M}}$$

Rule [Comm-T] makes communications possible, by describing when a participant p can send a message λ_h to participant q, and what is the effect of such message exchange. This rule is non-deterministic in the choice of messages. The condition $I \subseteq J$ ensures that the sender can freely choose the message, since the receiver must offer all sender messages and possibly more. This allows us to distinguish in the operational semantics between internal and external choices. Note that this condition will be ensured by the typing Rule [Comm] (see Definition 3.1).

Let Λ range over *labels*, namely triples of the form $\mathsf{p}\lambda\mathsf{q}$. We define *traces* as (possibly infinite) sequences of labels by:

$$\sigma ::=^{coind} \varepsilon \mid \Lambda\cdot\sigma$$

where ε is the empty sequence. We use $|\sigma|$ to denote the length of the trace σ, where $|\sigma| = \infty$ when σ is an infinite trace. We define the participants of labels and traces:

$$\mathsf{prt}(\mathsf{p}\lambda\mathsf{q}) = \{\mathsf{p},\mathsf{q}\} \qquad \mathsf{prt}(\varepsilon) = \emptyset \qquad \mathsf{prt}(\Lambda\cdot\sigma) = \mathsf{prt}(\Lambda) \cup \mathsf{prt}(\sigma)$$

When $\sigma = \Lambda_1 \cdot \ldots \cdot \Lambda_n$ $(n \geq 0)$ we write $\mathbb{M} \xrightarrow{\sigma} \mathbb{M}'$ as short for $\mathbb{M} \xrightarrow{\Lambda_1} \mathbb{M}_1 \cdots \xrightarrow{\Lambda_n} \mathbb{M}_n = \mathbb{M}'$. As usual we write $\mathbb{M} \rightarrow$ (resp. $\mathbb{M} \not\rightarrow$) when there exist (resp. no) Λ and $\mathbb{M}'$ such that $\mathbb{M} \xrightarrow{\Lambda} \mathbb{M}'$.

It is easy to verify that, in a transition, only the two participants of its label are involved, as formalised below.

Fact 2.4 *If* $\{\mathsf{p},\mathsf{q}\} \cap \{\mathsf{r},\mathsf{s}\} = \emptyset$ *and* $\mathsf{r}[R] \parallel \mathsf{s}[S] \parallel \mathbb{M} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{r}[R] \parallel \mathsf{s}[S] \parallel \mathbb{M}'$*, then*

$$\mathsf{r}[R'] \parallel \mathsf{s}[S'] \parallel \mathbb{M} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{r}[R'] \parallel \mathsf{s}[S'] \parallel \mathbb{M}'$$

for arbitrary R', S'.

We define now the property of $\mathscr{P}$-excluded Lock-freedom, a "partial" version of the standard Lock-freedom [19, 26]. The latter consists in the possible eventual completion of pending communications of any participant (this can be alternatively stated by saying that any participant is lock-free). We are interested instead in the progress of some specific participants only, namely those we decide not to "ignore". In the following, $\mathscr{P}$ will range over sets of ignored participants.

Definition 2.5 ($\mathscr{P}$-excluded Lock-freedom) *A multiparty session* $\mathbb{M}$ *is a* $\mathscr{P}$-excluded lock-free *session if* $\mathbb{M} \xrightarrow{\sigma} \mathbb{M}'$ *and* $\mathsf{p} \in \mathsf{prt}(\mathbb{M}') \setminus \mathscr{P}$ *imply* $\mathbb{M}' \xrightarrow{\sigma'\cdot\Lambda} \mathbb{M}''$ *for some* σ' *and* Λ *such that* $\mathsf{p} \in \mathsf{prt}(\Lambda)$.

It is natural to extend also the usual notion of Deadlock-freedom to our setting.

Definition 2.6 ($\mathscr{P}$-excluded Deadlock-freedom) *A multiparty session* $\mathbb{M}$ *is a* $\mathscr{P}$-excluded deadlock-free *session if* $\mathbb{M} \xrightarrow{\sigma} \mathbb{M}' \not\rightarrow$ *implies* $\mathsf{prt}(\mathbb{M}') \subseteq \mathscr{P}$.

It is immediate to check that, as for standard Lock- and Deadlock-freedom, the following hold.

Fact 2.7 *$\mathscr{P}$-excluded Lock-freedom implies $\mathscr{P}$-excluded Deadlock-freedom.*

The vice versa does not hold. For example if $P = \mathsf{q}!\lambda.P$, $Q = \mathsf{p}?\lambda.P$ and $R \neq \mathbf{0}$, then $\mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathsf{r}[R]$ is $\mathscr{P}$-excluded deadlock-free for any $\mathscr{P}$, but $\mathscr{P}$-excluded lock-free only when $\mathsf{r} \in \mathscr{P}$.

The following example illustrates the notion of $\mathscr{P}$-excluded Lock-freedom.

Example 2.8 (Social media) Let us consider a system describing a simplified social media situation. Participant q is allowed to greet participant p by sending a message HELLO. Participant p would like to reply to q, but in order to do that she needs to be granted a higher communication level. The task of granting permissions is performed by participant u which, upon p's request (REQ), decides – according to some parameters – whether the permission is granted (GRTD) or denied (DND). Her decision is communicated to both p and q. We assume that (for reusability motivation) u is implemented in order to process an unbounded number of requests. For what concerns p, however, once she is granted the higher communication level, she can return the greeting to q, so ending their interaction. The above system corresponds to the following session

$$\mathbb{M} \equiv \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathsf{u}[U]$$

where $P = \mathsf{q}?\textsc{hello}.\mathsf{u}!\textsc{req}.\mathsf{u}?\{\textsc{dnd}.P, \textsc{grtd}.\mathsf{q}!\textsc{hello}\}$, $Q = \mathsf{p}!\textsc{hello}.\mathsf{u}?\{\textsc{dnd}.Q, \textsc{grtd}.\mathsf{p}?\textsc{hello}\}$ and $U = \mathsf{p}?\textsc{req}.\mathsf{p}!\{\textsc{dnd}.\mathsf{q}!\textsc{dnd}.U, \textsc{grtd}.\mathsf{q}!\textsc{grtd}.U\}$. The session is $\{\mathsf{u}\}$-excluded lock-free, since, once p has been granted the higher communication level, we get

$$\mathsf{p}[\mathbf{0}] \parallel \mathsf{q}[\mathbf{0}] \parallel \mathsf{u}[U]$$

where u is willing to interact but she will never be able. This, however, should not be deemed a problem, since we are actually interested in that the interactions between p and q do proceed smoothly. ◇

The behaviour of multiparty sessions can be disciplined by means of types. Global types describe the conversation scenarios of multiparty sessions, possibly in a partial way.

Definition 2.9 (Global types) Global types *are defined by:*

$$\mathsf{G} ::=^{coind} \mathtt{End} \mid \mathsf{p} \to \mathsf{q} : \{\lambda_i.\mathsf{G}_i\}_{i\in I}$$

where $I \neq \emptyset$ *and* $\lambda_j \neq \lambda_h$ *for* $j,h \in I$ *and* $j \neq h$.

As for processes, we allow only *regular* global types. The type $\mathsf{p} \to \mathsf{q} : \{\lambda_i.\mathsf{G}_i\}_{i\in I}$ formalises a protocol where participant p must send to q a message λ_j for some $j \in I$ (and q must receive it) and then, depending on which λ_j was chosen by p, the protocol continues as G_j. We write $\mathsf{p} \to \mathsf{q} : \lambda.\mathsf{G}$ when there is only one message. We use $\mathtt{End}$ to denote the terminated protocol. We shall omit writing trailing $\mathtt{End}$s in global types.

We define the *set of paths of a global type*, notation $paths(\mathsf{G})$, as the greatest set of traces such that:

$$paths(\mathtt{End}) = \{\varepsilon\} \qquad paths(\mathsf{p} \to \mathsf{q} : \{\lambda_i.\mathsf{G}_i\}_{i\in I}) = \bigcup_{i\in I}\{\mathsf{p}\lambda_i\mathsf{q}\cdot\sigma \mid \sigma \in paths(\mathsf{G}_i)\}$$

The *set of participants of a global type* is the set of participants of its paths:

$$\mathsf{prt}(\mathsf{G}) = \bigcup_{\sigma\in paths(\mathsf{G})} \mathsf{prt}(\sigma)$$

The regularity of global types ensures that such sets of participants are always finite.

Boundedness is a property of global types that will enable us to get $\mathscr{P}$-excluded Lock-freedom from typability. This consists in requiring any participant of a global type to occur either in all the paths or in no path of any of its subterms which are global types. Notably this condition is a form of fairness, even if it strongly differs from the notions of fairness discussed in [15], where fairness assumptions rule out computational paths. Technically, we shall use the notions of *depth* and of *boundedness* as defined below. We denote by $\sigma[n]$ with $n \in \mathbb{N}$ the n-th label in the path σ, where $1 \leq n \leq |\sigma|$.

Definition 2.10 (Depth) *Let* G *be a global type. For* $\sigma \in paths(\mathsf{G})$ *we define*

$$depth(\sigma,\mathsf{p}) = \text{infimum}\{n \mid \mathsf{p} \in \mathsf{prt}(\sigma[n])\}$$

and define $depth(\mathsf{G},\mathsf{p})$*, the* depth *of* p *in* G*, as follows:*

$$depth(\mathsf{G},\mathsf{p}) = \begin{cases} \text{supremum}\{depth(\sigma,\mathsf{p}) \mid \sigma \in paths(\mathsf{G})\} & \text{if } \mathsf{p} \in \mathsf{prt}(\mathsf{G}) \\ 0 & \text{otherwise} \end{cases}$$

Note that $depth(\mathsf{G},\mathsf{p}) = 0$ iff $\mathsf{p} \notin \mathsf{prt}(\mathsf{G})$. Moreover, if $\mathsf{p} \in \mathsf{prt}(\mathsf{G})$, but for some $\sigma \in paths(\mathsf{G})$ it is the case that $\mathsf{p} \notin \mathsf{prt}(\sigma[n])$ for all $n \leq |\sigma|$, then $depth(\sigma,\mathsf{p}) = \text{infimum}\,\emptyset = \infty$. Hence, if p is a participant of a global type G and there is some path in G where p does not occur, then $depth(\mathsf{G},\mathsf{p}) = \infty$.

Definition 2.11 (Boundedness) *A global type* G *is* bounded *if* $depth(\mathsf{G}',\mathsf{p})$ *is finite for all participants* $\mathsf{p} \in \mathsf{prt}(\mathsf{G}')$ *and all types* G' *which occur in* G.

Intuitively, this means that if $\mathsf{p} \in \mathsf{prt}(\mathsf{G}')$ for a type G' which occurs in G, then the search for an interaction of the shape $\mathsf{p}\lambda\mathsf{q}$ or $\mathsf{q}\lambda\mathsf{p}$ along a path $\sigma \in paths(\mathsf{G}')$ terminates (and recall that G' can be infinite, in which case G is such). Hence the name.

Example 2 of [3] shows the necessity of considering all types occurring in a global type when defining boundedness and that also a finite global type can be unbounded.

Since global types are regular, the boundedness condition is decidable. We shall allow only bounded global types in typing sessions.

Example 2.12 (A global type for the social media example) The intended overall behaviour of the multiparty session $\mathbb{M}$ in Example 2.8, up to the point where the request of p is possibly accepted by u, is described by the following global type G.

$$\mathsf{G} = \mathsf{q} \to \mathsf{p}{:}\textsc{hello}.\,\mathsf{p} \to \mathsf{u}{:}\textsc{req}.\,\mathsf{u} \to \mathsf{p}{:}\begin{cases} \textsc{dnd}.\,\mathsf{u} \to \mathsf{q}{:}\textsc{dnd}.\,\mathsf{G} \\ \textsc{grtd}.\,\mathsf{u} \to \mathsf{q}{:}\textsc{grtd}.\,\mathsf{p} \to \mathsf{q}{:}\textsc{hello} \end{cases}$$

Typability of $\mathbb{M}$ with G – by means of the type system defined in the next section – will ensure (see Theorems 3.6 and 3.7 below) that the behaviours of participants of G, but u, will perfectly adhere to what G describes. ◇

We conclude this section by defining the standard LTS for global types. By means of such LTS we formalise the intended meaning of global types as overall (possibly partial) descriptions of sessions' behaviours. It will be used in the next section to prove the properties of Subject Reduction and Session Fidelity which, in our setting, will slightly differ from the standard ones [16, 17].

Definition 2.13 (LTS for Global Types) *The* labelled transition system (LTS) for global types *is specified by the following axiom and rule:*

$$[\textsc{Ecomm}]\ \frac{}{\mathsf{p} \to \mathsf{q} : \{\lambda_i.\mathsf{G}_i\}_{i\in I} \xrightarrow{\mathsf{p}\lambda_j\mathsf{q}} \mathsf{G}_j}\quad j \in I$$

$$[\textsc{Icomm}]\ \frac{\mathsf{G}_i \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{G}_i' \quad \forall i \in I \qquad \{\mathsf{p},\mathsf{q}\} \cap \{\mathsf{r},\mathsf{s}\} = \emptyset}{\mathsf{r} \to \mathsf{s} : \{\lambda_i.\mathsf{G}_i\}_{i\in I} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{r} \to \mathsf{s} : \{\lambda_i.\mathsf{G}_i'\}_{i\in I}}$$

Axiom [Ecomm] formalises the fact that, in a session exposing the behaviour $\mathsf{p} \to \mathsf{q} : \{\lambda_i.\mathsf{G}_i\}_{i\in I}$, there are participants p and q ready to exchange a message λ_j for any $j \in I$, the former as sender and the latter as receiver. If such a communication is actually performed, the resulting session will expose the behaviour G_j.

Rule [Icomm] makes sense since, in a global type $\mathsf{r} \to \mathsf{s} : \{\lambda_i.\mathsf{G}_i\}_{i\in I}$, communications involving participants p and q, ready to interact with each other uniformly in all branches, can be performed if neither of them is involved in a previous interaction between r and s. In this case, the interaction between p and q is independent of the choice of r, and may be executed before it.

3 Type System and its Properties

As in [3, 9, 13], our type assignment allows for a simple treatment of many technical issues, by avoiding projections, local types and subtyping [16, 17]. The novelty of the type system we present in this section with respect to those in [3, 9, 13] is that the judgments are parametrised by a set $\mathscr{P}$ of participants. These are the participants whose Lock-freedom we do not care about. The simplicity of our calculus allows us to formulate a type system deriving directly global types for multiparty sessions, i.e. judgments of the form $\mathsf{G} \vdash_{\mathscr{P}} \mathbb{M}$ (where G is bounded). Here and in the following the double line indicates that the rules are interpreted coinductively [27, Chapter 21].

Definition 3.1 (Type system) *The type system $\vdash_{\mathscr{P}}$ is defined by the following axiom and rules, where sessions are considered modulo structural congruence:*

$$[\texttt{End}]\ \texttt{End} \vdash_{\emptyset} \mathsf{p}[\mathbf{0}]$$

$$[\textsc{Comm}]\ \frac{\begin{array}{c}\mathsf{G}_i \vdash_{\mathscr{P}_i} \mathsf{p}[P_i] \parallel \mathsf{q}[Q_i] \parallel \mathbb{M} \\ (\mathsf{prt}(\mathsf{G}_i) \cup \mathscr{P}_i) \setminus \{\mathsf{p},\mathsf{q}\} = \mathsf{prt}(\mathbb{M}) \quad \forall i \in I\end{array}}{\mathsf{G} \vdash_{\mathscr{P}} \mathsf{p}[\mathsf{q}!\{\lambda_i.P_i\}_{i\in I}] \parallel \mathsf{q}[\mathsf{p}?\{\lambda_j.Q_j\}_{j\in J}] \parallel \mathbb{M}}\quad \begin{array}{c}\mathsf{G} = \mathsf{p} \to \mathsf{q} : \{\lambda_i.\mathsf{G}_i\}_{i\in I} \quad \mathsf{G}\text{ is bounded} \\ \mathscr{P} = \bigcup_{i\in I}\mathscr{P}_i \quad I \subseteq J\end{array}$$

$$[\textsc{Weak}]\ \frac{\mathsf{G} \vdash_{\mathscr{P}_1} \mathbb{M}_1}{\mathsf{G} \vdash_{\mathscr{P}_1 \cup \mathscr{P}_2} \mathbb{M}_1 \parallel \mathbb{M}_2}\quad \mathscr{P}_2 = \mathsf{prt}(\mathbb{M}_2) \neq \emptyset$$

Axiom [End] simply states that the null session has the behaviour described by End. In the null session there is obviously no participant whose Lock-freedom we do not care about, hence the $\emptyset$ subscript. An alternative and sound version of this axiom could be $\mathtt{End} \vdash_{\mathsf{prt}(\mathbb{M})} \mathbb{M}$ for any $\mathbb{M}$. Such a judgment however can be easily derived using Rule [Weak].

Rule [Comm] just adds simultaneous communications to global types and to corresponding processes inside sessions. Since the set $\mathscr{P}_i$ contains the ignored participants in branch i, the possibly non lock-free participants in the conclusion must be $\bigcup_{i\in I}\mathscr{P}_i$. Note that Rule [Comm] allows more inputs than corresponding outputs, in agreement with the condition in Rule [Comm-T]. It also allows more branches in the input process than in the global type, just mimicking the subtyping for session types [14]. Instead, the number of branches in the output process and the global type must be the same. This does not restrict typability, while it improves Session Fidelity (by arguing, respectively, as in [4] and [3]). The condition $(\mathsf{prt}(\mathsf{G}_i) \cup \mathscr{P}_i) \setminus \{\mathsf{p},\mathsf{q}\} = \mathsf{prt}(\mathbb{M})$ for all $i \in I$ ensures that the participants in the session are exactly those we keep track of either in G and/or in $\mathscr{P}$. This condition prevents, for example, to derive $\mathsf{G} \vdash_\emptyset \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathsf{r}[R]$, where $\mathsf{G} = \mathsf{p} \to \mathsf{q} : \lambda.\mathsf{G}$, $P = \mathsf{q}!\lambda.P$, $Q = \mathsf{p}?\lambda.Q$ and $R \neq \mathbf{0}$ is arbitrary. Note that, instead, it is possible to derive $\mathsf{G} \vdash_{\{\mathsf{r}\}} \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathsf{r}[R]$ with $R \neq \mathbf{0}$ arbitrary. Lock-freedom can be ensured only for the participants of $\mathbb{M}$ not belonging to $\mathscr{P}$.

Rule [Weak] enables to type check just a sub session as far as we do not care about the Lock-freedom of the participants of the rest of the session. We keep track of such participants in the subscript of the entailment symbol. The condition $\mathscr{P}_2 \neq \emptyset$ forbids infinite applications of this rule. This condition allows us to use coinduction on typing derivations.

Sessions are considered modulo structural congruence in typing rules following [3, 9, 13]. Clearly this could be avoided by adding an obvious typing rule, but we prefer to have a lighter type system.

The regularity of processes and global types ensures the decidability of type checking.

Example 3.2 (Typing of the social media) Let P, Q and U be defined as in Example 2.8 and

$$P_1 = \mathsf{u}!\textsc{req}.\mathsf{u}?\{\textsc{dnd}.P, \textsc{grtd}.P_2\},\ P_2 = \mathsf{q}!\textsc{hello},\ Q_1 = \mathsf{u}?\{\textsc{dnd}.Q, \textsc{grtd}.Q_2\},\ Q_2 = \mathsf{p}?\textsc{hello}$$
$$U_1 = \mathsf{p}!\{\textsc{dnd}.\mathsf{q}!\textsc{dnd}.U, \textsc{grtd}.\mathsf{q}!\textsc{grtd}.U\}$$

Moreover, let G be defined as in Example 2.12 and $\mathsf{G}_1 = \mathsf{p} \to \mathsf{u}{:}\textsc{req}.\,\mathsf{G}_2$

$$\mathscr{D} = \dfrac{\dfrac{\dfrac{\dfrac{\mathscr{D}}{\mathsf{G}_3 \vdash_{\{\mathsf{u}\}} \mathsf{p}[P] \parallel \mathsf{q}[Q_1] \parallel \mathsf{u}[\mathsf{q}!\textsc{dnd}.U]} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{End} \vdash_\emptyset \mathsf{p}[\mathbf{0}] \parallel \mathsf{q}[\mathbf{0}]}[\mathtt{End}]}{\mathsf{G}_5 \vdash_\emptyset \mathsf{p}[P_2] \parallel \mathsf{q}[Q_2]}}{\mathsf{G}_5 \vdash_{\{\mathsf{u}\}} \mathsf{p}[P_2] \parallel \mathsf{q}[Q_2] \parallel \mathsf{u}[U]}[\textsc{Weak}]}{\mathsf{G}_4 \vdash_{\{\mathsf{u}\}} \mathsf{p}[P_2] \parallel \mathsf{q}[Q_1] \parallel \mathsf{u}[\mathsf{q}!\textsc{grtd}.U]}}{\mathsf{G}_2 \vdash_{\{\mathsf{u}\}} \mathsf{p}[\mathsf{u}?\{\textsc{dnd}.P, \textsc{grtd}.P_2\}] \parallel \mathsf{q}[Q_1] \parallel \mathsf{u}[U_1]}}{\mathsf{G}_1 \vdash_{\{\mathsf{u}\}} \mathsf{p}[P_1] \parallel \mathsf{q}[Q_1] \parallel \mathsf{u}[U]}}{\mathsf{G} \vdash_{\{\mathsf{u}\}} \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathsf{u}[U]}$$

Figure 1: A type derivation for the social media.

$$\mathsf{G}_2 = \mathsf{u} \to \mathsf{p}:\begin{cases} \textsc{dnd}.\,\mathsf{G}_3 \\ \textsc{grtd}.\,\mathsf{G}_4 \end{cases} \qquad \mathsf{G}_3 = \mathsf{u} \to \mathsf{q}:\textsc{dnd}.\,\mathsf{G} \quad \mathsf{G}_4 = \mathsf{u} \to \mathsf{q}:\textsc{grtd}.\,\mathsf{G}_5 \quad \mathsf{G}_5 = \mathsf{p} \to \mathsf{q}:\textsc{hello}$$

Figure 1 shows a derivation of the global type G of Example 2.12 for the multiparty session $\mathbb{M}$ of Example 2.8. The missing rule names are all [Comm]. Note how in the leftmost branch of the derivation it is possible to get $\{\mathsf{u}\}$ as subscript without recurring to Rule [Weak] thanks to the infiniteness of the branch. For the same motivation, in case we had $P = \mathsf{q}?\textsc{hello}.\mathsf{u}!\textsc{req}.\mathsf{u}?\{\textsc{dnd}.P, \textsc{grtd}.P'\}$ with $P' = \mathsf{q}!\textsc{hello}.P'$ and $Q = \mathsf{p}!\textsc{hello}.\mathsf{u}?\{\textsc{dnd}.Q, \textsc{grtd}.Q'\}$ with $Q' = \mathsf{p}?\textsc{hello}.Q'$ (namely in case p and q kept on indefinitely exchanging HELLO messages after receiving the GRTD message) the whole resulting session would be typable without recurring to Rule [Weak]. ◇

We note that session participants are of three different kinds in a typing judgment:

1. the lock-free participants which behave as pointed out by the global type; these participants occur in the global type but do not belong to the set of ignored participants;
2. the participants which "partially" behave as pointed out by the global type and can get stuck; these participants occur in the global type and belong to the set of ignored participants;
3. the participants which behave in an umpredictable way; these participants do not occur in the global type but belong to the set of ignored participants.

We observe also that $\vdash_\emptyset$ coincides with the typing relation of [3].

In the remainder of this section we will show the main properties of our type system, i.e. Subject Reduction, Session Fidelity and $\mathscr{P}$-excluded Lock-freedom. We start with some lemmas which are handy for the subsequent proofs. All proofs are by coinduction on $\mathsf{G} \vdash_\mathscr{P} \mathbb{M}$ and by cases on the last applied rule.

The first lemma states that, when $\mathsf{G} \vdash_\mathscr{P} \mathbb{M}$, all participants of $\mathbb{M}$ must be participants of G and/or must belong to the set $\mathscr{P}$.

Lemma 3.3 $\mathsf{G} \vdash_\mathscr{P} \mathbb{M}$ *implies* $\mathsf{prt}(\mathsf{G}) \cup \mathscr{P} = \mathsf{prt}(\mathbb{M})$.

Proof. Rule [Comm]. Immediate by the condition $(\mathsf{prt}(\mathsf{G}_i) \cup \mathscr{P}_i) \setminus \{\mathsf{p},\mathsf{q}\} = \mathsf{prt}(\mathbb{M})$ for all $i \in I$.

Rule [Weak]. In such a case, $\mathbb{M} \equiv \mathbb{M}_1 \parallel \mathbb{M}_2$ and $\mathscr{P} = \mathscr{P}_1 \cup \mathscr{P}_2$. By coinduction we get $\mathsf{prt}(\mathsf{G}) \cup \mathscr{P}_1 = \mathsf{prt}(\mathbb{M}_1)$. We hence get the thesis by the condition $\mathscr{P}_2 = \mathsf{prt}(\mathbb{M}_2)$. □

By the above lemma, from $\mathsf{G} \vdash_\mathscr{P} \mathbb{M}$ it is immediate to get also that $\mathsf{p} \in \mathsf{prt}(\mathsf{G})$ implies $\mathsf{p} \in \mathsf{prt}(\mathbb{M})$ and that $\mathsf{p} \in \mathsf{prt}(\mathbb{M})$ and $\mathsf{p} \notin \mathscr{P}$ imply $\mathsf{p} \in \mathsf{prt}(\mathsf{G})$.

The process of a participant which does not occur in the global type can be freely replaced, since typing ensures nothing about the behaviour of this participant.

Lemma 3.4 *If* $\mathsf{G} \vdash_\mathscr{P} \mathsf{p}[P] \parallel \mathbb{M}$ *and* $\mathsf{p} \in \mathsf{prt}(\mathbb{M}) \setminus \mathsf{prt}(\mathsf{G})$, *then* $\mathsf{G} \vdash_{\mathscr{P}'} \mathsf{p}[P'] \parallel \mathbb{M}$ *with* $\mathscr{P}' \subseteq \mathscr{P}$ *for an arbitrary* P'.

Proof. Rule [Comm]. Then $\mathsf{G} = \mathsf{q} \to \mathsf{r} : \{\lambda_i.\mathsf{G}_i\}_{i\in I}$ and $\mathbb{M} \equiv \mathsf{q}[\mathsf{r}!\{\lambda_i.Q_i\}_{i\in I}] \parallel \mathsf{r}[\mathsf{q}?\{\lambda_j.R_j\}_{j\in J}] \parallel \mathbb{M}_0$ and $I \subseteq J$ and $\mathsf{G}_i \vdash_{\mathscr{P}_i} \mathsf{p}[P] \parallel \mathsf{q}[Q_i] \parallel \mathsf{r}[R_i] \parallel \mathbb{M}_0$ for all $i \in I$ with $\mathscr{P} = \bigcup_{i\in I} \mathscr{P}_i$. By coinduction we get $\mathsf{G}_i \vdash_{\mathscr{P}'_i} \mathsf{p}[P'] \parallel \mathsf{q}[Q_i] \parallel \mathsf{r}[R_i] \parallel \mathbb{M}_0$ with $\mathscr{P}'_i \subseteq \mathscr{P}_i$ for all $i \in I$ for an arbitrary P'. We conclude using Rule [Comm].

Rule [Weak]. Then $\mathscr{P} = \mathscr{P}_1 \cup \mathscr{P}_2$ and $\mathsf{p}[P] \parallel \mathbb{M} \equiv \mathbb{M}_1 \parallel \mathbb{M}_2$ and $\mathsf{G} \vdash_{\mathscr{P}_1} \mathbb{M}_1$. If $\mathbb{M}_1 \equiv \mathsf{p}[P] \parallel \mathbb{M}'_1$ by coinduction $\mathsf{G} \vdash_{\mathscr{P}'_1} \mathsf{p}[P'] \parallel \mathbb{M}'_1$ with $\mathscr{P}'_1 \subseteq \mathscr{P}_1$ for arbitrary P' and we conclude using Rule [Weak]. If $\mathbb{M}_2 \equiv \mathsf{p}[P] \parallel \mathbb{M}'_2$ we can apply Rule [Weak] to $\mathsf{G} \vdash_{\mathscr{P}_1} \mathbb{M}_1$ and $\mathsf{p}[P'] \parallel \mathbb{M}'_2$ for arbitrary P'. □

Note that in previous lemma $\mathscr{P}' = \mathscr{P}$ unless $P' = \mathbf{0}$ and in this case $\mathscr{P}' \cup \{\mathsf{p}\} = \mathscr{P}$.

If $\mathsf{p}[\mathsf{q}\,\dagger\,\{\lambda_i.P_i\}_{i\in I}] \in \mathbb{M}$ we say that q is the *top partner* of p and we write $\mathsf{tp}(\mathbb{M},\mathsf{p}) = \mathsf{q}$. Note that we can have $\mathsf{q} \notin \mathsf{prt}(\mathbb{M})$ or $\mathsf{tp}(\mathbb{M},\mathsf{q}) \neq \mathsf{p}$. For example, if $\mathbb{M} \equiv \mathsf{p}[\mathsf{q}\,\dagger\,\{\lambda_i.P_i\}_{i\in I}] \parallel \mathsf{q}[\mathsf{r}\,\dagger\,\{\lambda'_j.Q_j\}_{j\in J}]$, we have $\mathsf{tp}(\mathbb{M},\mathsf{p}) = \mathsf{q}$ and $\mathsf{tp}(\mathbb{M},\mathsf{q}) = \mathsf{r} \neq \mathsf{p}$.

Typing ensures that if a participant occurs in a global type then also her top partner occurs in the global type.

Lemma 3.5 *If* $\mathsf{G} \vdash_{\mathscr{P}} \mathbb{M}$ *and* $\mathsf{p} \in \mathsf{prt}(\mathsf{G})$*, then* $\mathsf{tp}(\mathbb{M},\mathsf{p}) \in \mathsf{prt}(\mathsf{G})$.

Proof. By Lemma 3.3 and $\mathsf{p} \in \mathsf{prt}(\mathsf{G})$ we have that $\mathsf{p} \in \mathsf{prt}(\mathbb{M})$ and then $\mathsf{tp}(\mathbb{M},\mathsf{p})$ is defined. So, let $\mathsf{tp}(\mathbb{M},\mathsf{p}) = \mathsf{q}$.

Rule [Comm]. Then $\mathsf{G} = \mathsf{r} \to \mathsf{s} : \{\lambda_i.\mathsf{G}_i\}_{i\in I}$ and $\mathbb{M} \equiv \mathsf{r}[\mathsf{s}!\{\lambda_i.R_i\}_{i\in I}] \parallel \mathsf{s}[\mathsf{r}?\{\lambda_j.S_j\}_{j\in J}] \parallel \mathbb{M}_0$ with $I \subseteq J$ and $\mathsf{G}_i \vdash_{\mathscr{P}_i} \mathsf{r}[R_i] \parallel \mathsf{s}[S_i] \parallel \mathbb{M}_0$ for all $i \in I$ with $\mathscr{P} = \bigcup_{i\in I} \mathscr{P}_i$. If $\mathsf{p} \in \{\mathsf{r},\mathsf{s}\}$, then $\{\mathsf{p},\mathsf{q}\} = \{\mathsf{r},\mathsf{s}\}$ and we are done. Otherwise $\mathsf{tp}(\mathbb{M},\mathsf{p}) = \mathsf{q}$ implies $\mathsf{tp}(\mathsf{r}[R_i] \parallel \mathsf{s}[S_i] \parallel \mathbb{M}_0,\mathsf{p}) = \mathsf{q}$ for all $i \in I$. Moreover $\mathsf{p} \in \mathsf{prt}(\mathsf{G})$ implies $\mathsf{p} \in \mathsf{prt}(\mathsf{G}_i)$ for all $i \in I$ since G is bounded. By coinduction we get $\mathsf{q} \in \mathsf{prt}(\mathsf{G}_i)$ for all $i \in I$. We conclude $\mathsf{q} \in \mathsf{prt}(\mathsf{G})$.

Rule [Weak]. Then $\mathscr{P} = \mathscr{P}_1 \cup \mathscr{P}_2$, $\mathbb{M} \equiv \mathbb{M}_1 \parallel \mathbb{M}_2$ and $\mathsf{G} \vdash_{\mathscr{P}_1} \mathbb{M}_1$. Since by Lemma 3.3 $\mathsf{p} \in \mathsf{prt}(\mathsf{G})$ implies $\mathsf{p} \in \mathsf{prt}(\mathbb{M}_1)$ we have $\mathsf{tp}(\mathbb{M}_1,\mathsf{p}) = \mathsf{q}$. We get by coinduction $\mathsf{q} \in \mathsf{prt}(\mathsf{G})$. □

In our particular setting, what Subject Reduction ensures depends on which participants we consider (unlike its standard version, e.g. in [4] and [3]). In particular, it ensures that, when the involved participants occur in the global types, the transitions of well-typed sessions are mimicked by those of global types (namely they proceed as prescribed by the global type). Otherwise the reduced session can be typed by the same global type. Key for this proof is Lemma 3.5, which ensures that the communicating participants either both occur or both do not occur in the global type.

Theorem 3.6 (Subject Reduction) *Let* $\mathsf{G} \vdash_{\mathscr{P}} \mathbb{M}$ *and* $\mathbb{M} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathbb{M}'$.

i) If $\{\mathsf{p},\mathsf{q}\} \subseteq \mathsf{prt}(\mathsf{G})$*, then* $\mathsf{G} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{G}'$ *and* $\mathsf{G}' \vdash_{\mathscr{P}'} \mathbb{M}'$ *with* $\mathscr{P}' \subseteq \mathscr{P}$.

ii) If $\mathsf{p},\mathsf{q} \notin \mathsf{prt}(\mathsf{G})$*, then* $\mathsf{G} \vdash_{\mathscr{P}'} \mathbb{M}'$ *with* $\mathscr{P}' \subseteq \mathscr{P}$.

Proof. From $\mathbb{M} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathbb{M}'$ we get $\mathbb{M} \equiv \mathsf{p}[\mathsf{q}!\{\lambda_i.P_i\}_{i\in I}] \parallel \mathsf{q}[\mathsf{p}?\{\lambda_j.Q_j\}_{j\in J}] \parallel \mathbb{M}_0$ with $I \subseteq J$ and $\mathbb{M}' \equiv \mathsf{p}[P_l] \parallel \mathsf{q}[Q_l] \parallel \mathbb{M}_0$ and $\lambda = \lambda_l$ for some $l \in I$. Note that Lemma 3.5 implies either $\{\mathsf{p},\mathsf{q}\} \subseteq \mathsf{prt}(\mathsf{G})$ or $\mathsf{p},\mathsf{q} \notin \mathsf{prt}(\mathsf{G})$.

ii). In this case $\mathsf{G} \vdash_{\mathscr{P}} \mathbb{M}$ implies $\mathsf{G} \vdash_{\mathscr{P}'} \mathsf{p}[P_l] \parallel \mathsf{q}[Q_l] \parallel \mathbb{M}_0$ with $\mathscr{P}' \subseteq \mathscr{P}$ by Lemma 3.4.

i). The proof is by coinduction on $\mathsf{G} \vdash_{\mathscr{P}} \mathsf{p}[P] \parallel \mathbb{M}$ and by cases on the last applied rule.

Rule [Comm]. We get $\mathsf{G} = \mathsf{r} \to \mathsf{s} : \{\lambda'_h.\mathsf{G}_h\}_{h\in H}$ and $\mathbb{M} \equiv \mathsf{r}[\mathsf{s}!\{\lambda'_h.R_h\}_{h\in H}] \parallel \mathsf{s}[\mathsf{r}?\{\lambda'_k.S_k\}_{k\in K}] \parallel \mathbb{M}_1$ and $H \subseteq K$ and $\mathsf{G}_h \vdash_{\mathscr{P}_h} \mathsf{r}[R_h] \parallel \mathsf{s}[S_h] \parallel \mathbb{M}_1$ for all $h \in H$ with $\mathscr{P} = \bigcup_{h\in H} \mathscr{P}_h$. If with $\mathsf{p} = \mathsf{r}$ and $\mathsf{q} = \mathsf{s}$, then $I = H$, $J = K$ and $\lambda_i = \lambda'_i$ for all $i \in I$. We conclude $\mathsf{G} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{G}_l$ and $\mathsf{G}_l \vdash_{\mathscr{P}_l} \mathbb{M}'$. Otherwise $\{\mathsf{p},\mathsf{q}\} \cap \{\mathsf{r},\mathsf{s}\} = \emptyset$, which implies $\mathsf{r}[R_h] \parallel \mathsf{s}[S_h] \parallel \mathbb{M}_1 \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{r}[R_h] \parallel \mathsf{s}[S_h] \parallel \mathbb{M}'_1$ for all $h \in H$ by Fact 2.4. Moreover $\{\mathsf{p},\mathsf{q}\} \subseteq \mathsf{prt}(\mathsf{G}_h)$ for all $h \in H$ since G is bounded. By coinduction we get $\mathsf{G}_h \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{G}'_h$ and $\mathsf{G}'_h \vdash_{\mathscr{P}'_h} \mathsf{r}[R_h] \parallel \mathsf{s}[S_h] \parallel \mathbb{M}'_1$ for some G'_h and $\mathscr{P}'_h \subseteq \mathscr{P}_h$ and for all $h \in H$. We conclude $\mathsf{G} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{r} \to \mathsf{s} : \{\lambda'_h.\mathsf{G}'_h\}_{h\in H}$ using Rule [iComm] and $\mathsf{r} \to \mathsf{s} : \{\lambda'_h.\mathsf{G}'_h\}_{h\in H} \vdash_{\mathscr{P}'} \mathbb{M}'$ with $\mathscr{P}' = \bigcup_{h\in H} \mathscr{P}'_h$ using Rule [Comm].

Rule [Weak]. In this case $\mathbb{M} \equiv \mathbb{M}_1 \parallel \mathbb{M}_2$ and $\mathscr{P} = \mathscr{P}_1 \cup \mathscr{P}_2$ and $\mathsf{G} \vdash_{\mathscr{P}_1} \mathbb{M}_1$. From $\{\mathsf{p},\mathsf{q}\} \subseteq \mathsf{prt}(\mathsf{G})$, by Lemma 3.3 we get $\{\mathsf{p},\mathsf{q}\} \subseteq \mathsf{prt}(\mathbb{M}_1)$ which implies $\mathbb{M}_1 \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathbb{M}'_1$. By coinduction we get $\mathsf{G} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{G}'$ and $\mathsf{G}' \vdash_{\mathscr{P}'_1} \mathbb{M}'_1$ with $\mathscr{P}'_1 \subseteq \mathscr{P}_1$. We conclude using Rule [Weak], since by construction $\mathbb{M}' \equiv \mathbb{M}'_1 \parallel \mathbb{M}_2$. □

We note that Subject Reduction, as formulated in previous theorem, fails if we allow unbounded global types. Let $\mathsf{G} = \mathsf{p} \to \mathsf{q}{:}\{\lambda_1.\mathsf{r} \to \mathsf{s}{:}\lambda, \lambda_2.\mathsf{G}\}$ and $\mathbb{M} \equiv \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathsf{r}[\mathsf{s}!\lambda] \parallel \mathsf{s}[\mathsf{r}?\lambda]$ and $P = \mathsf{q}!\{\lambda_1, \lambda_2.P\}$ and $Q = \mathsf{p}?\{\lambda_1, \lambda_2.Q\}$. Then we have $\mathsf{G} \vdash_\emptyset \mathbb{M}$ and $\mathbb{M} \xrightarrow{\mathsf{r}\lambda\mathsf{s}} \mathsf{p}[P] \parallel \mathsf{q}[Q]$, but there is no transition labelled $\mathsf{r}\lambda\mathsf{s}$ starting from G. Note that the session $\mathbb{M}$ can be typed, still with the $\emptyset$ subscript, by the bounded global type $\mathsf{G}' = \mathsf{r} \to \mathsf{s}{:}\lambda.\mathsf{p} \to \mathsf{q}{:}\{\lambda_1, \lambda_2.\mathsf{G}'\}$.

Session Fidelity ensures that the communications in a session typed by a global type proceed at least as prescribed by the global type.

Theorem 3.7 (Session Fidelity) *Let* $\mathsf{G} \vdash_{\mathscr{P}} \mathbb{M}$ *and* $\mathsf{G} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{G}'$. *Then* $\mathbb{M} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathbb{M}'$ *and* $\mathsf{G}' \vdash_{\mathscr{P}'} \mathbb{M}'$ *with* $\mathscr{P}' \subseteq \mathscr{P}$.

Proof. The proof is by coinduction on the derivation of $\mathsf{G} \vdash_{\mathscr{P}} \mathbb{M}$ and by cases on the last applied rule.

Rule [Comm]. The proof is by induction on the number t of transition rules used to derive $\mathsf{G} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{G}'$.

Case $t = 1$. Then $\mathsf{G} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{G}'$ is the Axiom [Ecomm] and $\mathsf{G} = \mathsf{p} \to \mathsf{q} : \{\lambda_i.\mathsf{G}_i\}_{i\in I}$, where $\lambda = \lambda_l$ and $\mathsf{G}' = \mathsf{G}_l$ with $l \in I$. We get $\mathbb{M} \equiv \mathsf{p}[\mathsf{q}!\{\lambda_i.P_i\}_{i\in I}] \parallel \mathsf{q}[\mathsf{p}?\{\lambda_j.Q_j\}_{j\in J}] \parallel \mathbb{M}_0$ with $I \subseteq J$ and $\mathsf{G}_i \vdash_{\mathscr{P}_i} \mathsf{p}[P_i] \parallel \mathsf{q}[Q_i] \parallel \mathbb{M}_0$ for all $i \in I$ with $\mathscr{P} = \bigcup_{i\in I} \mathscr{P}_i$. Then we conclude $\mathbb{M} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{p}[P_l] \parallel \mathsf{q}[Q_l] \parallel \mathbb{M}_0$ by Rule [Comm-T] and $\mathsf{G}_l \vdash_{\mathscr{P}_l} \mathsf{p}[P_l] \parallel \mathsf{q}[Q_l] \parallel \mathbb{M}_0$.

Case $t > 1$. Then $\mathsf{G} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{G}'$ is the conclusion of Rule [Icomm]. Moreover $\mathsf{G} = \mathsf{r} \to \mathsf{s} : \{\lambda'_h.\mathsf{G}_h\}_{h\in H}$ and $\mathsf{G}' = \mathsf{r} \to \mathsf{s} : \{\lambda'_h.\mathsf{G}'_h\}_{h\in H}$ and $\mathsf{G}_h \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{G}'_h$ for all $h \in H$ and $\{\mathsf{p},\mathsf{q}\} \cap \{\mathsf{r},\mathsf{s}\} = \emptyset$. We get

$$\mathbb{M} \equiv \mathsf{r}[\mathsf{s}!\{\lambda'_h.R_h\}_{h\in H}] \parallel \mathsf{s}[\mathsf{r}?\{\lambda'_k.S_k\}_{k\in K}] \parallel \mathbb{M}_1$$

with $H \subseteq K$ and $\mathsf{G}_h \vdash_{\mathscr{P}_h} \mathsf{r}[R_h] \parallel \mathsf{s}[S_h] \parallel \mathbb{M}_1$ for all $h \in H$ with $\mathscr{P} = \bigcup_{h\in H} \mathscr{P}_h$.

By induction $\mathsf{r}[R_h] \parallel \mathsf{s}[S_h] \parallel \mathbb{M}_1 \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathbb{M}'_h$ and $\mathsf{G}'_h \vdash_{\mathscr{P}'_h} \mathbb{M}'_h$ with $\mathscr{P}'_h \subseteq \mathscr{P}_h$ for all $h \in H$. The condition $\{\mathsf{p},\mathsf{q}\} \cap \{\mathsf{r},\mathsf{s}\} = \emptyset$ ensures that the reduction $\mathsf{r}[R_h] \parallel \mathsf{s}[S_h] \parallel \mathbb{M}_1 \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathbb{M}'_h$ does not modify the processes of participants r and s. Moreover the processes of participants p and q are the same in $\mathbb{M}'_h$ for all $h \in H$. This implies $\mathbb{M}'_h \equiv \mathsf{r}[R_h] \parallel \mathsf{s}[S_h] \parallel \mathbb{M}''$ for all $h \in H$ and some $\mathbb{M}''$. We conclude $\mathbb{M} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathbb{M}'$ where $\mathbb{M}' = \mathsf{r}[\mathsf{s}!\{\lambda'_h.R_h\}_{h\in H}] \parallel \mathsf{s}[\mathsf{r}?\{\lambda'_k.S_k\}_{k\in K}] \parallel \mathbb{M}''$ using Rule [Comm-T] and $\mathsf{G}' \vdash_{\mathscr{P}'} \mathbb{M}'$ with $\mathscr{P}' = \bigcup_{h\in H} \mathscr{P}'_h$ using Rule [Comm].

Rule [Weak]. In this case $\mathbb{M} \equiv \mathbb{M}_1 \parallel \mathbb{M}_2$ and $\mathscr{P} = \mathscr{P}_1 \cup \mathscr{P}_2$ and $\mathsf{G} \vdash_{\mathscr{P}_1} \mathbb{M}_1$. By coinduction $\mathbb{M}_1 \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathbb{M}'_1$ and $\mathsf{G}' \vdash_{\mathscr{P}'_1} \mathbb{M}'_1$ with $\mathscr{P}'_1 \subseteq \mathscr{P}_1$. Then $\mathbb{M}_1 \parallel \mathbb{M}_2 \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathbb{M}'_1 \parallel \mathbb{M}_2$ and $\mathsf{G}' \vdash_{\mathscr{P}'_1 \cup \mathscr{P}_2} \mathbb{M}'_1 \parallel \mathbb{M}_2$ using Rule [Weak]. □

We can show that typability ensures $\mathscr{P}$-excluded Lock-freedom. This follows from Subject Reduction and Session Fidelity thanks to the boundedness condition.

Theorem 3.8 ($\mathscr{P}$-excluded Lock-freedom) *If* $\mathsf{G} \vdash_{\mathscr{P}} \mathbb{M}$, *then* $\mathbb{M}$ *is* $\mathscr{P}$*-excluded lock-free.*

Proof. By Subject Reduction it is enough to prove that in well-typed sessions no active participant not belonging to $\mathscr{P}$ is prevented to progress. So, let $\mathsf{p} \in \mathsf{prt}(\mathbb{M})$ such that $\mathsf{p} \notin \mathscr{P}$. By Lemma 3.3 we have that $\mathsf{p} \in \mathsf{prt}(\mathbb{M})$ and $\mathsf{p} \notin \mathscr{P}$ imply $\mathsf{p} \in \mathsf{prt}(\mathsf{G})$. We proceed now by induction on $d = \mathit{depth}(\mathsf{G}, \mathsf{p})$.

If $d = 1$ then either $\mathsf{G} = \mathsf{p} \to \mathsf{q} : \{\lambda_i.\mathsf{G}_i\}_{i\in I}$ or $\mathsf{G} = \mathsf{q} \to \mathsf{p} : \{\lambda_i.\mathsf{G}_i\}_{i\in I}$ and $\mathsf{G} \xrightarrow{\Lambda} \mathsf{G}'$ with $\mathsf{p} \in \mathsf{prt}(\Lambda)$ by Axiom [Ecomm]. Then $\mathbb{M} \xrightarrow{\Lambda} \mathbb{M}'$ by Theorem 3.7.

If $d > 1$ then $\mathsf{G} = \mathsf{q} \to \mathsf{r} : \{\lambda_i.\mathsf{G}_i\}_{i\in I}$ with $\mathsf{p} \notin \{\mathsf{q},\mathsf{r}\}$ and $\mathsf{G} \xrightarrow{\mathsf{q}\lambda_i\mathsf{r}} \mathsf{G}_i$ for all $i \in I$ by Axiom [Ecomm]. Induction applies since $\mathit{depth}(\mathsf{G}, \mathsf{p}) > \mathit{depth}(\mathsf{G}_i, \mathsf{p})$ for all $i \in I$. Then, for all $i \in I$, we get $\mathsf{G}_i \xrightarrow{\sigma_i\cdot\Lambda_i} \mathsf{G}'_i$ for

some σ_i, Λ_i with $\mathsf{p} \in \mathsf{prt}(\Lambda_i)$. This implies $\mathsf{G} \xrightarrow{\sigma_i'} \mathsf{G}_i'$ where $\sigma_i' = \mathsf{q}\lambda_i\mathsf{r} \cdot \sigma_i \cdot \Lambda_i$ for all $i \in I$. We conclude $\mathbb{M} \xrightarrow{\sigma_i'} \mathbb{M}_i'$ for all $i \in I$ by Theorem 3.7. □

The following example shows that partial typing allows to type sessions which require unbounded global types in standard type systems [17].

Example 3.9 (Buyer-Seller-Carrier) Let us consider the following session (from [11, Sect.1])

$$\mathbb{M} = \mathsf{b}[B] \parallel \mathsf{s}[S] \parallel \mathsf{c}[\mathsf{s}?\textsc{ship}]$$

where $B = \mathsf{s}!\{\textsc{add}.B, \textsc{pay}\}$ and $S = \mathsf{b}?\{\textsc{add}.S, \textsc{pay}.\mathsf{c}!\textsc{ship}\}$

Such a session implements a system where a buyer can keep on adding goods - sold by a seller - in his shopping cart an unbounded number of times, until he decides to buy the shopping cart's content. In the latter case, the seller informs the carrier for the shipment. Session $\mathbb{M}$ is obviously non lock-free, since participant c would not be able to progress in case s be a seriously disturbed shopaholic who never stop adding goods in his cart. In fact $\mathbb{M}$ cannot be typed in $\vdash_\emptyset$ (which corresponds to the type system of [3]).

In this scenario participant b is a client, whereas s and c are part of the service used by b. It is hence natural to look at $\mathbb{M}$ with a bias towards the client, the one whose good property have to be ensured. As a matter of fact it is possible to ensure the lock freedom of b, namely the $\{\mathsf{s},\mathsf{c}\}$-excluded lock freedom of $\mathbb{M}$, by deriving

$$\mathsf{b} \to \mathsf{s}{:}\{\textsc{add}.\,\mathsf{G},\ \textsc{buy}\} \vdash_{\{\mathsf{s},\mathsf{c}\}} \mathbb{M}$$

as follows

$$\mathscr{D} \quad = \quad \dfrac{\mathscr{D} \qquad \dfrac{\dfrac{}{\mathtt{End} \vdash_\emptyset \mathsf{b}[\mathbf{0}]}\,[\mathtt{End}]}{\mathtt{End} \vdash_{\{\mathsf{s},\mathsf{c}\}} \mathsf{b}[\mathbf{0}] \parallel \mathsf{s}[\mathsf{c}!\textsc{ship}] \parallel \mathsf{c}[\mathsf{s}?\textsc{ship}]}\,[\text{S-WEAK}]}{\mathsf{b} \to \mathsf{s}{:}\{\textsc{add}.\,\mathsf{G},\ \textsc{buy}\} \vdash_{\{\mathsf{s},\mathsf{c}\}} \mathsf{b}[\mathsf{s}!\{\textsc{add}.B, \textsc{pay}\}] \parallel \mathsf{s}[\mathsf{b}?\{\textsc{add}.S, \textsc{pay}.\mathsf{c}!\textsc{ship}\}] \parallel \mathsf{c}[\mathsf{s}?\textsc{ship}]}$$

where $\mathsf{G} = \mathsf{b} \to \mathsf{s}{:}\{\textsc{add}.\,\mathsf{G},\ \textsc{buy}\}$. Note that $\mathsf{b} \to \mathsf{s}{:}\{\textsc{add}.\,\mathsf{G},\ \textsc{buy}\}$ is a bounded global type. ◇

4 Type Inference

In our type system each session can be trivially typed by the `End` type just applying Axiom [End] and Rule [Weak]:

$$\mathtt{End} \vdash_{\mathsf{prt}(\mathbb{M})} \mathbb{M}$$

Clearly, this typing does not provide any information on $\mathbb{M}$. We are interested in more informative typings, if any. In this section, we will describe an algorithm to infer global types and sets of participants from sessions, proving also its soundness and completeness with respect to our type system. In particular, the algorithm applied to a session $\mathbb{M}$ returns all and only those global types which can be assigned to $\mathbb{M}$ with derivations indexed by suitable sets of participants. Note that, since derivations indexed by the same or different sets of participants can assign different global types to a session, the algorithm needs to be non-deterministic in order to be complete.

The first step towards defining such an algorithm is the introduction of a finite representation for global types. Since global types are regular terms, they can be represented, by results in [1, 12], as finite systems of regular syntactic equations formally defined below.

We begin by defining a *global type pattern* as a finite term generated by the following grammar:

$$\mathbb{G} ::= \mathtt{End} \mid \mathsf{p} \to \mathsf{q} : \{\lambda_i.\mathbb{G}_i\}_{i\in I} \mid X$$

where X is a type variable taken from a countably infinite set. We denote by $\mathsf{vars}(\mathbb{G})$ the set of type variables occurring in $\mathbb{G}$. We also need to compute sets of participants, so we define p-set patterns by:

$$\mathbb{P} ::= \mathscr{P} \mid x \mid \mathbb{P} \cup \mathbb{P}$$

where x is a p-set variable taken from a countably infinite set and $\mathscr{P}$ can be any finite set of participants. We denote by $\mathsf{vars}(\mathbb{P})$ the set of p-set variables occurring in $\mathbb{P}$.

We use χ to range over type and p-set variables.

A *substitution* θ is a finite partial map from type variables to global types and from p-set variables to sets of participants. We denote by $\theta + \sigma$ the union of two substitutions such that $\theta(\chi) = \sigma(\chi)$, for all $\chi \in \mathsf{dom}(\theta) \cap \mathsf{dom}(\sigma)$, and by $\mathbb{G}\theta$ (resp. $\mathbb{P}\theta$) the application of θ to $\mathbb{G}$ (resp. $\mathbb{P}$). We define $\theta \preceq \sigma$ if $\mathsf{dom}(\theta) \subseteq \mathsf{dom}(\sigma)$ and $\theta(\chi) = \sigma(\chi)$, for all $\chi \in \mathsf{dom}(\theta)$. Note that, if $\mathsf{vars}(\mathbb{G}) \subseteq \mathsf{dom}(\theta)$, then $\mathbb{G}\theta$ is a global type and if $\mathsf{vars}(\mathbb{P}) \subseteq \mathsf{dom}(\theta)$, then $\mathbb{P}\theta$ is a set of participants.

A *type equation* has shape $X = \mathbb{G}$ and a *(regular) system of type equations* $\mathscr{E}$ is a finite set of equations such that $X = \mathbb{G}_1$ and $X = \mathbb{G}_2 \in \mathscr{E}$ imply $\mathbb{G}_1 = \mathbb{G}_2$. We denote by $\mathsf{dom}(\mathscr{E})$ the set $\{X \mid X = \mathbb{G} \in \mathscr{E}\}$ and by $\mathsf{vars}(\mathscr{E})$ the set $\bigcup\{\mathsf{vars}(\mathbb{G}) \cup \{X\} \mid X = \mathbb{G} \in \mathscr{E}\}$. A *solution* of a system $\mathscr{E}$ is a substitution θ such that $\mathsf{vars}(\mathscr{E}) \subseteq \mathsf{dom}(\theta)$ and, for all $X = \mathbb{G} \in \mathscr{E}$, $\theta(X) = \mathbb{G}\theta$ holds and $\theta(X)$ is bounded. We denote by $\mathsf{sol}(\mathscr{E})$ the set of all solutions of $\mathscr{E}$.

A *p-set equation* has shape $x = \mathbb{P}$. We use E to range over regular systems of p-set equations, which are defined similarly to regular systems of type equations. Also $\mathsf{dom}(E)$, $\mathsf{vars}(E)$ and $\mathsf{sol}(E)$ have the same meanings as for systems of type equations.

A *p-condition* has shape $(\mathsf{prt}(X) \cup x) \setminus \{\mathsf{p},\mathsf{q}\} \doteq \mathscr{P}$ and we let $\mathscr{C}$ range over sets of p-conditions with pairwise distinct type and p-set variables. A substitution θ *agrees* with
- $(\mathsf{prt}(X) \cup x) \setminus \{\mathsf{p},\mathsf{q}\} \doteq \mathscr{P}$ if $(\mathsf{prt}(\theta(X)) \cup \theta(x)) \setminus \{\mathsf{p},\mathsf{q}\} = \mathscr{P}$;
- $\mathscr{C}$, notation $\theta \propto \mathscr{C}$, if θ agrees with all p-conditions in $\mathscr{C}$.

We define $\mathsf{sol}(\mathscr{E},E,\mathscr{C})$ as the set of solutions of $\mathscr{E}$ and E which agree with $\mathscr{C}$, i.e. $\mathsf{sol}(\mathscr{E},E,\mathscr{C}) = \{\theta \in \mathsf{sol}(\mathscr{E}) \cap \mathsf{sol}(E) \mid \theta \propto \mathscr{C}\}$ and note that $\mathscr{E}_1 \subseteq \mathscr{E}_2$, $E_1 \subseteq E_2$, $\mathscr{C}_1 \subseteq \mathscr{C}_2$ imply $\mathsf{sol}(\mathscr{E}_2,E_2,\mathscr{C}_2) \subseteq \mathsf{sol}(\mathscr{E}_1,E_1,\mathscr{C}_1)$.

The algorithm follows essentially the structure of coSLD resolution of coinductive logic programming [29, 30, 31, 2], namely the extension of standard SLD resolution capable to deal with regular terms and coinductive predicates. A *goal* is a triple $(X,\mathbb{M},x)$ of a type variable X, a session $\mathbb{M}$ and a p-set variable x. The algorithm takes a goal $(X,\mathbb{M},x)$ as input, and returns a system of type equations $\mathscr{E}$ and a system of p-set equations E and a set of p-condition $\mathscr{C}$. A solution for the variable X in $\mathscr{E}$ is a global type for the session $\mathbb{M}$ in a derivation indexed by a solution for the variable x in E which satisfies the p-conditions in $\mathscr{C}$. The key idea, borrowed from coinductive logic programming, is to keep track of already encountered goals in order to detect cycles and so avoiding non-termination.

The inference judgements have the following shape: $\mathscr{S} \vdash (X,\mathbb{M},x) \Rightarrow (\mathscr{E},E,\mathscr{C})$, where $\mathscr{S}$ is a set of goals, all with different variables which are all different from X and x. Rules defining the inference algorithm are reported in Figure 2.

For a terminated session the algorithm returns just the two equations $X = \mathtt{End}$ and $x = \emptyset$ and the empty set of conditions (Axiom [A-End]).

In Rule [A-Comm] the algorithm nondeterministically selects one of the matching pairs of processes: $P = \mathsf{q}!\{\lambda_i.P_i\}_{i\in I}$ and $Q = \mathsf{p}?\{\lambda_j.Q_j\}_{j\in J}$, with $I \subseteq J$, i.e. two participants willing to communicate such that the output process can freely choose the message. The algorithm is then recursively applied, for each $i \in I$, to the session where the processes of p and q are, respectively, P_i and Q_i. In each call the goal $(X,\mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M},x)$ is added to the set of goals. At the end of the recursive calls the algorithm

$$\text{[A-End]}\ \frac{}{\mathscr{S} \vdash (X, \mathsf{p}[\mathbf{0}], x) \Rightarrow (\{X = \mathtt{End}\}, \{x = \emptyset\}, \emptyset)}$$

$$\text{[A-Cycle]}\ \frac{}{\mathscr{S}, (Y, \mathbb{M}, y) \vdash (X, \mathbb{M}, x) \Rightarrow (\{X = Y\}, \{x = y\}, \emptyset)}$$

$$\text{[A-Comm]}\ \frac{\mathscr{S}' \vdash (Y_i, \mathsf{p}[P_i] \parallel \mathsf{q}[Q_i] \parallel \mathbb{M}, y_i) \Rightarrow (\mathscr{E}_i, E_i, \mathscr{C}_i)\ \ \forall i \in I}{\mathscr{S} \vdash (X, \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M}, x) \Rightarrow (\mathscr{E}, E, \mathscr{C})}$$

$$\begin{array}{l} \mathscr{S}' = \mathscr{S}, (X, \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M}, x) \\ P = \mathsf{q}!\{\lambda_i.P_i\}_{i\in I} \\ Q = \mathsf{p}?\{\lambda_j.Q_j\}_{j\in J} \qquad I \subseteq J \\ Y_i, y_i \text{ fresh } \forall i \in I \\ \mathscr{E} = \{X = \mathsf{p} \to \mathsf{q} : \{\lambda_i.Y_i\}_{i\in I}\} \cup \bigcup_{i\in I} \mathscr{E}_i \\ E = \{x = \bigcup_{i\in I} y_i\} \cup \bigcup_{i\in I} E_i \\ \mathscr{C} = \{(\mathsf{prt}(Y_i) \cup y_i) \setminus \{\mathsf{p}, \mathsf{q}\} \doteq \mathsf{prt}(\mathbb{M}) \mid \forall i \in I\} \\ \qquad \cup \bigcup_{i\in I} \mathscr{C}_i \end{array}$$

$$\text{[A-Weak]}\ \frac{\mathscr{S}, (X, \mathbb{M}_1 \parallel \mathbb{M}_2, x) \vdash (Y, \mathbb{M}_1, y) \Rightarrow (\mathscr{E}_1, E_1, \mathscr{C})}{\mathscr{S} \vdash (X, \mathbb{M}_1 \parallel \mathbb{M}_2, x) \Rightarrow (\mathscr{E}, E, \mathscr{C})}$$

$$\begin{array}{l} Y, y \text{ fresh} \\ \mathscr{P} = \mathsf{prt}(\mathbb{M}_2) \neq \emptyset \\ \mathscr{E} = \{X = Y\} \cup \mathscr{E}_1 \\ E = \{x = y \cup \mathscr{P}\} \cup E_1 \end{array}$$

Figure 2: Rules of the inference algorithm.

collects all the resulting equations plus another two for the current variables. Note that variables for the goals in the premises are fresh. This is important to ensure that the sets of equations $\mathscr{E}$ and E in the conclusion are indeed regular systems of equations (there is at most one equation for each variable). The new p-condition ensures that the resulting global type associated to X and the resulting set of participants associated to x satisfy the conditions on participants required by Rule [Comm] in Definition 3.1.

In Rule [A-Weak] the algorithm nondeterministically partitions the input session into two subsessions $\mathbb{M}_1$ and $\mathbb{M}_2$ and then it is recursively called on the former. After the recursive call it simply adds the same participants to the session (together with their processes) and to the set of ignored participants by means of the equation $x = y \cup \mathscr{P}$.

Finally, Axiom [A-Cycle] detects cycles: if the session in the current goal appears also in the set $\mathscr{S}$, the algorithm can stop, returning just two equations unifying the type and p-set variables associated with the session together with the empty set of conditions.

Example 4.1 (Inference for the social media) Figure 3 gives a type inference where:
- the processes P, Q, U, P_1, Q_1, U_1, P_2, Q_2 are defined as in Example 3.2 and $U_2 = \mathsf{q}!\textsc{dnd}.U$, $U_3 = \mathsf{q}!\textsc{grtd}.U$;
- the goals are

$$\mathscr{S}_1 = \{(X, \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathsf{u}[U], x)\} \qquad \mathscr{S}_2 = \mathscr{S}_1 \cup \{(Y_1, \mathsf{p}[P_1] \parallel \mathsf{q}[Q_1] \parallel \mathsf{u}[U], y_1)\}$$

$$\mathscr{S}_3 = \mathscr{S}_4 = \mathscr{S}_2 \cup \{(Y_2, \mathsf{p}[\mathsf{u}?\{\textsc{dnd}.P, \textsc{grtd}.P_2\}] \parallel \mathsf{q}[Q_1] \parallel \mathsf{u}[U_1], y_2)\}$$

$$\mathscr{S}_5 = \mathscr{S}_3 \cup \{(Y_3, \mathsf{p}[P] \parallel \mathsf{q}[Q_1] \parallel \mathsf{u}[U_2], y_3)\} \qquad \mathscr{S}_6 = \mathscr{S}_4 \cup \{(Y_4, \mathsf{p}[P_2] \parallel \mathsf{q}[Q_1] \parallel \mathsf{u}[U_3], y_4)\}$$

$$\mathscr{S}_7 = \mathscr{S}_6 \cup \{(Y_6, \mathsf{p}[P_2] \parallel \mathsf{q}[Q_2] \parallel \mathsf{u}[U], y_6)\} \qquad \mathscr{S}_8 = \mathscr{S}_7 \cup \{(Y_7, \mathsf{p}[P_2] \parallel \mathsf{q}[Q_2], y_7)\}$$

- the systems of type equations are

$$\mathscr{E} = \{X = \mathsf{q} \to \mathsf{p}:\textsc{hello}.Y_1\} \cup \mathscr{E}_1 \qquad \mathscr{E}_1 = \{Y_1 = \mathsf{p} \to \mathsf{u}:\textsc{req}.Y_2\} \cup \mathscr{E}_2$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathscr{S}_5 \vdash (Y_5, \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathsf{u}[U], y_5) \Rightarrow (\mathscr{E}_5, E_5, \mathscr{C}_5)}}{\mathscr{S}_3 \vdash (Y_3, \mathsf{p}[P] \parallel \mathsf{q}[Q_1] \parallel \mathsf{u}[U_2], y_3) \Rightarrow (\mathscr{E}_3, E_3, \mathscr{C}_3)} \qquad \dfrac{\dfrac{\dfrac{}{\mathscr{S}_8 \vdash (Y_8, \mathsf{p}[\mathbf{0}] \parallel \mathsf{q}[\mathbf{0}], y_8) \Rightarrow (\mathscr{E}_8, E_8, \mathscr{C}_7)}}{\mathscr{S}_7 \vdash (Y_7, \mathsf{p}[P_2] \parallel \mathsf{q}[Q_2], y_7) \Rightarrow (\mathscr{E}_7, E_7, \mathscr{C}_6)}}{\mathscr{S}_6 \vdash (Y_6, \mathsf{p}[P_2] \parallel \mathsf{q}[Q_2] \parallel \mathsf{u}[U], y_6) \Rightarrow (\mathscr{E}_6, E_6, \mathscr{C}_6)}}{\mathscr{S}_4 \vdash (Y_4, \mathsf{p}[P_2] \parallel \mathsf{q}[Q_1] \parallel \mathsf{u}[U_3], y_4) \Rightarrow (\mathscr{E}_4, E_4, \mathscr{C}_4)}}{\mathscr{S}_2 \vdash (Y_2, \mathsf{p}[\mathsf{u}?\{\text{DND}.P, \text{GRTD}.P_2\}] \parallel \mathsf{q}[Q_1] \parallel \mathsf{u}[U_1], y_2) \Rightarrow (\mathscr{E}_2, E_2, \mathscr{C}_2)}}{\mathscr{S}_1 \vdash (Y_1, \mathsf{p}[P_1] \parallel \mathsf{q}[Q_1] \parallel \mathsf{u}[U], y_1) \Rightarrow (\mathscr{E}_1, E_1, \mathscr{C}_1)}}{\vdash (X, \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathsf{u}[U], x) \Rightarrow (\mathscr{E}, E, \mathscr{C})}$$

Figure 3: A type inference for the social media.

$$\mathscr{E}_2 = \{Y_2 \eqsim \mathsf{u} \to \mathsf{p}{:}\{\text{DND}.Y_3, \text{GRTD}.Y_4\}\} \cup \mathscr{E}_3 \cup \mathscr{E}_4 \qquad \mathscr{E}_3 = \{Y_3 \eqsim \mathsf{u} \to \mathsf{q}{:}\text{DND}.Y_5\} \cup \mathscr{E}_5$$

$$\mathscr{E}_4 = \{Y_4 \eqsim \mathsf{u} \to \mathsf{q}{:}\text{GRTD}.Y_6\} \cup \mathscr{E}_6 \qquad \mathscr{E}_5 = \{Y_5 \eqsim X\}$$

$$\mathscr{E}_6 = \{Y_6 \eqsim Y_7\} \cup \mathscr{E}_7 \qquad \mathscr{E}_7 = \{Y_7 \eqsim \mathsf{p} \to \mathsf{q}{:}\text{HELLO}.Y_8\} \cup \mathscr{E}_8 \qquad \mathscr{E}_8 = \{Y_8 = \mathtt{End}\}$$

- the systems of p-set equations are

$$E = \{x \eqsim y_1\} \cup E_1 \qquad E_1 = \{y_1 \eqsim y_2\} \cup E_2 \qquad E_2 = \{y_2 \eqsim y_3 \cup y_4\} \cup E_3 \cup E_4$$

$$E_3 = \{y_3 \eqsim y_5\} \cup E_5 \qquad E_4 = \{y_4 \eqsim y_6\} \cup E_6 \qquad E_5 = \{y_5 \eqsim x\}$$

$$E_6 = \{y_6 \eqsim y_7 \cup \{\mathsf{u}\}\} \cup E_7 \qquad E_7 = \{y_7 \eqsim y_8\} \cup E_8 \qquad E_8 = \{y_8 \eqsim \emptyset\}$$

- the sets of p-conditions are

$$\mathscr{C} = \{(\mathsf{prt}(Y_1) \cup y_1) \setminus \{\mathsf{p}, \mathsf{q}\} \eqsim \{\mathsf{u}\}\} \cup \mathscr{C}_1 \qquad \mathscr{C}_1 = \{(\mathsf{prt}(Y_2) \cup y_2) \setminus \{\mathsf{p}, \mathsf{u}\} \eqsim \{\mathsf{q}\}\} \cup \mathscr{C}_2$$

$$\mathscr{C}_2 = \{(\mathsf{prt}(Y_3) \cup y_3) \setminus \{\mathsf{u}, \mathsf{p}\} \eqsim \{\mathsf{q}\}, (\mathsf{prt}(Y_4) \cup y_4) \setminus \{\mathsf{u}, \mathsf{p}\} \eqsim \{\mathsf{q}\}\} \cup \mathscr{C}_3 \cup \mathscr{C}_4$$

$$\mathscr{C}_3 = \{(\mathsf{prt}(Y_5) \cup y_5) \setminus \{\mathsf{u}, \mathsf{q}\} \eqsim \{\mathsf{p}\}\} \cup \mathscr{C}_5 \qquad \mathscr{C}_4 = \{(\mathsf{prt}(Y_6) \cup y_6) \setminus \{\mathsf{u}, \mathsf{q}\} \eqsim \{\mathsf{p}\}\} \cup \mathscr{C}_6 \qquad \mathscr{C}_5 = \emptyset$$

$$\mathscr{C}_6 = \{y_6 \eqsim y_7 \cup \{\mathsf{u}\}\} \cup \mathscr{C}_7 \qquad \mathscr{C}_7 = \{(\mathsf{prt}(Y_8) \cup y_8) \setminus \{\mathsf{p}, \mathsf{q}\} \eqsim \emptyset\} \cup \mathscr{C}_8 \qquad \mathscr{C}_8 = \emptyset$$

One can easily verify that a solution of both systems of equations $\mathscr{E}$ and E satisfying the p-conditions (i.e. which agrees with $\mathscr{C}$) is $X = \mathsf{G}$ and $x = \{\mathsf{u}\}$, where G is the global type defined in Example 2.12 and derived for this session in Figure 1. ◇

Let $\mathscr{E}$, E be two systems of type and p-set equations, $\mathscr{C}$ a set of p-conditions and $\mathscr{S}$ a set of goals. A solution $\theta \in \mathsf{sol}(\mathscr{E}, E, \mathscr{C})$ *agrees* with $\mathscr{S}$ if $(X, \mathbb{M}, x) \in \mathscr{S}$ implies $\mathsf{prt}(\theta(X)) \cup \theta(x) = \mathsf{prt}(\mathbb{M})$ for all $X \in \mathsf{vars}(\mathscr{E})$ and all $x \in \mathsf{vars}(E)$. We denote by $\mathsf{sol}_{\mathscr{S}}(\mathscr{E}, E, \mathscr{C})$ the set of all solutions in $\mathsf{sol}(\mathscr{E}, E, \mathscr{C})$ agreeing with $\mathscr{S}$. We say that a system of type equations $\mathscr{E}$ is *guarded* if $X \eqsim Y$ and $Y \eqsim \mathbb{G}$ in $\mathscr{E}$ imply that $\mathbb{G}$ is not a variable. Moreover, $\mathscr{E}$ is *$\mathscr{S}$-closed* if it is guarded and $\mathsf{dom}(\mathscr{E}) \cap \mathsf{vars}(\mathscr{S}) = \emptyset$ and $\mathsf{vars}(\mathscr{E}) \setminus \mathsf{dom}(\mathscr{E}) \subseteq \mathsf{vars}(\mathscr{S})$. We define similarly when a set of p-set equations E is guarded and $\mathscr{S}$-closed.

Toward proving properties of the inference algorithm, we check a couple of auxiliary lemmas. As usual $\mathscr{S} \vdash (X, \mathbb{M}, x) \Rightarrow (\mathscr{E}, E, \mathscr{C})$ means that this judgment belongs to a derivation in the system of Figure 2 having a judgment with an empty sets of goals as conclusion (namely it represents the result of a recursive call during the execution of our algorithm).

$$[\iota\text{-End}]\ \frac{}{\mathscr{N} \vdash^{\iota}_{\emptyset} \mathsf{p}[\mathbf{0}] : \mathtt{End}} \qquad [\iota\text{-Cycle}]\ \frac{}{\mathscr{N}, (\mathsf{G}, \mathbb{M}, \mathscr{P}) \vdash^{\iota}_{\mathscr{P}} \mathbb{M} : \mathsf{G}}$$

$$[\iota\text{-Comm}]\ \frac{\mathscr{N}, (\mathsf{G}, \mathbb{M}, \mathscr{P}) \vdash^{\iota}_{\mathscr{P}_i} \mathsf{p}[P_i] \parallel \mathsf{q}[Q_i] \parallel \mathbb{M}' : \mathsf{G}_i \quad (\mathsf{prt}(\mathsf{G}_i) \cup \mathscr{P}_i) \setminus \{\mathsf{p}, \mathsf{q}\} = \mathsf{prt}(\mathbb{M}') \ \ \forall i \in I}{\mathscr{N} \vdash^{\iota}_{\mathscr{P}} \mathbb{M} : \mathsf{G}} \quad \begin{array}{c} \mathbb{M} \equiv \mathsf{p}[\mathsf{q}!\{\lambda_i.P_i\}_{i\in I}] \parallel \mathsf{q}[\mathsf{p}?\{\lambda_j.Q_j\}_{j\in J}] \parallel \mathbb{M}' \\ \mathsf{G} = \mathsf{p} \to \mathsf{q} : \{\lambda_i.\mathsf{G}_i\}_{i\in I} \quad \mathsf{G} \text{ is bounded} \\ \mathscr{P} = \cup_{i\in I} \mathscr{P}_i \quad I \subseteq J \end{array}$$

$$[\iota\text{-Weak}]\ \frac{\mathscr{N}, (\mathsf{G}, \mathbb{M}_1 \parallel \mathbb{M}_2, \mathscr{P}_1 \cup \mathscr{P}_2) \vdash^{\iota}_{\mathscr{P}_1} \mathbb{M}_1 : \mathsf{G}}{\mathscr{N} \vdash^{\iota}_{\mathscr{P}_1 \cup \mathscr{P}_2} \mathbb{M}_1 \parallel \mathbb{M}_2 : \mathsf{G}} \quad \mathscr{P}_2 = \mathsf{prt}(\mathbb{M}_2) \neq \emptyset$$

Figure 4: Inductive typing rules for sessions.

Lemma 4.2 *If $\mathscr{S} \vdash (X, \mathbb{M}, x) \Rightarrow (\mathscr{E}, E, \mathscr{C})$, then $\mathscr{E}$ and E are $\mathscr{S}$-closed.*

Proof. By a straightforward induction on the derivation of $\mathscr{S} \vdash (X, \mathbb{M}, x) \Rightarrow (\mathscr{E}, E, \mathscr{C})$. □

Lemma 4.3 *If $\mathscr{S} \vdash (X, \mathbb{M}, x) \Rightarrow (\mathscr{E}, E, \mathscr{C})$ and $\theta \in \mathsf{sol}_{\mathscr{S}}(\mathscr{E}, E, \mathscr{C})$, then $\mathsf{prt}(\theta(X)) \cup \theta(x) = \mathsf{prt}(\mathbb{M})$.*

Proof. By induction on the derivation of $\mathscr{S} \vdash (X, \mathbb{M}, x) \Rightarrow (\mathscr{E}, E, \mathscr{C})$. The only interesting case is when Rule [A-Comm] is applied. From $\theta \propto \mathscr{C}$ we get $(\mathsf{prt}(\theta(Y_i)) \cup \theta(y_i)) \setminus \{\mathsf{p}, \mathsf{q}\} = \mathsf{prt}(\mathbb{M})$ for all $i \in I$, which imply $\mathsf{prt}(\theta(X)) \cup \theta(x) = \mathsf{prt}(\mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M})$ since $X = \mathsf{p} \to \mathsf{q} : \{\lambda_i.Y_i\}_{i\in I} \in \mathscr{E}$ and $x = \bigcup_{i\in I} y_i \in E$. □

To show soundness and completeness of our inference algorithm, it is handy to formulate an inductive version of our typing rules, see Figure 4, where $\mathscr{N}$ ranges over sets of triples $(\mathsf{G}, \mathbb{M}, \mathscr{P})$. We can give an inductive formulation since all infinite derivations using the typing rules of Definition 3.1 are regular, i.e. the number of different subtrees of a derivation for a judgement $\mathsf{G} \vdash_{\mathscr{P}} \mathbb{M}$ is finite. In fact, it is bounded by the product of the number of different subterms of G and the number of different subsessions of $\mathbb{M}$, which are both finite as G and (processes in) $\mathbb{M}$ are regular. Applying the standard transformation according to [27, Section 21.9] from a coinductive to an inductive formulation we get the typing rules shown in Figure 4.

In the following two lemmas we relate inference and inductive derivability.

Lemma 4.4 *If $\mathscr{S} \vdash (X, \mathbb{M}, x) \Rightarrow (\mathscr{E}, E, \mathscr{C})$, then $\mathscr{S}\theta \vdash^{\iota}_{\theta(x)} \mathbb{M} : \theta(X)$ for all $\theta \in \mathsf{sol}_{\mathscr{S}}(\mathscr{E}, E, \mathscr{C})$ such that $\mathsf{vars}(\mathscr{S}) \subseteq \mathsf{dom}(\theta)$.*

Proof. By induction on the derivation of $\mathscr{S} \vdash (X, \mathbb{M}, x) \Rightarrow (\mathscr{E}, E, \mathscr{C})$.

Axiom [A-End] We have $\mathscr{E} = \{X = \mathtt{End}\}$, $E = \{x = \emptyset\}$ and $\mathscr{C} = \emptyset$, hence $\theta(X) = \mathtt{End}$, $\theta(x) = \emptyset$ and the thesis follows by Axiom [ι-End].

Axiom [A-Cycle] We have $\mathscr{E} = \{X = Y\}$, $E = \{x = y\}$, $\mathscr{C} = \emptyset$, and $\mathscr{S} = \mathscr{S}', (Y, \mathbb{M}, y)$. Then, $\theta(X) = \theta(Y)$, $\theta(x) = \theta(y)$ and the thesis follows by Axiom [ι-Cycle].

Rule [A-Comm] We have $\mathbb{M} \equiv \mathsf{p}[\mathsf{q}!\{\lambda_i.P_i\}_{i\in I}] \parallel \mathsf{q}[\mathsf{p}?\{\lambda_j.Q_j\}_{j\in J}] \parallel \mathbb{M}'$ with $I \subseteq J$ and $\mathscr{S}, (X, \mathbb{M}, x) \vdash (Y_i, \mathbb{M}_i, y_i) \Rightarrow (\mathscr{E}_i, E_i, \mathscr{C}_i)$ with Y_i, y_i fresh and $\mathbb{M}_i \equiv \mathsf{p}[P_i] \parallel \mathsf{q}[Q_i] \parallel \mathbb{M}'$ and $\mathscr{E} = \{X = \mathsf{p} \to \mathsf{q} : \{\lambda_i.Y_i\}_{i\in I}\} \cup \bigcup_{i\in I} \mathscr{E}_i$, $E = \{x = \bigcup_{i\in I} y_i\} \cup \bigcup_{i\in I} E_i$, $\mathscr{C} = \{(\mathsf{prt}(Y_i) \cup y_i) \setminus \{\mathsf{p}, \mathsf{q}\} \doteq \mathsf{prt}(\mathbb{M}') \mid \forall i \in I\} \cup \bigcup_{i\in I} \mathscr{C}_i$. Since $\mathscr{E}_i \subseteq \mathscr{E}$, $E_i \subseteq E$, and $\mathscr{C}_i \subseteq \mathscr{C}$, we have $\theta \in \mathsf{sol}(\mathscr{E}_i, E_i, \mathscr{C}_i)$ for all $i \in I$. Being $\theta \in \mathsf{sol}_{\mathscr{S}}(\mathscr{E}, E, \mathscr{C})$, Lemma 4.3 implies $\mathsf{prt}(\theta(X)) \cup \theta(x) = \mathsf{prt}(\mathbb{M})$. So we get that θ agrees with $\mathscr{S}, (X, \mathbb{M}, x)$. Then, by the induction hypothesis, we have $\mathscr{S}\theta, (\theta(X), \mathbb{M}, \theta(x)) \vdash^{\iota}_{\theta(y_i)} \mathbb{M}_i : \theta(Y_i)$ for all $i \in I$. The thesis follows by Rule [ι-Comm], since $\theta(X) = \mathsf{p} \to \mathsf{q} : \{\lambda_i.\theta(Y_i)\}_{i\in I}$ and $\theta(x) = \bigcup_{i\in I} \theta(y_i)$.

Rule [A-Weak] We have $\mathbb{M} \equiv \mathbb{M}_1 \parallel \mathbb{M}_2$ and $\mathscr{P} = \mathsf{prt}(\mathbb{M}_2) \neq \emptyset$ and $\mathscr{S}, (X, \mathbb{M}_1 \parallel \mathbb{M}_2, x) \vdash (Y, \mathbb{M}_1, y) \Rightarrow (\mathscr{E}_1, E_1, \mathscr{C})$ and $\mathscr{E} = \{X = Y\} \cup \mathscr{E}_1$ and $E = \{x = y \cup \mathscr{P}\} \cup E_1$. Being $\theta \in \mathsf{sol}_{\mathscr{S}}(\mathscr{E}, E, \mathscr{C})$, Lemma 4.3

implies $\mathsf{prt}(\theta(X)) \cup \theta(x) = \mathsf{prt}(\mathbb{M})$. So we get that θ agrees with $\mathscr{S}, (X, \mathbb{M}, x)$. Then, by the induction hypothesis, we have $\mathscr{S}\theta, (\theta(X), \mathbb{M}, \theta(x)) \vdash^{\iota}_{\theta(y)} \mathbb{M}_1 : \theta(Y)$. The thesis follows by Rule [ι-WEAK]. □

Lemma 4.5 *If $\mathscr{N} \vdash^{\iota}_{\mathscr{P}} \mathbb{M} : \mathsf{G}$ and $\mathsf{prt}(\mathsf{G}') \cup \mathscr{P}' = \mathsf{prt}(\mathbb{M}')$ for all $(\mathsf{G}', \mathbb{M}', \mathscr{P}') \in \mathscr{N}$, then, for all $\mathscr{S}$, X, x and σ such that $X, x \notin \mathsf{vars}(\mathscr{S})$, $\mathsf{dom}(\sigma) = \mathsf{vars}(\mathscr{S})$ and $\mathscr{S}\sigma = \mathscr{N}$, there are $\mathscr{E}$, E, $\mathscr{C}$ and θ such that $\mathscr{S} \vdash (X, \mathbb{M}, x) \Rightarrow (\mathscr{E}, E, \mathscr{C})$ and $\theta \in \mathsf{sol}_{\mathscr{S}}(\mathscr{E}, E, \mathscr{C})$ and $\mathsf{dom}(\theta) = \mathsf{vars}(\mathscr{E}) \cup \mathsf{vars}(E) \cup \mathsf{vars}(\mathscr{S})$ and $\sigma \preceq \theta$ and $\theta(X) = \mathsf{G}$ and $\theta(x) = \mathscr{P}$.*

Proof. By induction on the derivation of $\mathscr{N} \vdash^{\iota}_{\mathscr{P}} \mathbb{M} : \mathsf{G}$. It is easy to verify that $\mathscr{N} \vdash^{\iota}_{\mathscr{P}} \mathbb{M} : \mathsf{G}$ implies $\mathsf{prt}(\mathsf{G}) \cup \mathscr{P} = \mathsf{prt}(\mathbb{M})$.

Axiom [ι-END] The thesis is immediate by Axiom [A-END] taking $\theta = \sigma + \{X \mapsto \mathtt{End}, x \mapsto \emptyset\}$.

Axiom [ι-CYCLE] In this case we have $\mathscr{N} = \mathscr{N}', (\mathsf{G}, \mathbb{M}, \mathscr{P})$, then $\mathscr{S} = \mathscr{S}', (Y, \mathbb{M}, y)$ and $\sigma(Y) = \mathsf{G}$ and $\sigma(y) = \mathscr{P}$. By Axiom [A-CYCLE], we get $\mathscr{S} \vdash (X, \mathbb{M}, x) \Rightarrow (\{X \eqsim Y\}, \{x \eqsim y\}, \emptyset)$, hence $\theta = \sigma + \{X \mapsto \mathsf{G}, x \mapsto \mathscr{P}\}$ is a solution of $\{X \eqsim Y\}$ and of $\{x \eqsim y\}$, which agrees with $\mathscr{S}$, being $\mathsf{prt}(\mathsf{G}) \cup \mathscr{P} = \mathsf{prt}(\mathbb{M})$ as needed.

Rule [ι-COMM] In this case we have $\mathbb{M} \equiv \mathsf{p}[\mathsf{q}!\{\lambda_i.P_i\}_{i\in I}] \parallel \mathsf{q}[\mathsf{p}?\{\lambda_j.Q_j\}_{j\in J}] \parallel \mathbb{M}'$ with $I \subseteq J$ and $\mathsf{G} = \mathsf{p} \to \mathsf{q} : \{\lambda_i.\mathsf{G}_i\}_{i\in I}$ and $\mathscr{N}, (\mathsf{G}, \mathbb{M}, \mathscr{P}) \vdash^{\iota}_{\mathscr{P}_i} \mathbb{M}_i : \mathsf{G}_i$ with $\mathbb{M}_i \equiv \mathsf{p}[P_i] \parallel \mathsf{q}[Q_i] \parallel \mathbb{M}'$ and $(\mathsf{prt}(\mathsf{G}_i) \cup \mathscr{P}_i) \setminus \{\mathsf{p}, \mathsf{q}\} = \mathsf{prt}(\mathbb{M}')$, for all $i \in I$. This last condition implies $\mathsf{prt}(\mathsf{G}) \cup \mathscr{P} = \mathsf{prt}(\mathbb{M})$. Set $\sigma' = \sigma + \{X \mapsto \mathsf{G}, x \mapsto \mathscr{P}\}$ and $\mathscr{S}' = \mathscr{S}, (X, \mathbb{M}, x)$, then, by the induction hypothesis, we get that there are $\mathscr{E}_i, E_i, \mathscr{C}_i$ and θ_i such that $\mathscr{S}' \vdash (Y_i, \mathbb{M}_i, y_i) \Rightarrow (\mathscr{E}_i, E_i, \mathscr{C}_i)$ and $\theta_i \in \mathsf{sol}_{\mathscr{S}'}(\mathscr{E}_i, E_i, \mathscr{C}_i)$ and $\mathsf{dom}(\theta_i) = \mathsf{vars}(\mathscr{E}_i) \cup \mathsf{vars}(E_i) \cup \mathsf{vars}(\mathscr{S}')$ and $\sigma' \preceq \theta_i$ and $\theta_i(Y_i) = \mathsf{G}_i$ and $\theta_i(y_i) = \mathscr{P}_i$, for all $i \in I$. We can assume that $j \neq l$ implies $Y_j \neq Y_l$ and $\mathsf{dom}(\mathscr{E}_j) \cap \mathsf{dom}(\mathscr{E}_l) = \emptyset$ and $y_j \neq y_l$ and $\mathsf{dom}(E_j) \cap \mathsf{dom}(E_l) = \emptyset$ for all $j, l \in I$, because the algorithm always introduces fresh variables. This implies $\mathsf{dom}(\theta_j) \cap \mathsf{dom}(\theta_l) = \{X, x\}$ for all $j \neq l$, and so $\theta = \sum_{i\in I} \theta_i$ is well defined. Moreover, we have $\theta \in \mathsf{sol}_{\mathscr{S}'}(\mathscr{E}_i, E_i, \mathscr{C}_i)$ and $\sigma \preceq \theta$ and $\theta(X) = \mathsf{G}$ and $\theta(x) = \mathscr{P}$, as $\sigma \preceq \sigma'$ and $\sigma' \preceq \theta_i \preceq \theta$ for all $i \in I$. From $(\mathsf{prt}(\mathsf{G}_i) \cup \mathscr{P}_i) \setminus \{\mathsf{p}, \mathsf{q}\} = \mathsf{prt}(\mathbb{M}')$ we get $(\mathsf{prt}(\theta(Y_i)) \cup \theta(y_i)) \setminus \{\mathsf{p}, \mathsf{q}\} = \mathsf{prt}(\mathbb{M}')$ for all $i \in I$. By Rule [A-COMM] we get $\mathscr{S} \vdash (X, \mathbb{M}, x) \Rightarrow (\mathscr{E}, E, \mathscr{C})$ with $\mathscr{E} = \{X \eqsim \mathsf{p} \to \mathsf{q} : \{\lambda_i.Y_i\}_{i\in I}\} \cup \bigcup_{i\in I} \mathscr{E}_i$ and $E = \{x \eqsim \cup_{i\in I} y_i\} \cup \bigcup_{i\in I} E_i$ and $\mathscr{C} = \{(\mathsf{prt}(Y_i) \cup y_i) \setminus \{\mathsf{p}, \mathsf{q}\} \doteq \mathsf{prt}(\mathbb{M}) \mid \forall i \in I\} \cup \bigcup_{i\in I} \mathscr{C}_i$ and $\theta \in \mathsf{sol}_{\mathscr{S}}(\mathscr{E}, E, \mathscr{C})$, since $\theta(X) = \mathsf{G} = \mathsf{p} \to \mathsf{q} : \{\lambda_i.\mathsf{G}_i\}_{i\in I} = \mathsf{p} \to \mathsf{q} : \{\lambda_i.\theta_i(Y_i)\}_{i\in I} = (\mathsf{p} \to \mathsf{q} : \{\lambda_i.Y_i\}_{i\in I})\theta$ and $\theta(x) = \mathscr{P} = \cup_{i\in I} \mathscr{P}_i = \cup_{i\in I} \theta(y_i) = (\cup_{i\in I} y_i)\theta$ and $\sigma \preceq \theta$.

Rule [ι-WEAK] We have $\mathscr{N}, (\mathsf{G}, \mathbb{M}_1 \parallel \mathbb{M}_2, \mathscr{P}_1 \cup \mathscr{P}_2) \vdash^{\iota}_{\mathscr{P}_1} \mathbb{M}_1 : \mathsf{G}$ and $\mathscr{P}_2 = \mathsf{prt}(\mathbb{M}_2) \neq \emptyset$ and $\mathsf{prt}(\mathsf{G}) \cup \mathscr{P} = \mathsf{prt}(\mathbb{M})$, where $\mathbb{M} \equiv \mathbb{M}_1 \parallel \mathbb{M}_2$ and $\mathscr{P} = \mathscr{P}_1 \cup \mathscr{P}_2$. Set $\sigma' = \sigma + \{X \mapsto \mathsf{G}, x \mapsto \mathscr{P}_1 \cup \mathscr{P}_2\}$ and $\mathscr{S}' = \mathscr{S}, (X, \mathbb{M}, x)$, then, by the induction hypothesis, we get that there are $\mathscr{E}_1, E_1, \mathscr{C}_1$ and θ such that $\mathscr{S}' \vdash (Y, \mathbb{M}_1, y) \Rightarrow (\mathscr{E}_1, E_1, \mathscr{C}_1)$ and $\theta \in \mathsf{sol}_{\mathscr{S}'}(\mathscr{E}_1, E_1, \mathscr{C}_1)$ and $\mathsf{dom}(\theta) = \mathsf{vars}(\mathscr{E}_1) \cup \mathsf{vars}(E_1) \cup \mathsf{vars}(\mathscr{S}')$ and $\sigma' \preceq \theta$ and $\theta(Y) = \mathsf{G}$ and $\theta(y) = \mathscr{P}_1$. By Rule [A-WEAK] we get $\mathscr{S} \vdash (X, \mathbb{M}, x) \Rightarrow (\mathscr{E}, E, \mathscr{C}_1)$ with $\mathscr{E} = \{X \eqsim Y\} \cup \mathscr{E}_1$ and $E = \{x \eqsim y \cup \mathscr{P}_2\} \cup E_1$ and $\theta \in \mathsf{sol}_{\mathscr{S}}(\mathscr{E}, E, \mathscr{C}_1)$, since $\theta(X) = \mathsf{G} = \theta(Y)$ and $\theta(x) = \mathscr{P} = \mathscr{P}_1 \cup \mathscr{P}_2 = \theta(y) \cup \mathscr{P}_2 = (y \cup \mathscr{P}_2)\theta$ and $\sigma \preceq \theta$. □

Theorem 4.6 (Soundness and completeness of inference)

1. *If $\vdash (X, \mathbb{M}, x) \Rightarrow (\mathscr{E}, E, \mathscr{C})$, then $\theta(X) \vdash_{\theta(x)} \mathbb{M}$ for all $\theta \in \mathsf{sol}(\mathscr{E}, E, \mathscr{C})$.*
2. *If $\mathsf{G} \vdash_{\mathscr{P}} \mathbb{M}$, then there are $\mathscr{E}$, E, $\mathscr{C}$ and θ such that $\vdash (X, \mathbb{M}, x) \Rightarrow (\mathscr{E}, E, \mathscr{C})$ and $\theta \in \mathsf{sol}(\mathscr{E}, E, \mathscr{C})$ and $\theta(X) = \mathsf{G}$ and $\theta(x) = \mathscr{P}$.*

Proof. (1). By Lemma 4.4 $\vdash (X, \mathbb{M}, x) \Rightarrow (\mathscr{E}, E, \mathscr{C})$ implies $\vdash^{\iota}_{\theta(x)} \mathbb{M} : \theta(X)$ for all $\theta \in \mathsf{sol}(\mathscr{E}, E, \mathscr{C})$. This is enough, since $\vdash^{\iota}_{\theta(x)} \mathbb{M} : \theta(X)$ gives $\theta(X) \vdash_{\theta(x)} \mathbb{M}$.

(2). From $\mathsf{G} \vdash_{\mathscr{P}} \mathbb{M}$ we get $\vdash^{\iota}_{\mathscr{P}} \mathbb{M} : \mathsf{G}$. By Lemma 4.5 this implies that there are $\mathscr{E}$, E, $\mathscr{C}$ and θ such that $\vdash (X, \mathbb{M}, x) \Rightarrow (\mathscr{E}, E, \mathscr{C})$ and $\theta \in \mathsf{sol}(\mathscr{E}, E, \mathscr{C})$ and $\theta(X) = \mathsf{G}$ and $\theta(x) = \mathscr{P}$. □

Remark 4.7 (Termination) As happens for (co)SLD-resolution in logic programming, the termination of the inference algorithm depends on the choice of a resolution strategy. Indeed, we have many sources of non-determinism: we have to select two participants of the session with matching processes and expand them using Rule [A-COMM], or ignore part of the session using Rule [A-WEAK] or try to close a cycle using the Axiom [A-CYCLE]. A standard way to obtain a sound and complete resolution strategy is to build a tree where all such choices are performed in parallel and then visit the tree using a breadth-first strategy. The tree is potentially infinite in depth, but it is finitely branching, since at each point we have only finitely many different choices, hence this strategy necessarily enumerates all solutions.

Remark 4.8 (Use of Rule [A-WEAK]) Note that in a $\vdash^{\iota}_{\mathscr{P}}$ derivation the triple in the premise of Rule [ι-WEAK] can never be used in an application of Axiom [ι-CYCLE]. This – as already hinted at in Example 3.2 – immediately implies that Rule [WEAK] is not strictly necessary inside infinite branches of $\vdash_{\mathscr{P}}$ derivations. Moreover, a slight simplification of the algorithm can be got since, in the step corresponding to Rule [A-WEAK], we could avoid adding the goal $(X, \mathbb{M}_1 \parallel \mathbb{M}_2, x)$ to the current set of goals. This would reduce the number of goals to be checked during the step corresponding to Axiom [ι-CYCLE]. Rule [A-WEAK] turns out to be necessary, instead, when applying the algorithm to sessions where non-ignored participants expose a finite behaviour, like $\mathsf{p}[P_2] \parallel \mathsf{q}[Q_2] \parallel \mathsf{u}[U]$ in Example 3.2. Also the typing of stuck sessions with recursive processes like $\mathsf{p}[P] \parallel \mathsf{q}[Q]$ where $P = \mathsf{q}!\lambda.P$ and $Q = \mathsf{p}!\lambda.Q$ requires the use of Rule [A-WEAK]. ◇

5 Concluding Remarks, Related and Future Works

Lock-freedom is definitely a relevant and widely investigated communication property of concurrent and distributed systems. It ensures absence of *locks*, a lock being a reachable configuration where a communication action of a participant remains pending in any possible continuation of the system. In case the participant prevented to progress be p, such configuration is called a p-*lock* (see [5] for an abstract definition of Lock-freedom). Lock-freedom corresponds to the notion of liveness in [20, 22] where the synchronous communication is channel-based. Sometimes properties different from what we intend are referred to by "Lock-freedom": for instance the notion of Lock-freedom in [19], under fair scheduling, corresponds to what [28] and [5] refer to as *strong Lock-freedom*.

Various formalisms and methodologies have been developed in order to prove Lock-freedom while others do ensure Lock-freedom by construction. Among the former there are type assignment systems where typability entails Lock-freedom, both for asynchronous [26] and synchronous [3] communications.

Lock-freedom is quite a strong property: it entails Deadlock-freedom, whereas the vice versa does not hold. In several actual scenarios, lighter forms of Lock-freedom would however suffice. For instance in clients/servers scenarios where one can accept some servers to get locked after their interactions with the clients have been completed.

In the present paper we developed a type assignment system where typability ensures $\mathscr{P}$-*excluded Lock-freedom*: the absence of p-locks for each participant p not belonging to $\mathscr{P}$. This is achieved by means of "partial" typability, i.e. by disregarding typability of (sub)processes of participants that we can safely assume to get possibly locked. Multiparty sessions (parallel compositions of named processes) are (partially) typed by *global types*, which in turn describe the overall interactions inside the multiparty sessions. Our partial typability ensures also that the behaviours of the non-ignored participants adhere to what the global type describes. As far as we know, there are not other formalisms dealing with properties like $\mathscr{P}$-excluded Lock-freedom.

Our partial typing is reminiscent of connecting communications, a notion introduced in [18] and further investigated in [8, 10] in order to describe protocols with optional participants. The intuition behind connecting communications is that in some parts of the protocol, delimited by a choice construct, some participants may be optional, namely they are "invited" to join the interaction only in some branches of the choice, by means of connecting communications. As argued in [18, 8, 10], this feature allows for a more natural description of typical communication protocols. In [10], connecting communications also enable to express conditional delegation: this will be obtained by writing a choice where the delegation appears only in some branches of the choice, following a connecting communication. The participants offering connecting communications should be ignored in the present type system. An advantage of connecting communications over partial typing is that only participants offering connecting inputs can be stuck. The disadvantage is that the typing rules are more requiring, so many interesting sessions can be partially typed but cannot be typed by means of connecting communications.

In designing type inference we took inspiration from [13], where inputs and outputs are split in global types in order to better describe asynchronous communication. Our inference algorithm is related as goal, but very different as methodology, to the algorithm in [23], which builds global graphs from sets of communicating finite state machines satisfying suitable conditions. We are planning to implement our type inference algorithm.

Unlike many MPST formalisms in the literature, like [17], we type sessions with global types without recurring to local types and projections. It would be interesting to investigate the possibility of extending the standard projection operator to a relation between global types and possibly non lock-free local behaviours. Other simplifications of our calculus are the absence of values in messages and the unicity of channels. While we can easily enrich messages with values, allowing more than one channel requires sophisticated type systems in order to get Lock-freedom [26].

The following example shows a further direction for investigation of partial typing, namely to describe and analyse privacy matters.

Example [**Partial typing for privacy**] The communications written in global types can be viewed as public, while the others can be viewed as private. For example Alice and Bob want to discuss privately which version of a game would be the most suitable for their son Carl, who asked for it as birthday present. Taking participants a, b and c to incarnate, respectively, Alice, Bob and Carl this scenario can be represented by the session

$$\mathbb{M} \equiv \mathsf{a}[\mathsf{c}?\textsc{present}.P] \parallel \mathsf{b}[\mathsf{c}?\textsc{present}.Q] \parallel \mathsf{c}[\mathsf{a}!\textsc{present}.\mathsf{b}!\textsc{present}]$$

where $P = \mathsf{b}!\{\textsc{bla}.\mathsf{b}?\textsc{bla'}.P, \textsc{ok}\}$ and $Q = \mathsf{a}?\{\textsc{bla}.\mathsf{a}!\textsc{bla'}.P, \textsc{ok}\}$.
A suitable global type is $\mathsf{G} = \mathsf{c} \to \mathsf{a}{:}\textsc{present}.\mathsf{c} \to \mathsf{b}{:}\textsc{present}$. We can in fact derive $\mathsf{G} \vdash_{\{\mathsf{a},\mathsf{b}\}} \mathbb{M}$. ◇

We also plan to investigate partial typing for asynchronous communications, possibly modifying the type system of [13]. An advantage of that type system is the possibility of anticipating outputs over inputs without requiring the asynchronous subtyping of [25], which is known to be undecidable [6, 24]. A difficulty will come from the larger freedom in choosing the order of interactions due to the splitting between writing and reading messages on a queue.

Acknowledgments We wish to gratefully thank the anonymous reviewers for their thoughtful and helpful comments.

References

[1] Jirí Adámek, Stefan Milius & Jiri Velebil (2006): *Iterative algebras at work*. *Mathematical Structures in Computer Science* 16(6), pp. 1085–1131, doi:10.1017/S0960129506005706.

[2] Davide Ancona & Agostino Dovier (2015): *A theoretical perspective of coinductive logic programming*. *Fundamenta Informaticae* 140(3-4), pp. 221–246, doi:10.3233/FI-2015-1252.

[3] Franco Barbanera, Mariangiola Dezani-Ciancaglini & Ugo de'Liguoro (2022): *Open compliance in multiparty sessions*. In S. Lizeth Tapia Tarifa & José Proença, editors: *FACS*, *LNCS* 13712, Springer, pp. 222–243, doi:10.1007/978-3-031-20872-0_13.

[4] Franco Barbanera, Mariangiola Dezani-Ciancaglini, Ivan Lanese & Emilio Tuosto (2021): *Composition and decomposition of multiparty sessions*. *Journal of Logic and Algebraic Methods in Programming* 119, p. 100620, doi:10.1016/j.jlamp.2020.100620.

[5] Franco Barbanera, Ivan Lanese & Emilio Tuosto (2022): *Formal choreographic languages*. In Maurice H. ter Beek & Marjan Sirjani, editors: *COORDINATION*, *LNCS* 13271, Springer, pp. 121–139, doi:10.1007/978-3-031-08143-9_8.

[6] Mario Bravetti, Marco Carbone & Gianluigi Zavattaro (2017): *Undecidability of asynchronous session subtyping*. *Information and Computation* 256, pp. 300–320, doi:10.1016/j.ic.2017.07.010.

[7] Giuseppe Castagna, Nils Gesbert & Luca Padovani (2009): *A theory of contracts for Web services*. *ACM Transaction on Programming Languages and Systems* 31(5), pp. 19:1–19:61, doi:10.1145/1538917.1538920.

[8] Ilaria Castellani, Mariangiola Dezani-Ciancaglini & Paola Giannini (2019): *Reversible sessions with flexible choices*. *Acta Informatica* 56(7), pp. 553–583, doi:10.1007/s00236-019-00332-y.

[9] Ilaria Castellani, Mariangiola Dezani-Ciancaglini & Paola Giannini (2022): *Asynchronous sessions with input races*. In Marco Carbone & Rumyana Neykova, editors: *PLACES*, *EPTCS* 356, Open Publishing Association, pp. 12–23, doi:10.4204/EPTCS.356.2.

[10] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, Paola Giannini & Ross Horne (2020): *Global types with internal delegation*. *Theoretical Computer Science* 807, pp. 128–153, doi:10.1016/j.tcs.2019.09.027.

[11] Luca Ciccone, Francesco Dagnino & Luca Padovani (2022): *Fair termination of multiparty sessions*. In Karim Ali & Jan Vitek, editors: *ECOOP*, *LIPIcs* 222, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 26:1–26:26, doi:10.4230/LIPIcs.ECOOP.2022.26.

[12] Bruno Courcelle (1983): *Fundamental properties of infinite trees*. *Theoretical Computer Science* 25, pp. 95–169, doi:10.1016/0304-3975(83)90059-2.

[13] Francesco Dagnino, Paola Giannini & Mariangiola Dezani-Ciancaglini (2023): *Deconfined global types for asynchronous sessions*. *Logical Methods in Computer Science* 19(1), pp. 1–41, doi:10.46298/lmcs-19(1:3)2023.

[14] Romain Demangeon & Kohei Honda (2012): *Nested protocols in session types*. In Maciej Koutny & Irek Ulidowski, editors: *CONCUR*, *LNCS* 7454, Springer, pp. 272–286, doi:10.1007/978-3-642-32940-1_20.

[15] Rob van Glabbeek, Peter Höfner & Ross Horne (2021): *Assuming just enough fairness to make session types complete for lock-freedom*. In Leonid Libkin, editor: *LICS*, ACM Press, pp. 1–13, doi:10.1109/LICS52264.2021.9470531.

[16] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In George C. Necula & Philip Wadler, editors: *POPL*, ACM Press, pp. 273–284, doi:10.1145/1328897.1328472.

[17] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty asynchronous session types*. *Journal of the ACM* 63(1), pp. 9:1–9:67, doi:10.1145/2827695.

[18] Raymond Hu & Nobuko Yoshida (2017): *Explicit connection actions in multiparty session types*. In: *FASE*, *LNCS* 10202, Springer, pp. 116–133, doi:10.1007/978-3-662-54494-5.

[19] Naoki Kobayashi (2002): *A type system for lock-free processes*. *Information and Computation* 177(2), pp. 122–159, doi:10.1006/inco.2002.3171.

[20] Naoki Kobayashi & Davide Sangiorgi (2010): *A hybrid type system for lock-freedom of mobile processes*. *ACM Transactions on Programming Languages and Systems* 32(5), pp. 16:1–16:49, doi:10.1145/1745312.1745313.

[21] Dexter Kozen & Alexandra Silva (2017): *Practical Coinduction*. *Mathematical Structures in Computer Science* 27(7), pp. 1132–1152, doi:10.1017/S0960129515000493.

[22] Julien Lange, Nicholas Ng, Bernardo Toninho & Nobuko Yoshida (2017): *Fencing off Go: liveness and safety for channel-based programming*. In Giuseppe Castagna & Andrew D. Gordon, editors: *POPL*, ACM Press, pp. 748–761, doi:10.1145/3009837.3009847.

[23] Julien Lange, Emilio Tuosto & Nobuko Yoshida (2015): *From communicating machines to graphical choreographies*. In Sriram K. Rajamani & David Walker, editors: *POPL*, ACM Press, pp. 221–232, doi:10.1145/2676726.2676964.

[24] Julien Lange & Nobuko Yoshida (2017): *On the undecidability of asynchronous session subtyping*. In Javier Esparza & Andrzej S. Murawski, editors: *FOSSACS*, *LNCS* 10203, Springer, pp. 441–457, doi:10.1007/978-3-662-54458-7_26.

[25] Dimitris Mostrous, Nobuko Yoshida & Kohei Honda (2009): *Global principal typing in partially commutative asynchronous sessions*. In Giuseppe Castagna, editor: *ESOP*, *LNCS* 5502, Springer, pp. 316–332, doi:10.1007/978-3-642-00590-9_23.

[26] Luca Padovani (2014): *Deadlock and lock freedom in the linear π-calculus*. In Thomas A. Henzinger & Dale Miller, editors: *CSL-LICS*, ACM Press, pp. 72:1–72:10, doi:10.1007/978-3-662-43376-8_10.

[27] Benjamin C. Pierce (2002): *Types and Programming Languages*. MIT Press.

[28] Paula Severi & Mariangiola Dezani-Ciancaglini (2019): *Observational equivalence for multiparty sessions*. *Fundamenta Informaticae* 167, pp. 267–305, doi:10.3233/FI-2019-1863.

[29] Luke Simon (2006): *Extending logic programming with coinduction*. Ph.D. thesis, University of Texas at Dallas.

[30] Luke Simon, Ajay Bansal, Ajay Mallya & Gopal Gupta (2007): *Co-logic programming: extending logic programming with coinduction*. In Lars Arge, Christian Cachin, Tomasz Jurdzinski & Andrzej Tarlecki, editors: *ICALP*, *LNCS* 4596, Springer, pp. 472–483, doi:10.1007/11799573_25.

[31] Luke Simon, Ajay Mallya, Ajay Bansal & Gopal Gupta (2006): *Coinductive logic programming*. In Sandro Etalle & Miroslaw Truszczynski, editors: *ICLP*, *LNCS* 4079, Springer, pp. 330–345, doi:10.1007/11799573_25.

Algebraic Reasoning About Timeliness

Seyed Hossein HAERI
IOG, Belgium
University of Bergen, Norway
hossein.haeri@iohk.io

Peter W. THOMPSON
PNSol, UK
Peter.Thompson@pnsol.com

Peter VAN ROY
Université catholique de Louvain, Belgium
pvr@info.ucl.ac.be

Magne HAVERAAEN
University of Bergen, Norway
Magne.Haveraaen@uib.no

Neil J. DAVIES
PNSol, UK
Neil.Davies@pnsol.com

Mikhail BARASH
University of Bergen, Norway
mikhail.barash@uib.no

Kevin HAMMOND
IOG, UK
kevin.hammond@iohk.io

James CHAPMAN
IOG, UK
james.chapman@iohk.io

Designing distributed systems to have predictable performance under high load is difficult because of resource exhaustion, non-linearity, and stochastic behaviour. Timeliness, i.e., delivering results within the required time bounds, makes a major contribution to the predictability of performance. In this paper, we focus on timeliness using the ΔQ Systems Development paradigm (ΔQSD, developed by PNSol), which computes timeliness by modelling systems observationally using so-called outcome expressions. An outcome expression is a compositional definition of a system's observed behaviour in terms of its basic operations. Given the stochastic behaviour of the basic operations, ΔQSD efficiently computes the stochastic behaviour of the whole system including its timeliness.

This paper formally proves useful algebraic properties of outcome expressions w.r.t. timeliness: We prove the different algebraic structures the set of outcome expressions form with the different ΔQSD operators and demonstrate why those operators do not form richer structures. We prove or disprove the set of all possible distributivity results on outcome expressions. We prove 14 equivalences that have been used in the past in the practice of ΔQSD.

An immediate benefit is rewrite rules that can be used for design exploration under established timeliness equivalence. This work is part of an ongoing project to disseminate and build tool support for ΔQSD. The ability to rewrite outcome expressions is essential for efficient tool support.

1 Introduction

Designing distributed systems to have predictable performance under high load is difficult. At high load, resources such as network, memory, storage, or CPU capacity will be exhausted, causing a dramatic effect on performance. Prediction is difficult because the behaviour of system components and their interactions are both nonlinear and stochastic. For over 20 years, a small group of people associated with the company PNSol has worked on diagnosing and designing systems to predict and correct performance problems [17]. PNSol has developed the ΔQ Systems Development paradigm (ΔQSD) as part of this work. ΔQSD has been used in areas as diverse as telecommunications [20] [19] [6], WiFi [14], and distributed ledgers [5]. ΔQSD has been applied to many large industrial systems, including BT, Vodafone, Boeing Space and Defence, and IOG (formerly IOHK).

This paper defines and proves algebraic properties of the ΔQSD operators w.r.t. timeliness, i.e., delivering outcomes within the acceptable time-frames. That is, in this paper, our solo resource of concern is time, although ΔQSD includes other types of resources and their interaction.

Submitted to:
ICE 2023

This theoretical work is part of an ongoing project to disseminate and build tool support for ΔQSD, to make it available to the wide community of system engineers. We base our work on the ΔQSD formalisation given by Haeri et al. [11], which defines outcome expressions and their semantics, and gives a real-world example of ΔQSD taken from the blockchain domain.

Contributions

This paper gives a firm mathematical foundation for ΔQSD, and uses this to establish important algebraic properties of the ΔQSD operators with respect to timeliness, i.e., when the relevant resource is time. This paper is based on a general model theory of resource analysis for systems specified using outcome expressions [12]. That model theory is the first of its kind and we specialise it using the timeliness analysis recipe that is commonly used in ΔQSD (Definition 3).

- We show that the set of outcome expressions forms different algebraic structures with the different ΔQSD operators (Theorems 1–4).
- We establish 3 distributivity results in Section 7 about the ΔQSD operators (Theorem 6).
- We rule out the formation of certain richer algebraic structures by the set of outcome expressions and the current ΔQSD operators (Remarks 2, 3, and 4).
- We develop two new techniques for analysing the validity of algebraic equivalences: a new technique that we call *Properisation* (Section 7.2) and another based on counterexamples (Section 7.3). We use those techniques to refute the remaining possible distributivity results in their full generality: 8 using properisation (Theorems 8) and 4 using counterexamples (Theorem 9).
- We provide guidelines for studying the necessary/sufficient conditions for the distributivity results we refute the generality of (Section 7.1).
- We establish 14 equivalences that have been used in the past in the practice of ΔQSD (Section 6).

Full proofs can be found in the accompanying technical report [12], which also shows how Fig. 2 can be further elaborated using our Jupyter notebook. We will integrate some proofs in our post-proceedings version of this work.

The primary practical results of this paper are to establish distributive properties of ΔQSD operators and other equivalences that are useful for rewriting outcome expressions. These enable common subexpressions to be moved, for example, to reduce representational complexity, with associated gains in tool performance. Rewriting can also be used to produce normal forms, and, in particular, to extract reliability/failure probabilities without fully evaluating the outcome expression. More generally, it can be used to establish equivalences between different designs with respect to their timeliness, even though their usage of other resources might differ, by which allowing design exploration under equivalence.

2 Motivating Example: Cache Memory

We give an example of a memory system consisting of a local cache with a networked main memory. This example serves two purposes: first, it shows how outcome diagrams can be used to model nontrivial systems; and, second, it shows the usefulness of the algebraic transformations of this paper. We give the block diagram, the outcome diagram, and the outcome expression for this example. We then compute the quality attenuation and (delay distribution and failure rate) from this outcome expression. To that end, we use the algebraic transformations proved in this paper. Fig. 1 gives the block diagram of the

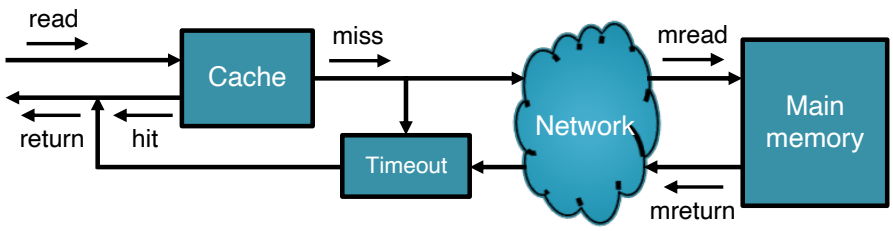

Figure 1: Block Diagram for a Cache with Networked Main Memory

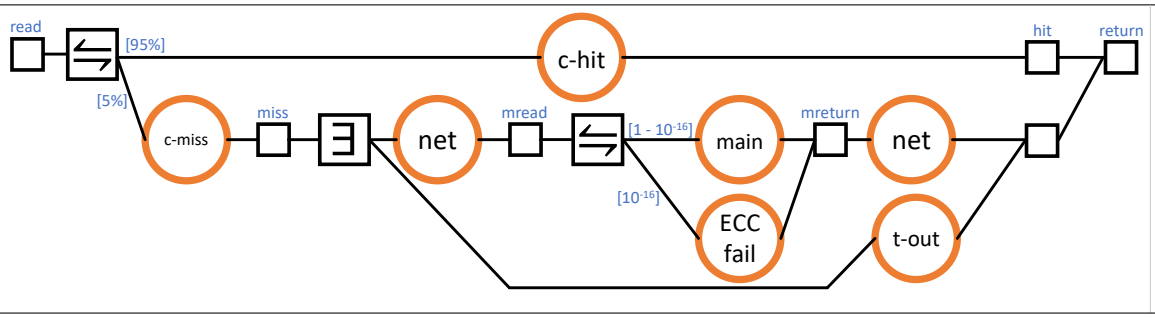

Figure 2: Outcome Diagram for the Cache of Figure 1

memory system. A read message enters the cache; a cache hit – when the memory word is in the cache – results in an immediate return message; a cache miss – when the memory word is not in the cache – results in a main memory read. The main memory is across the network, so accessing it requires a network communication in both directions. Main memory access is guarded by a timeout in case of network problems. The cache miss initialises the timeout timer; the mreturn message is passed through if it occurs before the timeout; otherwise, a timeout message is passed instead. Furthermore, there is a small probability that the remote main memory read fails.

Outcome Diagram for the Cache with Networked Memory Fig. 2 shows the outcome diagram for the memory system. The outcome diagram is a graphical representation of an outcome expression.[1] We can define an outcome as what the system obtains by performing one of its tasks. Outcomes are shown using orange circles in the diagrams. When there is a left-to-right path from one outcome to another, the right one is causally dependent on the left one. Small square boxes show the starting and terminating events of the corresponding outcomes. Large square boxes are operators. In Fig. 2 there are two *probabilistic choices*, "$\leftrightharpoons$", and one *first-to-finish* synchronisation, "$\exists$". We assume that the cache hit rate is 95%. That is modelled using the leftmost probabilistic choice with two leads – one to each outcome ("cache hit" and "cache miss"), decorated with their corresponding probabilities. Timeout is modelled by a first-to-finish relationship between the main memory read and the timer. We assume that the main memory uses Error-Correction Codes (ECC) to catch bit errors. We account for the possibility that a main memory access fails (e.g., because of hardware failure) by giving it a failure rate of 10^{-16}. This assumption is modelled in Fig. 2 as a probabilistic choice between the "main" and "ECC fail"

[1] In this paper, we take the equivalence between the outcome expressions and outcome diagrams for granted. That equivalence is not the focus of this paper.

outcomes. The outcome expression for the diagram of Fig. 2 is

$$\textit{c-hit} \underset{\leftharpoondown}{\overset{[95\%]}{\rightharpoonup}} (\textit{c-miss} \bullet\!\!\rightarrow\!\!\bullet ((\textit{net} \bullet\!\!\rightarrow\!\!\bullet (\textit{main} \underset{\leftharpoondown}{\overset{[1-10^{-16}]}{\rightharpoonup}} \bot) \bullet\!\!\rightarrow\!\!\bullet \textit{net}) \parallel^{\exists} \textit{t-out})) \tag{1}$$

where "$\bot$" and "$\bullet\!\!\rightarrow\!\!\bullet$", in ΔQSD, represent *(unconditional) failure* and *sequential composition*, respectively. Note that the operator "$\exists$" in the outcome diagram is "$\parallel^{\exists}$" in the outcome expression. That is to signify that when two outcomes are connected by first-to-finish, they are performed concurrently; hence the "$\parallel$" sign.

We can now ask what the quality attenuation of the memory system is (where quality attenuation is a measure for delay (and failure), represented using a Cumulative Distribution Function (CDF); see Section 3 for deatils. We can compute this using the semantics, as defined in [11]. From this we can also determine the failure rate as the asymptote of the quality attenuation as delay increases to infinity. However, there is an easier way to determine the failure rate, by using algebraic transformations on the underlying outcome expressions. The techniques used for this example generalise in a straightforward fashion to any system modelled using an outcome expression.

Quality Attenuation (Delay Part of the CDF) Using the semantics defined in Section 4, one can compute the overall quality attenuation from Equation (1), given the quality attenuations of the five outcomes that are taken primitive here: *c-hit*, *c-miss*, *t-out*, *net*, and *main*. These five cumulative distribution functions, $\Delta\mathrm{Q}_{c-hit}$, $\Delta\mathrm{Q}_{c-miss}$, $\Delta\mathrm{Q}_{t-out}$, $\Delta\mathrm{Q}_{net}$, and $\Delta\mathrm{Q}_{main}$, are known initially: the first two are properties of the cache, the timeout is chosen by the designer, the network performance is known, and the main memory read time is known. The computation is done from Equation (1) using the semantics that will be given by Definition 3. Take $\textit{mem} = \textit{net} \bullet\!\!\rightarrow\!\!\bullet (\textit{main} \underset{\leftharpoondown}{\overset{[1-10^{-16}]}{\rightharpoonup}} \bot) \bullet\!\!\rightarrow\!\!\bullet \textit{net}$ to be the outcome of the networked main memory read. We start by computing $\Delta\mathrm{Q}_{mem}$:

$$\Delta\mathrm{Q}_{mem} = \Delta\mathrm{Q}_{net} * ((1-10^{-16}) \cdot \Delta\mathrm{Q}_{main} + 10^{-16} \cdot \Delta\mathrm{Q}_{\bot}) * \Delta\mathrm{Q}_{net} \tag{2}$$

Note that the $*$ operation is a convolution. The $\cdot$ and $+$ operations are arithmetic multiplication and addition of CDFs. Since $\bot$ is a failure, we know that $\Delta\mathrm{Q}_{\bot} = \mathbf{0}$, so we simplify:

$$\Delta\mathrm{Q}_{mem} = \Delta\mathrm{Q}_{net} * (1-10^{-16}) \cdot \Delta\mathrm{Q}_{main} * \Delta\mathrm{Q}_{net} \tag{3}$$

The overall ΔQ is then given by:

$$\Delta\mathrm{Q} = 0.95 \cdot \Delta\mathrm{Q}_{c\text{-}hit} + 0.05 \cdot (\Delta\mathrm{Q}_{c\text{-}miss} * (\Delta\mathrm{Q}_{mem} + \Delta\mathrm{Q}_{t\text{-}out} - \Delta\mathrm{Q}_{mem} \cdot \Delta\mathrm{Q}_{t\text{-}out})) \tag{4}$$

This computation gives us the CDF for the execution time of a memory read. The numeric computation is easily performed by a software tool. For readers interested in seeing fully worked-out numerical examples, we recommend looking up the tutorial [22].

Failure Rate (Failure Part of the CDF) Let us now compute the failure rate by doing algebraic transformations as defined in this paper. Without loss of generality, we can assume that the network has zero delay and the timeout is infinite. One can then replace Figure 2 with Figure 3. Likewise, Equation (1) simplifies to:

$$\textit{c-hit} \underset{\leftharpoondown}{\overset{[95\%]}{\rightharpoonup}} (\textit{c-miss} \bullet\!\!\rightarrow\!\!\bullet (\textit{main} \underset{\leftharpoondown}{\overset{[1-10^{-16}]}{\rightharpoonup}} \bot)) \tag{5}$$

According to Theorem 5, expression (5) is equivalent to $\textit{c-hit} \underset{\leftharpoondown}{\overset{[95\%]}{\rightharpoonup}} ((\textit{c-miss} \bullet\!\!\rightarrow\!\!\bullet \textit{main}) \underset{\leftharpoondown}{\overset{[1-10^{-16}]}{\rightharpoonup}} \bot)$, which can be rewritten using Lemma 2 as

$$(\textit{c-hit} \underset{\leftharpoondown}{\overset{[p]}{\rightharpoonup}} (\textit{c-miss} \bullet\!\!\rightarrow\!\!\bullet \textit{main})) \underset{\leftharpoondown}{\overset{[q]}{\rightharpoonup}} \bot \tag{6}$$

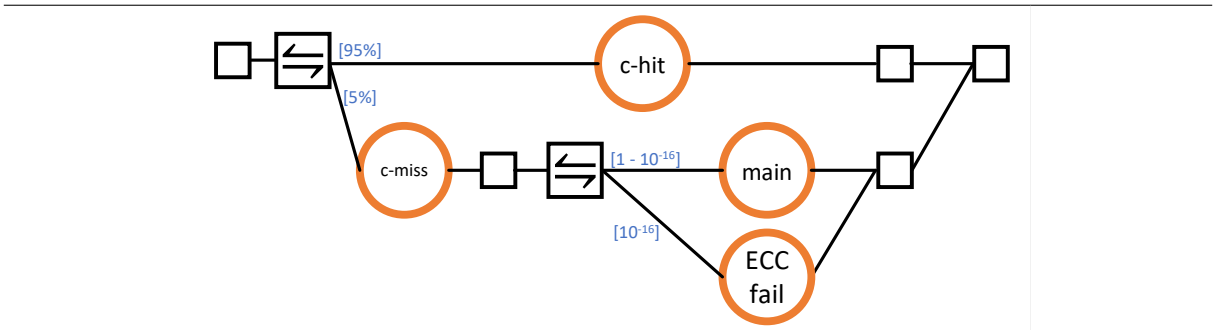

Figure 3: Outcome Diagram for the Cache Example, Disregarding Network Delay and Timeout

for some p, and for $q = (1 - 0.05 \cdot 10^{-16}) = 0.999999999999999995$. Expression (6) is a swiftly obtained, and immediately tells the system engineer that, under the current assumptions about cache hit and main memory failure rates, **every** design will be infeasible if the overall failure rate must be less than q.

Final Remarks on the Example Realistic cache memories are often more complex than this example, which gives rise to more complicated outcome expressions in which "$\bot$" will appear at multiple depths. Thanks to Theorem 6 as well as Lemmas 2 and 3, techniques such as that of this section can be used to accumulate those $\bot$s for similar infeasibility tests. It is important to notice that we can compute the delay (part of the CDF) independently of the failure rate. That is because, due to the properisation theorem (Theorem 7), we can replace $\mathit{c\text{-}hit} \overset{[95\%]}{\rightleftharpoons} (\mathit{c\text{-}miss} \bullet\!\!\rightarrow\!\!\bullet\, \mathit{main})$ by expression (6), provided that the rate of failure of *main* is 10^{-16}. In practice, the properisation theorem is a very handy result. Section 7.2 gives more details about the theoretical benefits of properisation.

While the probabilities in this example may seem small, they can combine with probabilities from other parts of the system, and it is important to be able to keep track of them. Dismissing them as 'minimal' risks missing potentially serious failures when many 'small' probabilities aggregate.

3 Background

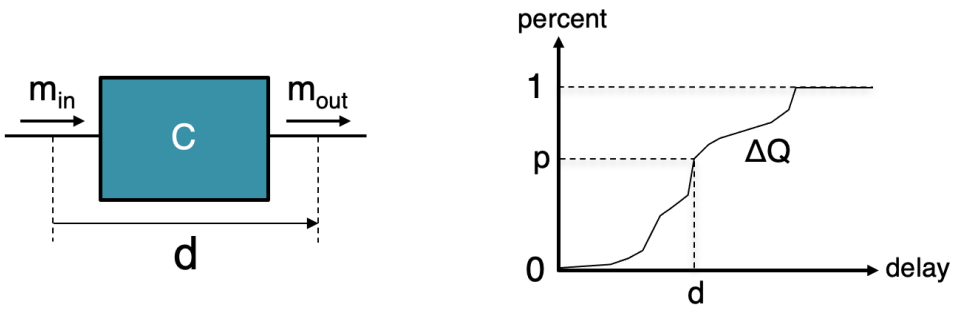

Figure 4: A Component's Operation and its Cumulative Delay Function

Outcome and Quality Attenuation Consider a component C which inputs message m_{in} and outputs message m_{out} after a delay d. Doing this many times will usually give different delays. We define a

cumulative delay function so that p percent of delays are less or equal to d. Figure 4 gives an illustration.

The ΔQSD paradigm generalises this simple measurement. We measure delay not only for messages, but for all system behaviours that have a starting event and a terminating event. Given a starting event e_{in} and a terminating event e_{out}, what the system gains within the (e_{in}, e_{out}) time frame is called an instance of an *outcome*. We also generalise the property that we measure: we measure not only delay, but any property that makes the system less than perfect. The cumulative distribution function of the property is then called a *quality attenuation*. In what follows, we will consistently use the terms outcome and quality attenuation.

Failure It is straightforward to generalise the quality attenuation to model both delay and failure. It suffices to allow the cumulative delay function's limit to be less than 1. Figure 5 illustrates this possibility. There is an f percent probability that the delay is infinite, which corresponds precisely to a failure. For the component, it means simply that there is an input message m_{in} with no corresponding output message m_{out}. Mathematically, the delay is modelled by a random variable that is allowed to be **improper**: The probability that it is infinite can be greater than 0. This probability is called the intangible mass of the Improper Random Variable (IRV) [21].

The ability to model delay and failure as a single quantity is a strength of ΔQSD. It makes it easy to explore trade-offs between delay and failure in the system design. This ability shows up clearly in the algebra presented in this paper.

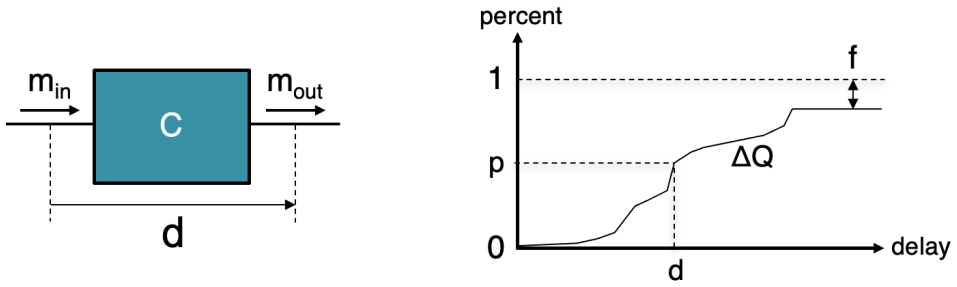

Figure 5: Failure is modelled as a quality attenuation whose limit is less than 1

Timeliness We define *timeliness* as a relation between an observed ΔQ_{obs} and a required ΔQ_{req}. We say that the system *satisfies timeliness* for a given outcome if $\Delta Q_{obs} \leq \Delta Q_{req}$. Figure 6 illustrates this condition.

Outcome Expressions For a system consisting of multiple interconnected components, one can define a graph that combines all the components' outcomes. This graph defines the causal relationships between the outcomes and is called an *outcome diagram*. For example, Figure 2 shows the outcome diagram of a cache whose block diagram is given in Figure 1. Each outcome diagram has a corresponding *outcome expression* – a mathematical description of the diagram. Given an outcome expression and given the quality attenuations of all its components, it is possible to compute the quality attenuation of the complete system as a whole. The reverse process is also possible: Given an outcome expression and given the required quality attenuation of the complete system, it is possible to compute the required quality attenuations of its components. This gives the system designer a powerful tool for both design and diagnosis.

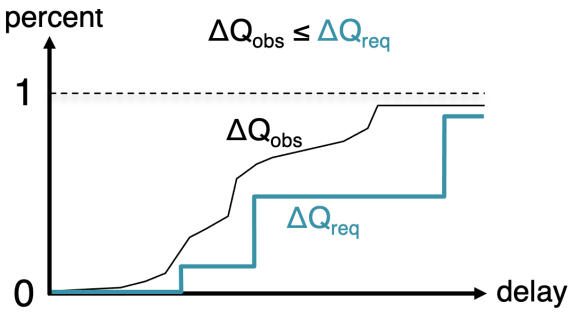

Figure 6: Timeliness: the observed quality attenuation ΔQ is always to the left and above the required ΔQ

Outcome expressions can be manipulated according to algebraic rules. An important set of algebraic rules is presented in this paper. These rules give additional abilities for system designers who use ΔQSD. As part of an ongoing project, we are building software tools to support ΔQSD. The algebraic rules presented here are essential for making practical the symbolic manipulation of outcome expressions needed by these tools.

ΔQSD

ΔQSD is a system development paradigm that is able to compute many system properties early on in the design process, such as performance (latency and throughput), timeliness, risks, and feasibility. ΔQSD is used both for diagnosis and design:

- *System Diagnosis*. ΔQSD can analyse an existing system, to pinpoint anomalous behaviours so their origin can be found and the system can be corrected.
- *System Design*. ΔQSD can estimate performance trade-offs during the design process. At **every** step of the design process, performance of the complete system can be estimated by a computation on the partial design. This computation also determines whether or not the system is feasible, i.e., whether it can or cannot meet the requirements.

While historically ΔQSD has primarily been used to diagnose and correct problems in large industrial systems, PNSol has recently used ΔQSD to design the Shelley block diffusion algorithm as used in the Cardano blockchain [11]. More information on ΔQSD can be found in a tutorial given at HiPEAC 2023 tutorial [22].

4 An Algebraic Perspective on Timeliness

4.1 Syntax of Outcome Expressions

Definition 1 (Haeri et al. [11])**.** Assume a set $\overline{\mathbb{B}}$ of primitive outcomes. We use variables $\beta \in \overline{\mathbb{B}}$ to represent individual primitive outcomes. We define the abstract syntax of outcome expressions as follows:

$$
\begin{array}{lllll}
\mathbb{O} \ni o ::= & \beta & \text{primitive outcome} & & \\
\quad | & o \bullet\!\!\rightarrow\!\!\bullet\, o' & \text{sequential composition} & | \ (o \parallel^{\forall} o') & \text{all-to-finish (a.k.a. last-to-finish)} \\
\quad | & o \overset{m}{\underset{m'}{\leftrightharpoons}} o' & \text{probabilistic choice} & | \ (o \parallel^{\exists} o') & \text{any-to-finish (a.k.a. first-to-finish).}
\end{array}
$$

This defines outcome expressions as combinations of primitive outcomes β and four composition operators. In the case of probabilistic choice, m and m' are numeric weights which give the probabilities of choosing the left or right alternative, respectively. For convenience, we also introduce another notation $o \underset{\longrightarrow}{\overset{[p]}{}} o'$ where the probability $(1-p)$ for the right alternative is implied. We distinguish two constant outcomes: $\top$ for "perfection" and $\bot$ for "unconditional failure."

4.2 Timeliness Semantics for Outcome Expressions

Let $\Delta\mathrm{Q}(x)$ denote the probability that an outcome occurs in a time $t \leq x$. In order to represent both delay and failure in a single quantity, a ΔQ is represented by an improper random variable (IRV), allowing the total probability not to reach 100% [21]. The *intangible mass* of such an IRV is $\Im(\Delta\mathrm{Q}) = 1 - \lim_{x\to\infty} \Delta Q(x)$. For a given ΔQ, the intangible mass $\Im(\Delta\mathrm{Q})$ encodes the probability of exceptions or failure occurring.

Denote the set $\mathbb{I}$ of all IRVs that are differentiable and the values of which are always greater than or equal to zero. Statistically speaking, every $\iota \in \mathbb{I}$ can be represented both using its Probability Density Function (PDF) or its Cumulative Distribution Function (CDF), where the former is the derivative of the latter. For convenience, we will freely switch between the two representations as the need rises. Fix a countable set of ΔQ variables Δ_v. We define $\Delta = \Delta_v \cup \mathbb{I}$ to denote both IRVs and ΔQ variables. When $\delta \in \Delta$ is in its CDF representation, we write δ' for its derivative, which is the PDF representation.

We first define a mapping between primitive outcomes $\overline{\mathbb{B}}$ and ΔQs.

Definition 2. We call a function $\Delta_\circ[\![.]\!] : \overline{\mathbb{B}} \to \Delta$ a *basic assignment* when $\Delta_\circ[\![\top]\!] = \mathbf{1}$ and $\Delta_\circ[\![\bot]\!] = \mathbf{0}$, where $\mathbf{1}$ and $\mathbf{0}$ are the functions always returning the constants 1 and 0, respectively.

We now define the semantics of an outcome expression as a mapping between the outcome expression and an IRV, for a given basic assignment.

Definition 3 (Haeri et al. [11])**.** Given a basic assignment $\Delta_\circ[\![.]\!] : \overline{\mathbb{B}} \to \Delta$, define $\Delta\mathrm{Q}[\![.]\!]_{\Delta_\circ} : \mathbb{O} \to \mathbb{I}$ such that

$$
\begin{aligned}
\Delta\mathrm{Q}[\![\beta]\!]_{\Delta_\circ} &= \begin{cases} \mathbf{1} & \text{when } \Delta_\circ[\![\beta]\!] \notin \mathbb{I} \\ \Delta_\circ[\![\beta]\!] & \text{otherwise} \end{cases} \\
\Delta\mathrm{Q}[\![o \bullet\!\!\to\!\!\bullet\, o']\!]_{\Delta_\circ} &= \Delta\mathrm{Q}[\![o]\!]_{\Delta_\circ} * \Delta\mathrm{Q}[\![o']\!]_{\Delta_\circ} \\
\Delta\mathrm{Q}[\![o \underset{m'}{\overset{m}{\longrightarrow}} o']\!]_{\Delta_\circ} &= \tfrac{m}{m+m'}\Delta\mathrm{Q}[\![o]\!]_{\Delta_\circ} + \tfrac{m'}{m+m'}\Delta\mathrm{Q}[\![o']\!]_{\Delta_\circ} \\
\Delta\mathrm{Q}[\![o \parallel^{\forall} o']\!]_{\Delta_\circ} &= \Delta\mathrm{Q}[\![o]\!]_{\Delta_\circ} \times \Delta\mathrm{Q}[\![o']\!]_{\Delta_\circ} \\
\Delta\mathrm{Q}[\![o \parallel^{\exists} o']\!]_{\Delta_\circ} &= \Delta\mathrm{Q}[\![o]\!]_{\Delta_\circ} + \Delta\mathrm{Q}[\![o']\!]_{\Delta_\circ} - \Delta\mathrm{Q}[\![o]\!]_{\Delta_\circ} \times \Delta\mathrm{Q}[\![o']\!]_{\Delta_\circ}
\end{aligned}
$$

The notation $*$ denotes the convolution of two ΔQs. In the above formulae, the random variables are always represented using their CDFs except for sequential composition, where the representation is PDFs on both sides. Note that the PDF of $\top$ is the Dirac δ function. In what follows, we will drop $\Delta_\circ$ whenever the basic assignment is fixed throughout a computation.

Recall that, in Section 3, we defined timeliness as $\Delta\mathrm{Q}_{obs} \leq \Delta\mathrm{Q}_{req}$ (this relation is a partial order, defined in [11]). Definition 3 gives this more context. Using Definition 3, the systems engineer can work out the $\Delta\mathrm{Q}_{obs}$ of an outcome so they can compare the result against the required $\Delta\mathrm{Q}_{req}$.

Remark 1. Note that, according to Definition 3, we get $\Delta\mathrm{Q}[\![o_1 \bullet\!\!\to\!\!\bullet\, o_2]\!] = \Delta\mathrm{Q}[\![o_2 \bullet\!\!\to\!\!\bullet\, o_1]\!]$. This may seem counter-intuitive because $o_1 \bullet\!\!\to\!\!\bullet\, o_2 \neq o_2 \bullet\!\!\to\!\!\bullet\, o_1$. $\Delta\mathrm{Q}[\![o_1 \bullet\!\!\to\!\!\bullet\, o_2]\!] = \Delta\mathrm{Q}[\![o_2 \bullet\!\!\to\!\!\bullet\, o_1]\!]$ is, nonetheless, valid because, intuitively, $o_1 \bullet\!\!\to\!\!\bullet\, o_2$ is just as timely as $o_2 \bullet\!\!\to\!\!\bullet\, o_1$. See the proof of Theorem 2 [12] for the mathematical justification of that intuition.

4.3 Connecting Algebra to Timeliness

In our accompanying technical report [12], we give a model theoretic formulation for studying the algebraic properties of resource consumption. This paper focuses on time as its solo resource of interest and uses that formulation for time exclusively without getting into the technical details of the formulation itself.

An algebraic structure often consists of a carrier set, a few operations on the carrier set, and a finite set of identities that those operations need to satisfy. Given our focus on timeliness à la ΔQSD, the carrier set will always be $\mathbb{O}$ in this paper. The full set of operators on $\mathbb{O}$ is $\{\bullet\!\!\rightarrow\!\!\bullet, \|^{\forall}, \|^{\exists}, \leftrightharpoons\}$. However, most algebraic structures do not need all those operators. Different structures work with different number of operations. (For example, a monoid works with only one operation; whilst a group works with two.) Finally, the identities are of the form $o_l = o_r$.

We take $\Delta\mathrm{Q}[\![.]\!]$ (Definition 3) to be the model of time consumption for $\mathbb{O}$. We write

- $\circledcirc\!\!\odot\, time \vDash o_l = o_r$ when $\Delta\mathrm{Q}[\![o_l]\!] = \Delta\mathrm{Q}[\![o_r]\!]$. That is when o_l and o_r are as timely.
- $\odot\!\!\odot\, time \vDash (\mathbb{O}, P) : s$ for an algebraic structure s and a set of ΔQSD operators P when $\circledcirc\!\!\odot\, time \vDash o_l = o_r$, for every equation $o_l = o_r$
  - that is constructed using the operators in P, and
  - that is required for the formation of s.

With time being our solo resource of interest in this paper, we might drop the initial "$\circledcirc\!\!\odot\, time \vDash$" from the above formulation hereafter.

5 Algebraic Structures

This section establishes several important properties on $\mathbb{O}$:

- probabilistic choice forms a magma (Theorem 1);
- sequential composition forms a commutative monoid with $\top$ and $\bot$ as the identity and absorbing elements (Theorem 2);
- all-to-finish forms a commutative monoid with $\top$ and $\bot$ as the identity and absorbing elements (Theorem 3);
- any-to-finish forms a commutative monoid with $\bot$ and $\top$ as the identity and absorbing elements (Theorem 4); and
- neither all-to-finish nor any-to-finish nor their combination form the familiar richer algebraic structures (Remarks 2, 3, and 4).

Theorem 1. $(\mathbb{O}, \leftrightharpoons)$ *forms a magma when observing time.*

A magma is the weakest algebraic structure. That is because $\leftrightharpoons$ is not even associative. Despite this, expressions containing two consecutive occurrences of $\leftrightharpoons$ can still be re-associated. However, in this case the coefficients will change. Lemmas 2 and 3 give the exact formulae.

Theorem 2. $\circledcirc\!\!\odot\, time \vDash (\mathbb{O}, \bullet\!\!\rightarrow\!\!\bullet) : forms\ a\ commutative\ monoid\ with\ \top\ and\ \bot\ as\ the\ identity\ and\ absorbing\ elements,\ respectively.$

Theorem 3. $\circledcirc\!\!\odot\, time \vDash (\mathbb{O}, \|^{\forall}) : forms\ a\ commutative\ monoid\ with\ \top\ and\ \bot\ as\ the\ identity\ and\ absorbing\ elements,\ respectively.$

Remark 2. It is important to notice that, when observing time, $(\mathbb{O}, \|^{\forall})$ does *not* form a group. That is because, in general, an outcome has no inverse element - intuitively, one can never undo an outcome!

In order to prove that claim formally, suppose otherwise. That is, suppose that there exist a pair of outcomes o_1 and o_2 such that $o_1 \|^{\forall} o_2 = \top$. Then, $\Delta\mathrm{Q}[\![o_1 \|^{\forall} o_2]\!] = \Delta\mathrm{Q}[\![\top]\!]$ which implies $\delta_1 \times \delta_2 = \mathbf{1} \Rightarrow \delta_2 = \frac{\mathbf{1}}{\delta_1}$. However, given that $\delta_1 \leq \mathbf{1}$, we get $\delta_2 \geq \mathbf{1}$. The latter inequality can only be satisfied when $o_1 = \top$. Restricting the application of ΔQSD to perfection is not practical.

Theorem 4. $\circledcirc\!\!\cdot$ *time* $\vDash (\mathbb{O}, \|^{\exists})$: *forms a commutative monoid with* $\bot$ *and* $\top$ *as the identity and absorbing elements, respectively.*

Remark 3. Similar to the case for $\|^{\forall}$, it is important to note that, when observing time, $(\mathbb{O}, \|^{\exists})$ does not form a group. Again, it is the lack of an inverse element that is causing the trouble. Here is how. Suppose that there exist a pair of outcomes o_1 and o_2 such that $o_1 \|^{\exists} o_2 = \bot$. Then, $\Delta\mathrm{Q}[\![o_1 \|^{\exists} o_2]\!] = \Delta\mathrm{Q}[\![\bot]\!]$ which implies $\delta_1 + \delta_2 - \delta_1 \times \delta_2 = \mathbf{0} \Rightarrow \delta_2 = \frac{\delta_1}{\delta_1 - 1}$. However, because $\delta_1 \leq \mathbf{1}$, we get $\delta_2 \leq \mathbf{0}$. But, only $\bot$ can satisfy the latter inequality. There is no reason to develop a system in which all the outcomes will fail unconditionally!

Having established that both $(\mathbb{O}, \|^{\forall})$ and $(\mathbb{O}, \|^{\exists})$ form commutative monoids for time, a natural question is whether $(\mathbb{O}, \|^{\forall}, \|^{\exists})$ or $(\mathbb{O}, \|^{\exists}, \|^{\forall})$ form semi-rings. This is not the case, since they do not distribute over one another.

Lemma 1 helps Remark 4 demonstrate how the above desirable distributivities fail.

Lemma 1. $\circledcirc\!\!\cdot$ *time* $\vDash o_1 \|^{\exists} o_2 = \top$ *implies* $o_1 = \top$ *and* $o_2 = \top$.

Remark 4. Neither $(\mathbb{O}, \|^{\forall}, \|^{\exists})$ nor $(\mathbb{O}, \|^{\exists}, \|^{\forall})$ form a semi-ring when observing time: for this to be the case, $\|^{\forall}$ and $\|^{\exists}$ would need to distribute over one another. The first distributivity requirement is:

$$o_1 \|^{\exists} (o_2 \|^{\forall} o_3) \stackrel{?}{=} (o_1 \|^{\exists} o_2) \|^{\forall} (o_1 \|^{\exists} o_3) \tag{7}$$

Equating $\Delta\mathrm{Q}[\![.]\!]$s of the two sides, one eventually makes it to the requirement that either $\delta_1 = \mathbf{0}$ or $\Delta\mathrm{Q}[\![(o_1 \|^{\exists} o_3) \|^{\exists} o_2]\!] = \top$. In other words, it follows by Lemma 1 that Equation (7) can only hold under the trivial conditions when either $o_1 = \bot$ or $o_1 = o_2 = o_3 = \top$. The second distributivity requirement is

$$o_1 \|^{\forall} (o_2 \|^{\exists} o_3) \stackrel{?}{=} (o_1 \|^{\forall} o_2) \|^{\exists} (o_1 \|^{\forall} o_3) \tag{8}$$

Again, equating $\Delta\mathrm{Q}[\![.]\!]$s of the two sides, one eventually comes to observe that Equation (8) only holds when $\delta_1 = \mathbf{1} \wedge \delta_2 \neq \mathbf{0} \wedge \delta_3 \neq \mathbf{0}$, i.e., when $o_1 = \top \wedge o_2 \neq \bot \wedge o_3 \neq \bot$.

6 Equivalences Containing Constant Outcomes

ΔQSD is already in use by its practitioners, who, amongst other usages, simplify outcome expressions according to their timeliness analysis. In particular, Figure 7 distils a list of equivalences that are used in such simplifications. Those equivalences all contain constant outcomes ($\top$ or $\bot$).

Equivalences of Figure 7 provide the basis for rewrite rules that are useful for construction of normal forms, such as expressing a given system as a convolution of probabilistic choices or a probabilistic choice of convolutions. Such rewriting allows for: extraction of common sub-expressions permitting aggregation of failure rates (distinguishing between conditional and non-conditional failure); identifying minimal delays; and highlighting branching probabilities to identify issues of relative criticality. This is useful for quickly assessing whether a particular outcome decomposition is *feasible* without having to

$$\bot \leftrightharpoons \bot = \bot \qquad (o_1 \leftrightharpoons \bot) \bullet\!\!\rightarrow\!\!\bullet\, o_2 = (o_1 \bullet\!\!\rightarrow\!\!\bullet\, o_2) \leftrightharpoons \bot \qquad o \bullet\!\!\rightarrow\!\!\bullet\, \bot = \bot \qquad \top \leftrightharpoons \top = \top$$

$$\bot \bullet\!\!\rightarrow\!\!\bullet\, o = \bot \qquad o_1 \bullet\!\!\rightarrow\!\!\bullet\, (o_2 \leftrightharpoons \bot) = (o_1 \bullet\!\!\rightarrow\!\!\bullet\, o_2) \leftrightharpoons \bot \qquad \top \bullet\!\!\rightarrow\!\!\bullet\, o = o \qquad o \bullet\!\!\rightarrow\!\!\bullet\, \top = o$$

$$\top \|^{\forall} o = o \qquad (o_1 \leftrightharpoons \top) \bullet\!\!\rightarrow\!\!\bullet\, o_2 = (o_1 \bullet\!\!\rightarrow\!\!\bullet\, o_2) \leftrightharpoons o_2 \qquad o_1 \bullet\!\!\rightarrow\!\!\bullet\, (o_2 \leftrightharpoons \top) = (o_1 \bullet\!\!\rightarrow\!\!\bullet\, o_2) \leftrightharpoons o_1$$

$$\bot \|^{\exists} o = o \qquad o_1 \overset{[p]}{\leftrightharpoons} (o_2 \overset{[q]}{\leftrightharpoons} \top) = o_2 \overset{[q(1-p)]}{\leftrightharpoons} (o_1 \overset{\left[\frac{p}{1-q(1-p)}\right]}{\leftrightharpoons} \top) \qquad \bot \overset{[p]}{\leftrightharpoons} (\bot \overset{[q]}{\leftrightharpoons} o) = \bot \overset{[p+(1-p)q]}{\leftrightharpoons} o$$

Figure 7: Equivalences Containing $\top$ and $\bot$

compute the complete ΔQ. See Section 2, for example. In addition, the equivalences of Figure 7 are very handy in the proofs of properties such as those established in this paper. Two examples, amongst many, are the proofs of Theorem 9 and Lemma 4.

Before we delve into Figure 7, we prove a result about re-associating probabilistic choice. Given an expression with two consecutive probabilistic choices, one of which wrapped inside a pair of parentheses, the ΔQSD practitioner might be interested in wrapping the other two inside a pair of parentheses – re-associating the probabilistic choices, in effect. Lemmata 2 and 3 give the conditions on the coefficients of those probabilistic choices.

Lemma 2. $o_1 \overset{[p]}{\leftrightharpoons} (o_2 \overset{[q]}{\leftrightharpoons} o_3) = (o_1 \overset{[p']}{\leftrightharpoons} o_2) \overset{[q']}{\leftrightharpoons} o_3$ *iff* $p' = \frac{p}{1-(1-p)(1-q)}$ *and* $q' = 1-(1-p)(1-q)$.

Lemma 3. $(o_1 \overset{[p]}{\leftrightharpoons} o_2) \overset{[q]}{\leftrightharpoons} o_3 = o_1 \overset{[p']}{\leftrightharpoons} (o_2 \overset{[q']}{\leftrightharpoons} o_3)$ *iff* $p' = pq$ *and* $q' = \frac{q(1-p)}{1-pq}$.

Theorem 5. *The equivalences in Fig. 7 are correct.*

Proof. We will only present the proof of $\bot \overset{m_1}{\underset{m_2}{\leftrightharpoons}} \bot = \bot$ here. The rest of the equivalences are proved similarly:

$$\Delta Q[\![\bot \overset{m_1}{\underset{m_2}{\leftrightharpoons}} \bot]\!] = \frac{m_1}{m_1+m_2}\mathbf{0} + \frac{m_2}{m_1+m_2}\mathbf{0} = \mathbf{0} = \Delta Q[\![\bot]\!].$$

□

Remark 5. The very last equivalence in Fig. 7 was incorrectly formulated (though never published) prior to this paper. Thanks to the formalisation developed in [11], that mistake was corrected, and the equivalences have been given a sound footing.

7 Distributivity

In this section, we consider the distributivity results between the ΔQSD operators. Recall that out of the four $\mathbb{P}$ operators, three are commutative (i.e., $\bullet\!\!\rightarrow\!\!\bullet$, $\|^{\forall}$, and $\|^{\exists}$) and one is not (i.e., $\leftrightharpoons$). Hence, it is only possible for right- and left-distributivity to differ when $\leftrightharpoons$ is the outermost operator. That gives rise to $2 \times \binom{3}{1} + \binom{3}{1}\binom{3}{1} = 15$ possible ways for distributing $\mathbb{P}$ operators over each other. Theorem 6 establishes 3 of those 15. In Section 7.1, we show how the routine technique for examining the equivalence of expressions (i.e., equating the $\Delta Q[\![.]\!]$ of the two sides) is not that helpful for the study of the remaining 12 distributivity results. That leads to Sections 7.2 and 7.3, which disprove the generality of 8 and 4 distributivity results using properisation (Theorem 8) and counterexamples (Theorem 9), respectively.

We use the following syntactic convention: when, in an equivalence, two $\leftrightharpoons$s are used without weights, each on precisely one side of the equivalence, we will assume that the weights of those $\leftrightharpoons$s are the same. We therefore do not bother to repeat those weights. For example, in the theorem below, there exist weights m_2 and m_3 such that $o_2 \overset{m_2}{\underset{m_3}{\leftrightharpoons}} o_3$ and $(o_1 \bullet\!\!\rightarrow\!\!\bullet\, o_2) \overset{m_2}{\underset{m_3}{\leftrightharpoons}} (o_1 \bullet\!\!\rightarrow\!\!\bullet\, o_3)$, but we omit these.

Theorem 6. *Let $o_1, o_2, o_3 \in \mathbb{O}$ and $p \in \{\bullet\!\!\rightarrow\!\!\bullet, \|^{\forall}, \|^{\exists}\}$. Then,*

- $\circledcirc$ *time* $\vDash o_1\ p\ (o_2 \leftrightharpoons o_3) = (o_1\ p\ o_2) \leftrightharpoons (o_1\ p\ o_3)$, *and*
- $\circledcirc$ *time* $\vDash (o_1 \leftrightharpoons o_2)\ p\ o_3 = (o_1\ p\ o_3) \leftrightharpoons (o_2\ p\ o_3)$.

7.1 Potential Distributivity

As we are going to see in Sections 7.2 and 7.3, the remaining 12 potential distributivity results do not hold **in general**. Nevertheless, this section uses the routine technique for studying the equivalence of expressions: Equating the $\Delta\mathrm{Q}[\![.]\!]$ of the two sides. That is important because

- firstly, it shows why the routine technique does not help, thereby motivating the next sections.
- secondly, it presents some of the necessary conditions for those distributivity results to hold. Although pretty immature, such conditions help the ΔQSD practitioner to verify, under special circumstances, whether their given IRVs can satisfy the provided conditions.

We do not know of better necessary conditions for the remaining 12 results (*if indeed they are soluble at all*). In this section, we demonstrate the necessary conditions of one distributivity result out the 12.

We begin by Proposition 1, which is a simple yet handy result.

Proposition 1. *Suppose that $o_1 = o_2 \bullet\!\!\rightarrow\!\!\bullet\, o_3$. Then, $\circledcirc$ time $\vDash \delta_1(t) = \int (\delta_2' * \delta_3')(t)\,\mathrm{d}t$.*

When observing time, for

$$(o_1 \bullet\!\!\rightarrow\!\!\bullet\, o_2) \overset{m}{\underset{m'}{\leftleftarrows}} o_3 \overset{?}{=} (o_1 \overset{m}{\underset{m'}{\leftleftarrows}} o_3) \bullet\!\!\rightarrow\!\!\bullet\, (o_2 \overset{m}{\underset{m'}{\leftleftarrows}} o_3) \tag{9}$$

to hold, according to Proposition 1,

$$\begin{aligned}\Delta\mathrm{Q}[\![(o_1 \bullet\!\!\rightarrow\!\!\bullet\, o_2) \overset{m}{\underset{m'}{\leftleftarrows}} o_3]\!] &= \frac{m}{m+m'} \int (\delta_1' * \delta_2')(t)\,\mathrm{d}t + \frac{m'}{m+m'}\delta_3 \\ &= \frac{m}{m+m'} \iint \delta_1'(\tau)\delta_2'(t-\tau)\,\mathrm{d}\tau\,\mathrm{d}t + \frac{m'}{m+m'}\delta_3 \end{aligned} \tag{10}$$

and

$$\begin{aligned}\Delta\mathrm{Q}[\![(o_1 \overset{m}{\underset{m'}{\leftleftarrows}} o_3) \bullet\!\!\rightarrow\!\!\bullet\, (o_2 \overset{m}{\underset{m'}{\leftleftarrows}} o_3)]\!] &= \int \left(\frac{m}{m+m'}\delta_1' + \frac{m'}{m+m'}\delta_3'\right) * \left(\frac{m}{m+m'}\delta_2' + \frac{m'}{m+m'}\delta_3'\right)(t)\,\mathrm{d}t \\ &= \iint \left(\frac{m}{m+m'}\delta_1'(t) + \frac{m'}{m+m'}\delta_3'(t)\right) \times \left(\frac{m}{m+m'}\delta_2'(t-\tau) + \frac{m'}{m+m'}\delta_3'(t-\tau)\right) \mathrm{d}\tau\,\mathrm{d}t. \end{aligned} \tag{11}$$

For Equation (9) to hold, the right-hand-sides of Equations (10) and (11) need to be equal. That is,

$$\begin{aligned}&\frac{m}{m+m'} \iint \delta_1'(\tau)\delta_2'(t-\tau)\,\mathrm{d}\tau\,\mathrm{d}t + \frac{m'}{m+m'}\delta_3 = \\ &\quad \iint \left(\frac{m}{m+m'}\delta_1'(t) + \frac{m'}{m+m'}\delta_3'(t)\right) \times \left(\frac{m}{m+m'}\delta_2'(t-\tau) + \frac{m'}{m+m'}\delta_3'(t-\tau)\right) \mathrm{d}\tau\,\mathrm{d}t \end{aligned} \tag{12}$$

This is a differential equation for which we do not know a general solution. Given particular values for δ_1, δ_2, and δ_3, however, the ΔQSD practitioner might be able to solve it.

7.2 Properisation

This section sets the stage using Theorem 7 for a technique that we call *properisation* and use for disproving equivalences (in their full generality).

Properisation is based on the following important observation: if two outcomes do not fail similarly, they are not equivalent. Properisation is an algebraic technique for swiftly extracting the failure behaviour of outcomes via rewriting but without assessing the rest of their timeliness behaviour. Once the failure parts of the timeliness behaviours are at hand for the two sides, one can check whether they are equal, and if they are not, deduce that the outcomes in question are therefore unequal.

Our intuition for the choice of name "properisation" for this technique follows: recall that ΔQs are CDFs (or PDFs) of **im**proper random variables. Properisation is a technique based on making the ΔQ of an outcome o proper (by proportionating it) and restoring its amount of improperness – i.e., o's intangible mass, denoted by $\Im(\Delta\mathrm{Q}(o))$ – as a probabilistic choice (of the right weights) between o and $\bot$. That is also the intention behind the symbol we use for properisation: "$\overline{\uparrow}$." As one can see in Figure 5, the CDF of an improper random variable needs not to make it to the "ceiling" (i.e., 1). The symbol "$\overline{\uparrow}$" that we use is to resemble the act of sticking the CDF to the ceiling (represented by the horizontal bar at the top of "$\overline{\uparrow}$")!

Now, the formal definition of properisation; both of an outcome and a basic assignment.

Notation 1. Write $o[o'/\beta]$ for the familiar λ-Calculus notation for substitutions: o in which every instance of β is replaced by o'.

Definition 4. Fix a basic assignment Δ and a base variable β such that $\Delta(\beta) = \delta$ where $\Im(\delta) = i$. Write $o\overline{\uparrow}^{\beta}_{\Delta} = o[(\beta \overset{[1-i]}{\leftrightharpoons} \bot)/\beta]$. Also, write $\Delta\overline{\uparrow}^{\beta}$ for the basic assignment such that

$$\Delta\overline{\uparrow}^{\beta}(\beta') = \Delta(\beta') \quad \text{when } \beta' \neq \beta \qquad\qquad \Delta\overline{\uparrow}^{\beta}(\beta') = \tfrac{1}{1-i}\delta \quad \text{otherwise.}$$

Finally, write $o\overline{\uparrow}^{\beta_1,\beta_2}_{\Delta}$ for $\left(o\overline{\uparrow}^{\beta_1}_{\Delta}\right)\overline{\uparrow}^{\beta_2}_{\Delta}$ and $\Delta\overline{\uparrow}^{\beta_1,\beta_2}$ for $\left(\Delta\overline{\uparrow}^{\beta_1}\right)\overline{\uparrow}^{\beta_2}$.

We say $\Delta\overline{\uparrow}^{\beta}$ is the result of *properisation* of β in Δ. Likewise, we say that $o\overline{\uparrow}^{\beta}_{\Delta}$ is the result *properisation* of β in o according to Δ.

As one can see from Definition 4, the act of properisation of a base variable β is according to a given basic assignment Δ. That act is performed by taking two steps in unison:

1. proportionating according to the intangible mass of $\Delta(\beta)$ so that β is no longer improper in the resulting new basic assignment $\Delta\overline{\uparrow}^{\beta}$; and,
2. replacing every occurrence of β in every outcome o with the probabilistic choice that is weighted according to the intangible mass of $\Delta(\beta)$, resulting in the new outcome $o\overline{\uparrow}^{\beta}_{\Delta}$.

We now have all the prerequisites of Theorem 7.

Theorem 7. *Suppose Δ and Δ' are two basic assignments such that $\Delta' = \Delta\overline{\uparrow}^{\beta_1,\beta_2,\ldots,\beta_n}$, for some $\beta_1, \beta_2, \ldots, \beta_n \in \overline{\mathbb{B}}$. Suppose also that $o_1, o_2 \in \mathbb{O}$. Then, $\Delta Q[\![o_1]\!]_{\Delta} = \Delta Q[\![o_2]\!]_{\Delta}$ iff*

$$\Delta Q[\![o_1\overline{\uparrow}^{\beta_1,\beta_2,\ldots,\beta_n}_{\Delta}]\!]_{\Delta'} = \Delta Q[\![o_2\overline{\uparrow}^{\beta_1,\beta_2,\ldots,\beta_n}_{\Delta}]\!]_{\Delta'}.$$

Armed with Theorem 7, we can now outline the properisation technique:

Suppose two outcome expressions o and o' the equivalence of which is to be studied. One begins by studying the equivalence of $o\overline{\uparrow}^{\beta_1,\ldots,\beta_n}$ and $o'\overline{\uparrow}^{\beta_1,\ldots,\beta_n}$ for some $\beta_1, \ldots, \beta_n \in \overline{\mathbb{B}}$. Now, suppose that

– after the application of algebraic laws – one gets to rewrite $o\overset{\top}{\uparrow}^{\beta_1,\ldots,\beta_n}$ to $(\ldots)\underset{\rightharpoonup}{\overset{[p]}{\leftharpoondown}}\bot$ and $o'\overset{\top}{\uparrow}^{\beta_1,\ldots,\beta_n}$ to $(\ldots)\underset{\rightharpoonup}{\overset{[p']}{\leftharpoondown}}\bot$. One concludes that $o \neq o'$ if one can show that $p \neq p'$.

We start the application of our properisation technique by obtaining some useful results. Lemma 4 paves the way for the applications of the above technique. They instruct one on how to accumulate failure at the rightmost corner when the operator between two pairs of parentheses is $\bullet\!\!\rightarrow\!\!\bullet$, $\leftrightharpoons$, and $\|^{\forall}$, respectively. Unfortunately, $\|^{\exists}$ has no such property, as will be shown by Remark 6.

Lemma 4. *For every $o_1, o_2, o_3 \in \mathbb{O}$,*

$$(o_1 \underset{\rightharpoonup}{\overset{[p_1]}{\leftharpoondown}} \bot) \bullet\!\!\rightarrow\!\!\bullet (o_2 \underset{\rightharpoonup}{\overset{[p_2]}{\leftharpoondown}} \bot) = (o_1 \bullet\!\!\rightarrow\!\!\bullet o_2) \underset{\rightharpoonup}{\overset{[p_1p_2]}{\leftharpoondown}} \bot$$
$$(o_1 \underset{\rightharpoonup}{\overset{[p_1]}{\leftharpoondown}} \bot) \underset{\rightharpoonup}{\overset{[p]}{\leftharpoondown}} (o_2 \underset{\rightharpoonup}{\overset{[p_2]}{\leftharpoondown}} \bot) = (o_1 \underset{\rightharpoonup}{\overset{[q]}{\leftharpoondown}} o_2) \underset{\rightharpoonup}{\overset{[r]}{\leftharpoondown}} \bot \text{ where } q = \frac{pp_1}{p_2 - pp_2 + pp_1} \text{ and } r = p_2 - pp_2 + pp_1$$
$$(o_1 \underset{\rightharpoonup}{\overset{[p_1]}{\leftharpoondown}} \bot) \|^{\forall} (o_2 \underset{\rightharpoonup}{\overset{[p_2]}{\leftharpoondown}} \bot) = (o_1 \|^{\forall} o_2) \underset{\rightharpoonup}{\overset{[p_1p_2]}{\leftharpoondown}} \bot.$$

Proof. We only prove the first equivalence here. The proof is similar for the other two equivalences.

By Theorems 6 and 5,

$$(o_1 \underset{\rightharpoonup}{\overset{[p_1]}{\leftharpoondown}} \bot) \bullet\!\!\rightarrow\!\!\bullet (o_2 \underset{\rightharpoonup}{\overset{[p_2]}{\leftharpoondown}} \bot) = ((o_1 \underset{\rightharpoonup}{\overset{[p_1]}{\leftharpoondown}} \bot) \bullet\!\!\rightarrow\!\!\bullet o_2) \underset{\rightharpoonup}{\overset{[p_2]}{\leftharpoondown}} \bot = ((o_1 \bullet\!\!\rightarrow\!\!\bullet o_2) \underset{\rightharpoonup}{\overset{[p_1]}{\leftharpoondown}} \bot) \underset{\rightharpoonup}{\overset{[p_2]}{\leftharpoondown}} \bot = (o_1 \bullet\!\!\rightarrow\!\!\bullet o_2) \underset{\rightharpoonup}{\overset{[p_1p_2]}{\leftharpoondown}} \bot.$$

□

Remark 6. Interestingly enough, there is no p such that the following holds in its full generality:

$$(o_1 \underset{\rightharpoonup}{\overset{[p_1]}{\leftharpoondown}} \bot) \|^{\exists} (o_2 \underset{\rightharpoonup}{\overset{[p_2]}{\leftharpoondown}} \bot) \overset{?}{=} (o_1 \|^{\exists} o_2) \underset{\rightharpoonup}{\overset{[p]}{\leftharpoondown}} \bot.$$

Suppose there were such a p. One gets to observe after some workout that equating the $\Delta\mathrm{Q}[\![.]\!]$ of the two sides implies $p = p_1 = p_2 = 1$ or $p = p_1 = p_2 = 0$. When $(o_1 \underset{\rightharpoonup}{\overset{[p_1]}{\leftharpoondown}} \bot) \|^{\exists} (o_2 \underset{\rightharpoonup}{\overset{[p_2]}{\leftharpoondown}} \bot)$ is $o_1 \|^{\exists} o_2$, in which o_1 and o_2 are being properised, that is either when $o_1 = o_2 = \top$ or $o_1 = o_2 = \bot$.

Hereafter, we will write $o_1 \underset{\rightharpoonup}{\overset{[.]}{\leftharpoondown}} o_2$ to mean $o_1 \underset{\rightharpoonup}{\overset{[p]}{\leftharpoondown}} o_2$ for some unimportant p.

The desirable inequalities in Theorem 8 are all of the form $o_l \neq o_r$, with the outcome variables in o_l and o_r being o_1, o_2, and o_3. In order to show $o_l \neq o_r$, we proceed by properisation of o_1, o_2, and o_3 in o_l and o_r.

To that end, we fix a basic assignment Δ, such that $\Delta\mathrm{Q}[\![o_k]\!]_\Delta = \delta_k$ and $\Im(\delta_k) = i_k$ for $k \in \{1,2,3\}$. Then, we take $p_k = 1 - i_k$ for $k \in \{1,2,3\}$, $o'_k = o_k \overset{\top}{\uparrow}_{\Delta}^{o_1,o_2,o_3}$ for $k \in \{l,r\}$, and $\Delta' = \Delta\overset{\top}{\uparrow}^{o_1,o_2,o_3}$. We show that $\Delta\mathrm{Q}[\![o'_l]\!]_{\Delta'} \neq \Delta\mathrm{Q}[\![o'_r]\!]_{\Delta'}$ to conclude that $\Delta\mathrm{Q}[\![o_l]\!]_\Delta \neq \Delta\mathrm{Q}[\![o_r]\!]_\Delta$ by Theorem 7 and the result follows.

Theorem 8. *For every $o_1, o_2, o_3 \in \mathbb{O}$,*

$$(o_1 \bullet\!\!\rightarrow\!\!\bullet o_2) \leftrightharpoons o_3 \neq (o_1 \leftrightharpoons o_3) \bullet\!\!\rightarrow\!\!\bullet (o_2 \leftrightharpoons o_3) \qquad o_1 \leftrightharpoons (o_2 \bullet\!\!\rightarrow\!\!\bullet o_3) \neq (o_1 \leftrightharpoons o_2) \bullet\!\!\rightarrow\!\!\bullet (o_1 \leftrightharpoons o_3)$$
$$(o_1 \|^{\forall} o_2) \leftrightharpoons o_3 \neq (o_1 \leftrightharpoons o_3) \|^{\forall} (o_2 \leftrightharpoons o_3) \qquad o_1 \leftrightharpoons (o_2 \|^{\forall} o_3) \neq (o_1 \leftrightharpoons o_2) \|^{\forall} (o_1 \leftrightharpoons o_3)$$
$$(o_1 \|^{\forall} o_2) \bullet\!\!\rightarrow\!\!\bullet o_3 \neq (o_1 \bullet\!\!\rightarrow\!\!\bullet o_3) \|^{\forall} (o_2 \bullet\!\!\rightarrow\!\!\bullet o_3) \qquad o_1 \bullet\!\!\rightarrow\!\!\bullet (o_2 \|^{\forall} o_3) \neq (o_1 \bullet\!\!\rightarrow\!\!\bullet o_2) \|^{\forall} (o_1 \bullet\!\!\rightarrow\!\!\bullet o_3).$$

Proof. We only prove

$$(o_1 \bullet\!\!\rightarrow\!\!\bullet o_2) \underset{\rightharpoonup}{\overset{[p]}{\leftharpoondown}} o_3 \neq (o_1 \underset{\rightharpoonup}{\overset{[p]}{\leftharpoondown}} o_3) \bullet\!\!\rightarrow\!\!\bullet (o_2 \underset{\rightharpoonup}{\overset{[p]}{\leftharpoondown}} o_3) \tag{13}$$

for a given p here. The rest can be proved similarly using Lemma 4.

One can rewrite the left-hand-side of Inequality (13)'s properisation using Lemma 4 as $((o_1 \bullet\!\!\rightarrow\!\!\bullet o_2) \underset{\rightharpoonup}{\overset{[.]}{\leftharpoondown}} o_3) \underset{\rightharpoonup}{\overset{[q]}{\leftharpoondown}} \bot$ where $q = p_3 - pp_3 + pp_1p_2$. Likewise, the right-hand-side of Inequality (13)'s properisation can be rewritten as $((o_1 \underset{\rightharpoonup}{\overset{[.]}{\leftharpoondown}} o_3) \bullet\!\!\rightarrow\!\!\bullet (o_2 \underset{\rightharpoonup}{\overset{[.]}{\leftharpoondown}} o_3)) \underset{\rightharpoonup}{\overset{[r_1r_2]}{\leftharpoondown}} \bot$ where $r_1 = p_3 - pp_3 + pp_1$ and $r_2 = p_3 - pp_3 + pp_2$. Should the desirable inequality not hold, the conclusion would be $q = r_1r_2$. That is $p_3 - pp_3 + pp_1p_2 = (p_3 - pp_3 + pp_1)(p_3 - pp_3 + pp_2)$. But, that is not an equation that holds in general. □

7.3 Counterexamples

As worked out in Remark 6, properisation does not quite work for outcome expressions containing $\|^{\exists}$ because $\bot$ is not compositional under $\|^{\exists}$. In this section, we present another technique for refuting distributivity results, which is even easier: counterexamples. It suffices for one to refute an equivalence to simply provide a single counterexample. That is how we refute the remaining four distributivity results (in their full generality).

Theorem 9. *For every $o_1, o_2, o_3 \in \mathbb{O}$,*

$$o_1 \leftrightharpoons (o_2 \|^{\exists} o_3) \neq (o_1 \leftrightharpoons o_2) \|^{\exists} (o_1 \leftrightharpoons o_3) \qquad (o_1 \|^{\exists} o_2) \leftrightharpoons o_3 \neq (o_1 \leftrightharpoons o_3) \|^{\exists} (o_2 \leftrightharpoons o_3)$$
$$o_1 \|^{\exists} (o_2 \bullet\!\!\rightarrow\!\!\bullet\, o_3) \neq (o_1 \|^{\exists} o_2) \bullet\!\!\rightarrow\!\!\bullet\, (o_1 \|^{\exists} o_3) \qquad o_1 \bullet\!\!\rightarrow\!\!\bullet\, (o_2 \|^{\exists} o_3) \neq (o_1 \bullet\!\!\rightarrow\!\!\bullet\, o_2) \|^{\exists} (o_1 \bullet\!\!\rightarrow\!\!\bullet\, o_3).$$

Proof. We only prove the last item here. The other inequalities can be proved similarly using the same technique. Take $o_2 = o_3 = \top$ and let $\Delta\mathrm{Q}[\![o_1]\!] = \delta_1$. By Theorem 5, $o_1 \bullet\!\!\rightarrow\!\!\bullet\, (o_2 \|^{\exists} o_3) = o_1 \bullet\!\!\rightarrow\!\!\bullet\, (\top \|^{\exists} \top) = o_1 \bullet\!\!\rightarrow\!\!\bullet\, \top = o_1$. Therefore,

$$\Delta\mathrm{Q}[\![o_1 \bullet\!\!\rightarrow\!\!\bullet\, (o_2 \|^{\exists} o_3)]\!] = \delta_1. \tag{14}$$

On the other hand, by Theorem 5, $(o_1 \bullet\!\!\rightarrow\!\!\bullet\, o_2) \|^{\exists} (o_1 \bullet\!\!\rightarrow\!\!\bullet\, o_3) = (o_1 \bullet\!\!\rightarrow\!\!\bullet\, \top) \|^{\exists} (o_1 \bullet\!\!\rightarrow\!\!\bullet\, \top) = o_1 \|^{\exists} o_1$. Thus,

$$\Delta\mathrm{Q}[\![(o_1 \bullet\!\!\rightarrow\!\!\bullet\, o_2) \|^{\exists} (o_1 \bullet\!\!\rightarrow\!\!\bullet\, o_3)]\!] = \delta_1 + \delta_1 - \delta_1 \delta_1. \tag{15}$$

Equations (14) and (15) together imply $\delta_1 = 2\delta_1 - \delta_1^2 \Rightarrow \delta_1 = \mathbf{0} \vee \delta_1 = \mathbf{1} \Rightarrow o_1 = \bot \vee o_1 = \top$. The result follows because, for any other o_1 and $o_2 = o_3 = \top$, the two sides will not be equal. □

8 Related Work

ΔQSD has been used in practice by a small group of practitioners for a couple of decades now [20, 19, 6, 14, 5]. The first formalisation of ΔQSD was, however, done quite recently by Haeri et al. [11]. We use that formalisation as a foundation.

Teigen et al [14] use ΔQ to develop a novel model of WiFi performance that produces complete latency distributions. The model is validated by comparison with previous modeling work and real-world measurements. It would be very interesting to apply ΔQSD to an outcome description of the protocol to see if this can replicate the same results.

Elsewhere, Gajda [10] attempts to model latency distributions but allows operations that do not preserve total probability. In our context, these would lead to incorrect conclusions about failure probabilities.

Business Process Modelling and Notation (BPMN) [18] is a diagram scheme which is closely related to Outcome Diagrams (although with some details that are not considered relevant to ΔQSD). BPMN supports all ΔQSD operators except probabilistic choice. The closest operator is their "xor" gateway, which is essentially $\overset{[0.5]}{\rightleftharpoons}$. It is less expressive to the extent that it makes it impossible to consider systems such as the example in Section 2. Of the attempts for formalising BPMN, those of Wong and Gibbons [23, 24] are the most related to our work. Wong and Gibbons use the CSP process algebra for that purpose and further develop it to enable the specification of timing constraints on concurrent systems. Their developments allow mechanical verification of behavioural properties of BPMN diagrams using the FDR2 [15] refinement checker. Whilst Wong and Gibbons prove many interesting properties of their BPMN instances, they do not consider algebraic equivalences or algebraic structures for BPMN as we do in this work for ΔQSD. A less related BPMN formalisation work is that of El Hichami et al. [8],

which provides a denotational semantics based on the Max+ algebra as an execution model for BPMN. They list a handful of algebraic equivalences in Max+ only axiomatically. Nevertheless, El Hichami et al. make no attempt to study the equivalence of BPMN diagrams based on their Max+ semantics.

When it comes to timeliness analysis, an important advantage of outcome diagrams over BPMNs is Definition 3, which formally defines the timeliness analysis of outcome diagrams. Definition 3 is fundamental to the applicability of the model theory we employ in this paper (Section 4.3). We are not aware of any formally defined recipe for timeliness analysis of BPMNs. The two closest attempts that we could find are the following two: Friedenstab et al. [9] borrow constructs from Business Activity Monitoring [4] to augment BPMN with a graphical notation for describing certain timeliness matters. Likewise, Morales [16] informally describes how to transform BPMN diagrams to timed automata networks, suggesting qualitative analysis of timeliness.

Performance Evaluation Process Algebra (PEPA) [13] is an algebraic language for performance modelling of systems. PEPA is successful and well-published with a rich family of formalisations with various interesting theoretical properties. However, PEPA suffers from several shortcomings that make difficult to apply to real-world software systems. For example, PEPA does not model open or partially-specified systems; every detail of the system needs to be determined in advance. Since PEPA does not allow goals and objectives to be specified, it offers no assistance when comparing the predicted performance with the requirements. PEPA also suffers from state explosion, rapidly making it impractical, although more recent PEPA technology employs continuous approximations of the states, which contain some of the state explosion. This is similar to the use of IRVs in ΔQSD but rather *ad hoc* compared with the systematic use of ΔQs in ΔQSD. Less conservative alternatives to PEPA like SCEL [7] allow open systems but suffer from even more state explosion. CARMA [2] addresses a lot of the problems with PEPA, using a fluid approximation to manage the state explosion.

PerformERL [3] is an Erlang toolset, which focuses on monitoring the relationship between load repeatability and internal resource allocation. The authors advertise their toolset as an assistant for making early stage performance decisions, but it is unclear how it does this. Uunlike ΔQSD, monitoring (like testing) requires implementation of the system specification up to a certain level. The closer the implementation is to the full specification, the more reliable the monitoring will become, but the analysis is then no longer early-stage. Less accurate monitoring, on the other hand, is not reliable for decision making. The closest PerformERL gets to the work described in this paper is its lightweight theoretical work out of the monitoring overhead it imposes to the system under development.

Finally, Failure Modes Effects Analysis [1] (FMEA) considers how failures propagate through a system but, unlike ΔQSD, does not model delays. We are not aware of any formalisation of FMEA that can serve algebraic developments like those on failure in this paper.

9 Conclusion and Future Work

This paper lays down model-theoretic foundations for timeliness analysis à la ΔQSD. It establishes time as a resource that is consumed by outcomes. In doing so, it enables timeliness analysis *via* the study of quality attenuation, simultaneously capturing both delay and failure. With our focus being exclusively on timeliness, we discuss the algebraic structures that the ΔQSD operators form with outcome expressions (Theorems 1–4). We refute the formation of richer algebraic structures by the ΔQSD operators and outcome expressions (Remarks 2, 3, and 4). We consider the 15 distributivity results about the ΔQSD operators. We prove 3 (Theorem 6) and disprove 8 (Theorem 8) using the newly formalised technique developed in this paper called properisation (Theorem 7) and 4 using counterexamples (Theorem 9). We

also provide guidelines for studying the existence of potential distributivity (Section 7.1). Finally, we establish 14 important equivalences that have already been used in the practice of ΔQSD over the past few decades (Lemmas 2–3 and Theorem 5).

Our immediate future work is to study the algebraic properties of other resources à la ΔQSD, with the eventual goal of providing an algebraic categorisation of resources. A sound theoretical foundation is essential for the construction of robust tool support, which is, in turn, a prerequisite for wider application of the ΔQSD paradigm. Currently, there is a numerically-based tool prototype. However, to deal effectively with large complex systems, this needs to be made more symbolic. The aim is for the expressions to be simplified before calculation, and to be able to represent performance unknowns. Algebraic structures are essential for correctly manipulating and simplifying expressions. This work informs both ongoing practical work and tool development. Conversely, consideration of specific aspects of system design and operation will inform the most productive directions for the theoretical developments.

To conclude, this paper has introduced a number of important algebraic properties for ΔQSD outcome expressions. These properties have a highly practical application in the analysis of timeliness and resource consumption. For the first time, we have shown distributivity of the ΔQSD operators over probabilistic choice, and placed a set of 'folklore' equivalences (Theorem 5) that are in common usage for ΔQSD on a sound footing. These equivalences are essential for rapid recognition of infeasibility and for sound manipulation of outcome expressions to reduce computational complexity.

Acknowledgements

This research is funded by IOG, Singapore as a part of an ongoing project for incorporating performance as a first-class factor of the software development life cycle. When the routine proof technique did not work for distributivity, Andre Knispel (of IOG) suggested that we could utilise easier properties to obtain the disproofs using contrapositive reasoning. We would like to thank him for that suggestion.

References

[1] (1980): *MIL-STD-1629A – Procedures for performing a failure mode effect and criticality analysis.* Technical Report, United States Department of Defense.

[2] L. Bortolussi, R. De Nicola, V. Galpin, S. Gilmore, J. Hillston, D. Latella, M. Loreti & M. Massink (2015): *CARMA: Collective Adaptive Resource-sharing Markovian Agents*. In N. Bertrand & M. Tribastone, editors: *Proc. 13th W. Quant. Aspects of Prog. Lang. and Sys., EPTCS* 194, pp. 16–31, doi:10.4204/EPTCS.194.2. Available at `https://doi.org/10.4204/EPTCS.194.2`.

[3] W. Cazzola, F. Cesarini & L. Tansini (2022): *PerformERL: A Performance Testing Framework for Erlang*. *Distributed Comp.* 35(5), pp. 439–454, doi:10.1007/s00446-022-00429-7. Available at `https://doi.org/10.1007/s00446-022-00429-7`.

[4] C. Costello & O. Molloy (2008): *Towards a Semantic Framework for Business Activity Monitoring and Management*. In: *AAAI Spring Symposium: AI meets business rules and process management*, pp. 17–27.

[5] D. Coutts, N. Davies, M. Szamotulski & P. Thompson (2020): *Introduction to the design of the Data Diffusion and Networking for Cardano Shelley*. Technical Report, IOHK.

[6] N. Davies, P. Thompson, G. Young, J. Newton, B. Teigen & M. Olden (2021): *Measuring Network Impact on Application Outcomes Using Quality Attenuation*. In: *Measuring Network Quality for End-Users*, Internet Architecture Board.

[7] R. De Nicola, D. Latella, A. L. Lafuente, M. Loreti, A. Margheri, M. Massink, A. Morichetta, R. Pugliese, F. Tiezzi & A. Vandin (2015): *The SCEL Language: Design, Implementation, Verification*, pp. 3–71. Springer, doi:10.1007/978-3-319-16310-9_1. Available at https://doi.org/10.1007/978-3-319-16310-9_1.

[8] O. El Hichami, M. Naoum, M. Al Achhab, I. Berrada & B. E. El Mohajir (2015): *An Algebraic Method for Analysing Control Flow of BPMN Models*. *iJES* 3(3), pp. 20—26, doi:10.3991/ijes.v3i3.4862. Available at https://online-journals.org/index.php/i-jes/article/view/4862.

[9] J.-P. Friedenstab, C. Janiesch, M. Matzner & O. Muller (2012): *Extending BPMN for Business Activity Monitoring*. In: 45^{th} *HICSS*, pp. 4158–4167, doi:10.1109/HICSS.2012.276.

[10] M. J. Gajda (2020): *Curious Properties of Latency Distributions*. *CoRR* abs/2011.05219. Available at https://arxiv.org/abs/2011.05219.

[11] S. H. Haeri, P. Thompson, N. Davies, P. Van Roy, K. Hammond & J. Chapman (2022): *Mind Your Outcomes: The ΔQSD Paradigm for Quality-Centric Systems Development and Its Application to a Blockchain Case Study*. *Computers* 11(3), p. 45, doi:10.3390/computers11030045. Available at https://www.mdpi.com/2073-431X/11/3/45.

[12] S. H. Haeri, P. W. Thompson, P. Van Roy, M. Haveraaen, N. J. Davies, M. Barash & J. Chapman (2023): *On the Algebraic Properties of Timeliness*. Technical Report, IOG. Available at http://www.pnsol.com/public/Algebraic-Timeliness-TR.pdf.

[13] J. Hillston (1996): *A Compositional Approach to Performance Modelling*. Cambridge University Press.

[14] B. Ivar Teigen, N. Davies, K. Olav Ellefsen, T. Skeie & J. Torresen (2022): *Quantifying the Quality Attenuation of WiFi*. In S. Oteafy, E. Bulut & F. Tschorsch, editors: *IEEE* 47^{th} *LCN*, IEEE, pp. 189–197, doi:10.1109/LCN53696.2022.9843690.

[15] Formal Systems (Europe) Ltd (2012): *Failures-Divergence Refinement: FDR2 User Manual*. Available at https://www.cs.ox.ac.uk/projects/concurrency-tools/download/fdr2manual-2.94.pdf.

[16] L. E. M. Morales (2014): *Specifying BPMN Diagrams with Timed Automata: Proposal of Some Mapping Rules*. In: 9^{th} *CISTI*, pp. 1–6, doi:10.1109/CISTI.2014.6876897.

[17] Predictable Network Solutions Ltd (PNSol) (2022): Available at http://www.pnsol.com.

[18] K. J. Sherry (2012): *Business Process Modelling with BPMN: Modelling and Designing Business Processes Course Book using The Business Process Model and Notation Specification Version 2.0*. CreateSpace Independent Publishing Platform.

[19] P. Thompson (2022): *TR-452.2 Quality Attenuation Measurements using Active Test Protocols*. Technical Report, The Broadband Forum.

[20] P. Thompson & R. Hernadaz (2020): *Quality Attenuation Measurement Architecture and Requirements*. Technical Report TR-452.1, Broadband Forum. Available at https://www.broadband-forum.org/download/TR-452.1.pdf.

[21] K. S. Trivedi (2002): *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, 2 edition. Wiley, New York, NY, USA.

[22] P. Van Roy, N. Davies, P. Thompson & S. H. Haeri (2023): *ΔQSD: Designing Systems with Predictable Latency at High Load*. Tutorial, HiPEAC 2023 (Conf. High Performance Embedded Architecture and Compilation).

[23] P. Y. H. Wong & J. Gibbons (2011): *Formalisations and Applications of BPMN*. *SCP* 76(8), pp. 633–650, doi:https://doi.org/10.1016/j.scico.2009.09.010. Available at https://www.sciencedirect.com/science/article/pii/S0167642309001282.

[24] P. Y. H. Wong & J. Gibbons (2011): *Property Specifications for Workflow Modelling*. *SCP* 76(10), pp. 942–967, doi:https://doi.org/10.1016/j.scico.2010.09.007. Available at https://www.sciencedirect.com/science/article/pii/S0167642310001735.

On the Introduction of Guarded Lists in Bach: Expressiveness, Correctness, and Efficiency Issues

Manel Barkallah
Nadi Research Institute
Faculty of Computer Science
University of Namur
Namur, Belgium
manel.barkallah@unamur.be

Jean-Marie Jacquet
Nadi Research Institute
Faculty of Computer Science
University of Namur
Namur, Belgium
jean-marie.jacquet@unamur.be

Concurrency theory has received considerable attention, but mostly in the scope of synchronous process algebras such as CCS, CSP, and ACP. As another way of handling concurrency, data-based coordination languages aim to provide a clear separation between interaction and computation by synchronizing processes asynchronously by means of information being available or not on a shared space. Although these languages enjoy interesting properties, verifying program correctness remains challenging. Some works, such as Anemone, have introduced facilities, including animations and model checking of temporal logic formulae, to better grasp system modelling. However, model checking is known to raise performance issues due to the state space explosion problem. In this paper, we propose a guarded list construct as a solution to address this problem. We establish that the guarded list construct increases performance while strictly enriching the expressiveness of data-based coordination languages. Furthermore, we introduce a notion of refinement to introduce the guarded list construct in a correctness-preserving manner.

1 Introduction

Concurrency theory has been the attention of a considerable effort these last decades. However most of the effort has been devoted to algebra based on synchronous communication, such as CCS [31], CSP [21] and ACP [3]. Another path of research has been initiated by Gelernter and Carriero, who advocated in [18] that a clear separation between the interactional and the computational aspects of software components has to take place in order to build interactive distributed systems. Their claim has been supported by the design of a model, Linda [9], originally presented as a set of inter-agent communication primitives which may be added to almost any programming language. Besides process creation, this set includes primitives for adding, deleting, and testing the presence/absence of data in a shared dataspace. In doing so they proposed a new form of synchronization of processes, occurring asynchronously, through the availability or absence of pieces of information on a shared space.

A number of other models, now referred to as coordination models, have been proposed afterwards. However, although many pieces of work have been devoted to the proposal of new languages, semantics and implementations, few articles have addressed the concerns of practically constructing programs in coordination languages, in particular in checking that what is described by programs actually corresponds to what has to be modelled.

Based on previous results [6, 7, 11, 12, 13, 14, 24, 26, 27, 28, 29], we have introduced in [22] a workbench Scan to reason on programs written in Bach, a Linda-like dialect developed by the authors. It has been refined in [23] to cope with relations, processes and multiple scenes. The resulting workbench is named Anemone. In both cases, one of our goals was to allow the user to check properties by model

Preliminary Report. Final version to appear in:
ICE 2023

Figure 1: Rush Hour Problem. On the left part, the game as illustrated at https://www.michaelfogleman.com/rush. On the right part, the game modeled as a grid of 6×6, with cars and trucks depicted as rectangles of different colors.

checking temporal logic formulae and by producing traces that can be replayed as evidences of the establishment of the formulae. However, as well-known in model checking, this goal raises performance issues related to the state space explosion. In particular, letting animation-related primitives interleave in many ways duplicates research paths during model checking, with considerable performance problems to check that formulae are established. To address this problem, we introduce in this paper a guarded list construct and establish that it yields an increase in performance while strictly enriching the expressiveness of Bach.

The rest of the paper is organized as follows. Section 2 presents the reference Linda-like language Bach employed by Scan and Anemone. Section 3 introduces the guarded list construction as well as the refinement relation. It is proved to increase the expressiveness of the Bach language in Section 4 while the gain of efficiency in model-checking is established in Section 5. Finally, Section 6 compares our work with related work and Section 7 sums up the paper and sketches future work.

It is worth observing that, as duly compared in Section 6, introducing an atomic construct is not new. However, our contribution is (i) to introduce a construct tailored to coordination languages, (ii) to establish that it yields a gain of performance in model checking and also an increase of expressiveness, and finally (iii) to identify refinement-based criteria so as to guide the programmer to introduce the guarded list construct in a correctness-preserving manner.

To make the article more concrete, we shall use the running example of [23], namely a solution to the rush hour puzzle. This game, illustrated in Figure 1, consists in moving cars and trucks on a 6×6 grid, according to their direction, such that the red car can exit. It can be formulated as a coordination problem by considering cars and trucks as autonomous agents which have to coordinate on the basis of free places.

2 The Anim-Bach language

2.1 Definition of data

Following Linda, the Bach language [14, 25] uses four primitives for manipulating pieces of information: *tell* to put a piece of information on a shared space, *ask* to check its presence, *nask* to check its absence and *get* to check its presence and remove one occurrence. In its simplest version, named BachT, pieces

of information consist of atomic tokens and the shared space, called the store, amounts to a multiset of tokens. Although in principle such a framework is sufficient to code many applications, it is however too elementary in practice to code them easily. To that end, we introduce more structured pieces of information which may employ sets defined as in

```
eset RCInt = { 1 , 2 , 3 , 4 , 5 , 6 }.
```

in which the set *RCInt* is defined as the set containing the elements 1 to 6. In addition to sets, maps can be defined between them as functions that take zero or more arguments. In practice, mapping equations are used as rewriting rules, from left to right in the aim of progressively reducing a complex map expression into a set element.

As an example of a map, assuming a grid of 6 by 6 featuring the rush hour problem as in [23] and assuming that trucks in this game take three cells and are identified by the upper and left-most cell they occupy, the operation `down_truck` determines the cell to be taken by a truck moving down:

```
map down_truck : RCInt -> RCInt.
eqn down_truck(1) = 4. down_truck(2) = 5. down_truck(3) = 6.
```

Note from this example that mappings may be partially defined, with the responsibility put on the programmer to use them only when defined.

Structured pieces of information to be placed on the store consist of flat tokens as well as expressions of the form $f(a_1,\cdots,a_n)$ where f is a functor and a_1, ..., a_n are set elements or structured pieces of information. As an example, in the rush hour example, it is convenient to represent the free places of the game as pieces of information of the form `free(i,j)` with i a row and j a column.

The set of structured pieces of information is subsequently denoted by $\mathscr{I}$. For short, si-term is used later to denote a structured piece of information. Mapping definitions induce a rewriting relation that we shall subsequently denote by $\rightsquigarrow$, that rewrites si-terms to final si-terms, namely si-terms that cannot be reduced further.

2.2 Primitives

The primitives consist of the `tell`, `ask`, `nask` and `get` primitives already introduced, which take as arguments elements of $\mathscr{I}$. A series of graphical primitives are added to them. They aim at animating the executions. They include `draw`, `move_to`, `place_at`, `hide`, `show` primitives, to cite only a few. The key point for this paper is that they always succeed and do not interfere with the shared space. For the rest of the paper, we shall assume a set *GPrim* of graphical primitives and will take primitives from it. The coordinated Bach language enriched by graphical primitives is subsequently referred to as Anim-Bach.

The execution of primitives is formalized by the transition steps of Figure 2. Configurations are taken there as pairs of instructions, for the moment reduced to simple primitives, coupled to the contents of the shared space. Following the constraint-like setting of Bach in which the Linda primitives have been rephrased, the shared space is renamed as *store* and is formally defined as a multiset of si-terms. As a result, rule (T) states that the execution of the $tell(t)$ primitive amounts to enriching the store by an occurrence of t. The E symbol is used in this rule as well as in other rules to denote a terminated computation. Similarly, rules (A) and (G) respectively state that the $ask(t)$ and $get(t)$ primitives check whether t is present on the store with the latter removing one occurrence. Dually, as expressed in rule (N), the primitive $nask(t)$ tests whether t is absent from the store. Finally, rule (Gr) expresses that any graphical primitive succeeds without modifying the store.

$$(\mathbf{T}) \quad \frac{t \rightsquigarrow u}{\langle\, tell(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{u\} \,\rangle} \qquad (\mathbf{N}) \quad \frac{t \rightsquigarrow u, u \notin \sigma}{\langle\, nask(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

$$(\mathbf{A}) \quad \frac{t \rightsquigarrow u}{\langle\, ask(t) \mid \sigma \cup \{u\} \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{u\} \,\rangle} \qquad (\mathbf{Gr}) \quad \frac{p \in GPrim}{\langle\, p \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

$$(\mathbf{G}) \quad \frac{t \rightsquigarrow u}{\langle\, get(t) \mid \sigma \cup \{u\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

Figure 2: Transition rules for the primitives

2.3 Agents

Primitives can be composed to form more complex agents by using traditional composition operators from concurrency theory: sequential composition, parallel composition and non-deterministic choice. Another mechanism is added in Anim-Bach: conditional statements of the form $c \rightarrow s_1 \diamond s_2$, which computes s_1 if c evaluates to true or s_2 otherwise. As a shorthand, $c \rightarrow s_1$ is used to compute s_1 when c evaluates to true. Conditions of type c are obtained from elementary ones, thanks to the classical and, or and negation operators, denoted respectively by &, | and !. Elementary conditions are obtained by relating set elements or mappings on them by equalities (denoted $=$) or inequalities (denoted $=$, $<$, $<=$, $>$, $>=$).

Procedures are defined similarly to mappings through the `proc` keyword by associating an agent with a procedure name. As in classical concurrency theory, it is assumed that the defining agents are guarded, in the sense that any call to a procedure is preceded by the execution of a primitive or can be rewritten in such a form.

As an example, the behavior of a vertical truck in the rush hour puzzle can be modelled by the following code:

```
proc VerticalTruck(r: RCInt, c: RCInt) =
  ( (r>1 & r<5) -> ( get(free(pred(r),c)); tell(free(succ(succ(r)),c);
                     VerticalTruck(pred(r),c) )
  +
  ( (r<4) -> ( get(free(down_truck(r),c)); tell(free(r,c));
               VerticalTruck(succ(r),c)) ).
```

To understand it, remember that a truck is identified by the upper and left-most cell it occupies. The parameters of the `VerticalTruck` procedure are precisely the row number and the line number of this cell. Given that a vertical truck can move one cell up or one cell down, the procedure offers two alternatives through the "+" operator. The first one corresponds to a truck moving one cell up. To make this move realistic, the row r occupied by the truck should be strictly greater than one. Otherwise, the truck is already on the first row (like the yellow truck of Figure 1) and cannot move up. Moreover, as we shall see in a few seconds, the row r should also be strictly smaller than 5. Assuming the two conditions hold ($r > 1 \& r < 5$) moving a truck one cell up proceeds in three steps. First we need to make sure that the cell up is free. This is obtained by getting the si-term $free(pred(r),c))$ by means of the execution of the `get(free(pred(r),c))` primitive. Note that $pred(r)$ is actualy coded by a map as being $r-1$. Second

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

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

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

$$(\mathbf{Co}) \quad \frac{\models C,\ \langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle C \to A \diamond B \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle \\ \langle !C \to B \diamond A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle\end{array}}$$

$$(\mathbf{Pc}) \quad \frac{P(\overline{x}) = A,\ \langle A[\overline{x}/\overline{u} \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\langle P(\overline{u}) \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}$$

Figure 3: Transition rules for the operators

the cell liberated by moving the truck one cell up is to be declared free. This is obtained by telling the corresponding *free* si-term on the store, namely by executing `tell(free(succ(succ(r)),c)`. Note that $succ(r)$ is coded as $r+1$ by a map, which is why r needs to be smaller than 5. Third the truck procedure has to be called recursively with $pred(r)$ and c as new coordinates for the upper and left-most cell it occupies.

The behavior of the alternative movement in which the truck goes down by one cell is similar. As r is assumed to be in set $RCInt = \{1,\cdots,6\}$ and we do not perform a *pred* operation there is no need to check that r is greater or equal to 1. However to get the cell down we need to check that r is strictly less than 4.

The operational semantics of complex agents is defined through the transition rules of Figure 3. They are quite classical. Rules (S), (P) and (C) provide the usual semantics for sequential, parallel and choice compositions. As expected, rule (Co) specifies that the conditional instruction $C \to A \diamond B$ behaves as A if condition C can be evaluated to true and as B otherwise. Note that the notation $\models C$ is used to denote the fact that C evaluates to true. Finally, rule (Pc) makes procedure call $P(\overline{u})$ behave as the agent A defining procedure P with the formal arguments $\overline{x}$ replaced by the actual ones $\overline{u}$.

In these rules, it is worth noting that we assume agents of the form $(E;A)$, $(E \,||\, A)$ and $(A \,||\, E)$ to be rewritten as A.

2.4 A fragment of temporal logic

Linear temporal logic is widely used to reason on dynamic systems. The Scan and Anemone workbenches use a fragment of PLTL [16].

As usual, the logic employed relies on propositional state formulae. In the coordination context, these formulae are to be verified on the current contents of the store. Consequently, given a structured piece of information t, the notation $\#t$ is introduced to denote the number of occurrences of t on the store and basic propositional formulae are defined as equalities or inequalities combining algebraic expressions involving integers and number of occurrences of structured pieces of information. An example of such a basic formulae is $\#free(1,1) = 1$ which states that the cell of coordinates $(1,1)$ is free.

Propositional state formulae are built from these basic formulae by using the classical propositional connectors. On the point of notations, given a store σ and a propositional state formulae PF, we shall write $\sigma \models PF$ to indicate that PF is established on store σ.

The fragment of temporal logic used in Scan and Anemone is then defined from these propositional state formulae by the following grammar :

$$TF ::= PF \,|\, Next\ TF \,|\, PF\ Until\ TF$$

where PF is a propositional formula. A classical use, on which we shall focus in this paper, is to determine whether a propositional state formulae can be reached at some state. As an example, coming back to the rush hour problem, if the red car indicates that it leaves the grid by placing *out* on the store, a solution to the rush problem is obtained by verifying the formula

$$true\ Until\ (\#out = 1)$$

which we shall subsequently abbreviate as $Reach(\#out = 1)$.

The algorithm used in Scan and Anemone to establish reach properties basically consists of a breadth-first search on the state space engendered by an agent starting from the empty store. During this search, for each newly created state, a test is made to check whether the considered reach property holds.

Such an elementary algorithm works well for simple problems. However it becomes difficult to use when more complex problems are tackled. One of the reasons comes from the fact that states are duplicated many times by interleaving. Consider for instance the code for the `VerticalTruck` procedure introduced above. With primitives to animate its execution and colors introduced for visualization purposes, its more complete code is as follows:

```
proc VerticalTruck(r: RCInt, c: RCInt, p: Colors) =
        ( (r>1 & r<5) -> ( get(free(pred(r),c));
                           moveTruck(pred(r),c,p);
                           tell(free(succ(succ(r)),c));
                           VerticalTruck(pred(r),c,p) ))
        +
        ( (r<4) -> ( get(free(down_truck(r),c));
                     moveTruck(succ(r),c,p);
                     tell(free(r,c));
                     VerticalTruck(succ(r),c,p) )).
```

Consider now two vertical trucks in parallel and for illustration the first three statements: `get(free(pred(r),c))`, `moveTruck(pred(r),c,p)` `tell(free(succ(succ(r)),c))`. Interleaving them in the two parallel instances of `VerticalTruck` is of no interest for checking whether *out* has been produced since what really matters is the state resulting after the three steps. Hence, provided the first get primitive succeeds, the two other primitives may be executed in a row. This observation leads us to introduce so-called guarded lists of primitives.

3 A guarded list construct

A *guarded list* of primitives is a construct of the form $[p \rightarrow p_1, \cdots, p_n]$ where p, p_1, ..., p_n are primitives, with the list p_1, ..., p_n being possibly empty. In that latter case, we shall write $[p]$ for simplicity of the notations.

Basically, a guarded list of primitives is a list of primitives containing at least one primitive. The reason for writing guarded lists with an arrow and for calling it guarded comes from the fact that, provided the first primitive can be successfully executed, all the others are executed immediately after without rollback in case of failure. It is of course the responsibility of the programmers to guarantee that in

$$(\mathbf{Le}) \qquad \langle\, [] \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle$$

$$(\mathbf{Ln}) \quad \frac{\langle\, p \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \tau \,\rangle,\ \langle\, L \mid \tau \,\rangle \longrightarrow^* \langle\, E \mid \phi \,\rangle}{\langle\, [p|L] \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \phi \,\rangle}$$

$$(\mathbf{GL}) \quad \frac{\langle\, p \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \tau \,\rangle,\ \langle\, L \mid \tau \,\rangle \longrightarrow^* \langle\, E \mid \phi \,\rangle}{\langle\, [p \rightarrow L] \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \phi \,\rangle}$$

Figure 4: Transition rules for guarded lists

case the first primitive can be successfully evaluated the remaining primitives can also be successfully executed. Note that this is obviously the case for tell primitives and the graphical primitives which always succeed regardless of the current content of the store. Note also that we shall subsequently identify criteria to introduce guarded lists while preserving correctness.

It is worth observing that guarded lists are atomic constructs which makes them different from conditional statements. In two words, the execution of $[p \rightarrow p_1, \cdots, p_n]$ is as follows. First the store is locked and the execution of p is tested. If it fails then no modification is performed on the store and the store is released. Otherwise not only p is executed but also after $p_1, \ldots, p_n$ in a row. After that the store is released. In contrast, the execution of the conditional statement $c \rightarrow s_1 \diamond s_2$ amounts to check c, which does not require to lock the store since conditions are built on comparing si-terms and not their presence or absence on the store. If c is evaluated to true then s_1 is executed, which means that one step of s_1 is done if this is possible. If c is evaluated to false then one step of s_2 is attempted.

The operational semantics of guarded lists is defined by rules (Le), (Ln) and (GL) of Figure 4. The first two rules define the semantics of lists of primitives, as being successively executed. Rule (Le) concerns the empty list of primitives $[]$ while rule (Ln) inductively specifies that of a non-empty list $[p|L]$ with p the first primitive and L is the list of the other primitives[1]. Rule (GL) then states that the guarded list $[p \rightarrow L]$ can do a computation step from the store σ to ϕ provided the primitive p can do a step changing the store σ to τ and provided the list of primitives L can change τ to ϕ.

Of course, introducing guarded lists as an atomic construct reduces the interleaving possibilities between parallel processes. This is in fact what we want to achieve to get speed ups in the model checking phase. However from a programming point of view, one needs to guarantee that computations are kept in some way. This is the purpose of the introduction of the histories and of their contractions.

Definition 1

1. *Define the set of computational histories (or histories for short) Shist as the set* $Sstore^{\omega} \cup Sstore^{*}.\{\delta^{+}, \delta^{-}\}$ *where Sstore denotes the set of stores (namely of finite multisets of final si-terms), the* $*$ *and* ω *symbols are used to respectively denote finite and infinite repetitions and where* δ^{+} *and* δ^{-} *are used as ending marks respectively denoting successful and failing computations.*
2. *A history* h_c *is a contraction of an history h if it can be obtained from the latter by removing a finite number (possibly 0) elements of it, except the terminating marks* δ^{+} *and* δ^{-}*. This is subsequently denoted by* $h_c \preceq h$*.*

[1] These list notations $[]$ and $[p|L]$ come from the logic programming way of handling lists.

3. *Given a contraction $h_c = \sigma_0.\cdots.\sigma_n.\delta$ (resp. $h_c = \sigma_0.\cdots.\sigma_n.\cdots$) of an history h, there are thus sequences of stores, $\overline{\sigma_0}$, ..., $\overline{\sigma_n}$ such that $h = \overline{\sigma_0}.\sigma_0.\cdots.\overline{\sigma_n}.\sigma_n.\delta$ (resp. $h = \overline{\sigma_0}.\sigma_0.\cdots.\overline{\sigma_n}.\sigma_n.\cdots$). For any logic formula F, the history h_c is said to be F-preserving iff, for any i and for any store τ of $\overline{\sigma_i}$, one has $\tau \models F$ iff $\sigma_i \models F$. This is subsequently denoted as $h_c \ll_F h$.*

Contractions and F-preserving contractions can be lifted in an obvious way to sets of histories.

Definition 2 *A set S_c of histories is a contraction (resp. a F-preserving contraction) of a set S of histories if any history of S_c is the contraction (resp. a F-preserving contraction) of a history of S. By lifting notations on histories, this is subsequently denoted by $S_c \preceq S$ (resp. $S_c \ll_F S$).*

We can now define the history-based operational semantics as the one delivering all the computational histories. To make it general, we shall define it on any contents of the initial store.

Definition 3 *Define the language $\mathscr{L}_g$ as the* Anim-Bach *language with the guard list construct.*

Definition 4 *Define the operational semantics $\mathscr{O}_h : \mathscr{L}_g \to \mathscr{P}(Shist)$ as the following function. For any agent A and any store τ*

$$\begin{aligned}\mathscr{O}_h(A)(\tau) = \\ &\{\sigma_0.\cdots.\sigma_n.\delta^+ : \langle A \mid \sigma_0\rangle \longrightarrow \cdots \longrightarrow \langle E \mid \sigma_n\rangle, \sigma_0 = \tau, n \geq 0\} \\ \cup &\{\sigma_0.\cdots.\sigma_n.\delta^- : \langle A \mid \sigma_0\rangle \longrightarrow \cdots \longrightarrow \langle A_n \mid \sigma_n\rangle \not\longrightarrow, \sigma_0 = \tau, A_n \neq E, n \geq 0\} \\ \cup &\{\sigma_0.\cdots.\sigma_n.\cdots : \langle A \mid \sigma_0\rangle \longrightarrow \cdots \longrightarrow \langle A_n \mid \sigma_n\rangle \longrightarrow \cdots, \sigma_0 = \tau, \forall n \geq 0 : A_n \neq E\}\end{aligned}$$

We are now in a position to define the refinement of agents.

Definition 5 *Agent A is said to refine agent B iff $\mathscr{O}_h(A)(\tau) \preceq \mathscr{O}_h(B)(\tau)$, for any store τ.*

The following proposition is a direct consequence of the above definitions. Its interest is to establish contractions and F-preserving properties from a syntactic characterization.

Proposition 1

1. *If $p_1, \cdots, p_n$ are tell primitives or graphical primitives then for any primitive p, the guarded list $GL = [p \to p_1, \cdots, p_n]$ refines the sequential composition $SC = p; p_1; \cdots; p_n$. As a result, any reachable property proved on the stores generated by the execution of GL from a given store τ is also established on the stores generated by the execution of SC from τ.*
2. *Assuming additionally that the arguments of the tell primitives of $p_1, \cdots, p_n$ are distinct from the si-terms appearing in the reachable formulae F, then GL is also a F-preserving contraction of SC. It results that F is established on the stores resulting from the execution of SC from any store τ iff it is established on the stores resulting from the execution of GL from τ.*

For the study of expressiveness, it will be useful to turn to a simpler semantics focusing on the resulting stores of finite computations. Such a semantics is defined as follows.

Definition 6 *Define the* operational semantics *$\mathscr{O}_f : \mathscr{L}_g \to \mathscr{P}(Sstore \times \{\delta^+, \delta^-\})$ as the following function: for any agent $A \in \mathscr{L}_g$*

$$\begin{aligned}\mathscr{O}_f(A) \quad = \quad &\{(\sigma, \delta^+) : \langle A \mid \emptyset\rangle \to^* \langle E \mid \sigma\rangle\} \\ &\cup \\ &\{(\sigma, \delta^-) : \langle A \mid \emptyset\rangle \to^* \langle B \mid \sigma\rangle \not\to, B \neq E\}\end{aligned}$$

It is immediate to verify that, for any agent A, the semantics $\mathscr{O}_f(A)$ is obtained by considering the final stores of the finite histories of $\mathscr{O}_h(A)(\emptyset)$.

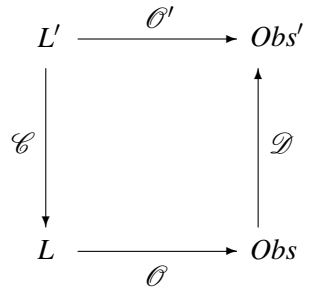

Figure 5: Basic embedding.

4 Expressiveness

Although it is interesting to bring efficiency during model checking, the guarded list construct also brings an increase of expressiveness. This is evidenced in this section by using the notion of modular embedding introduced in [5]. As pointed out there, from a computational point of view, all "reasonable" sequential programming languages are equivalent, as they express the same class of functions. Still it is common practice to speak about the "power" of a language on the basis of the expressibility or non-expressibility of programming constructs. In general, a sequential language L is considered to be more expressive than another sequential language L' if the constructs of L' can be translated in L without requiring a "global reorganization of the program" [17], that is, in a compositional way. Of course the translation must preserve the meaning, at least in the weak sense of preserving termination.

When considering concurrent languages, the notion of termination must be reconsidered as each possible computation represents a possible different evolution of a system of interacting processes. Moreover *deadlock* represents an additional case of termination. We shall consequently rely on the operational semantics $\mathscr{O}_f$ of Definition 6, focused on the final store of finite computations together with the termination mark.

The basic definition of embedding, given by Shapiro [34] is the following. Consider two languages L and L'. Moreover assume we are given the semantics mappings $\mathscr{O} : L \to Obs$ and $\mathscr{O}' : L' \to Obs'$, where Obs and Obs' are some suitable domains. Then L can *embed* L' if there exists a mapping $\mathscr{C}$ (*coder*) from the statements of L' to the statements of L, and a mapping $\mathscr{D}$ (*decoder*) from Obs to Obs', such that the diagram of Figure 5 commutes, namely such that for every statement $A \in L'$: $\mathscr{D}(\mathscr{O}(\mathscr{C}(A))) = \mathscr{O}'(A)$.

The basic notion of embedding is too weak since, for instance, the above equation is satisfied by any pair of Turing-complete languages. De Boer and Palamidessi hence proposed in [5] to add three constraints on the coder $\mathscr{C}$ and on the decoder $\mathscr{D}$ in order to obtain a notion of *modular* embedding usable for concurrent languages:

1. $\mathscr{D}$ should be defined in an element-wise way with respect to $\mathscr{O}$:

$$\forall X \in Obs : \ \mathscr{D}(X) = \{\mathscr{D}_{el}(x) \mid x \in X\} \qquad (P_1)$$

   for some appropriate mapping $\mathscr{D}_{el}$;

2. the coder $\mathscr{C}$ should be defined in a compositional way with respect to the sequential, parallel and

choice operators[2]:

$$\begin{array}{c} \mathscr{C}(A\ ;\ B) = \mathscr{C}(A)\ ;\ \mathscr{C}(B) \\ \mathscr{C}(A\ ||\ B) = \mathscr{C}(A)\ ||\ \mathscr{C}(B) \\ \mathscr{C}(A\ +\ B) = \mathscr{C}(A)\ +\ \mathscr{C}(B) \end{array} \qquad (P_2)$$

3. the embedding should preserve the behavior of the original processes with respect to deadlock, failure and success (*termination invariance*):

$$\forall X \in Obs, \forall x \in X :\ tm'(\mathscr{D}_{el}(x)) = tm(x) \qquad (P_3)$$

where tm and tm' extract the information on termination from the observables of L and L', respectively.

An embedding is then called *modular* if it satisfies properties P_1, P_2, and P_3.

The existence of a modular embedding from L' into L is denoted as $L' \leq L$. It is easy to see that $\leq$ is a pre-order relation. Moreover if $L' \subseteq L$ then $L' \leq L$ that is, any language embeds all its sublanguages. This property descends immediately from the definition of embedding, by setting $\mathscr{C}$ and $\mathscr{D}$ equal to the identity function.

Let us now compare the Anim-Bach language with guarded lists with the Anim-Bach language without guarded lists. As introduced before, the former is denoted by $\mathscr{L}_g$. The latter will be denoted by $\mathscr{L}_r$. Following [7], we shall also test three sublanguages composed (i) of the ask, tell primitives, (ii) of the ask, tell, get primitives and (iii) of the ask, tell, get, nask primitives. These sublanguages will be denoted by specifying the primitives between parentheses, as in $\mathscr{L}_g(ask, tell)$. Moreover, to focus on the core features, we shall discard conditional statements and procedures, which are essentially introduced for the ease of coding applications.

By language inclusion, a first obvious result is that the Anim-Bach sublanguages with guarded lists embed their counterparts without guarded lists.

Proposition 2 *For any subset $\mathscr{X}$ of primitives, one has $\mathscr{L}_r(\mathscr{X}) \leq \mathscr{L}_g(\mathscr{X})$.*

The converse relations do not hold. Intuitively, this is due to the fact that, in contrast to $\mathscr{L}_r$, the languages $\mathscr{L}_g$ have the possibility of *atomically* testing the simultaneous presence of two si-terms on the store. The formal proof requires of course a deeper treatment. It turns out however that the techniques employed in [7] can be adapted to guarded lists. One of them, which results from classical concurrency theory, is that any agent can be reformulated in a so-called normal form.

Definition 7 *Agents (of $\mathscr{L}_g$) in normal forms are agents of $\mathscr{L}_g$ which obey the following grammar, where N is an agent in normal form, p is a primitive (either graphical or store-related) or a guarded list of primitives and A denotes an arbitrary (non restricted) agent*

$$N \quad ::= \quad p \mid p\ ;\ A \mid N\ +\ N.$$

Proposition 3 *For any agent A of $\mathscr{L}_g$, there is an agent N of $\mathscr{L}_g$ in normal form which has the same derivation sequences as A.*

[2] Actually, this is not required for the sequential operator in [5] since it does not occur in that work.

Proof. Indeed, it is possible to associate to any agent A an agent $\tau(A)$ in normal form by using the following translation defined inductively on the structure of A:

$$\begin{aligned}
\tau(p) &= p \\
\tau(X;Y) &= \tau(X);Y \\
\tau(X+Y) &= \tau(X)+\tau(Y) \\
\tau(X \parallel Y) &= \tau(X)\mathbb{L}Y+\tau(Y)\mathbb{L}X
\end{aligned}$$

$$\begin{aligned}
p\mathbb{L}Z &= p;Z \\
(p;A)\mathbb{L}Z &= p;(A \parallel Z) \\
(N_1+N_2)\mathbb{L}Z &= N_1\mathbb{L}Z+N_2\mathbb{L}Z
\end{aligned}$$

It is easy to verify that, for any agent A, the agent $\tau(A)$ is in normal form. Moreover, it is straightforward to verify that A and $\tau(A)$ share the same derivation sequences. □

We are now in a position to establish that $\mathscr{L}_g(ask,tell)$ cannot be embedded in $\mathscr{L}_r(ask,tell)$.

Proposition 4 $\mathscr{L}_g(ask,tell) \not\leq \mathscr{L}_r(ask,tell)$

Proof. Let us proceed by contradiction and assume the existence of a coder $\mathscr{C}$ and a decoder $\mathscr{D}$. The proof is composed of three main steps.

STEP 1: on the coding of $tell(a)$ and $tell(b)$. Let a, b be two distinct si-terms. Since $\mathscr{O}_f([tell(a)]) = \{(\{a\},\delta^+)\}$, any computation of $\mathscr{C}([tell(a)])$ starting in the empty store succeeds by property P_3. Let

$$\langle \mathscr{C}([tell(a)]) \mid \emptyset\rangle \longrightarrow \cdots \longrightarrow \langle E \mid \{a_1,\cdots,a_m\}\rangle$$

be one computation of $\mathscr{C}([tell(a)])$. Similarly, any computation of $\mathscr{C}([tell(b)])$ starting on the empty store succeeds. Let

$$\langle \mathscr{C}([tell(b)]) \mid \emptyset\rangle \longrightarrow \cdots \longrightarrow \langle E \mid \{b_1,\cdots,b_n\}\rangle$$

be one computation of $\mathscr{C}([tell(b)])$. Note that, as we only consider ask and tell primitives, this computations can be reproduced on any store τ. We thus have also that

$$\langle \mathscr{C}([tell(b)]) \mid \tau\rangle \longrightarrow \cdots \longrightarrow \langle E \mid \tau \cup \{b_1,\cdots,b_n\}\rangle$$

In particular, as $\mathscr{C}([tell(a)];[tell(b)]) = \mathscr{C}([tell(a)]);\mathscr{C}([tell(b)])$, we have that

$$\begin{aligned}
\langle \mathscr{C}([tell(a)];[tell(b)]) \mid \emptyset\rangle &\longrightarrow \cdots \\
&\longrightarrow \langle \mathscr{C}([tell(b)]) \mid \{a_1,\cdots,a_m\}\rangle \longrightarrow \cdots \\
&\longrightarrow \langle E \mid \{a_1,\cdots,a_m,b_1,\cdots,b_n\}\rangle
\end{aligned}$$

STEP 2: coding of an auxiliary statement AB. Consider now $AB = [ask(a) \rightarrow ask(b)]$. Obviously, as it requires a to be present, the execution of AB on the empty store cannot do any step and thus $\mathscr{O}_f(AB) = \{(\emptyset,\delta^-)\}$. Let us now turn to its coding $\mathscr{C}(AB)$. By Proposition 3, it can be regarded in its normal form. As it is in $\mathscr{L}_r(tell,ask)$, its more general form is as follows

$$tell(t_1)\ ;\ A_1+\cdots+tell(t_p)\ ;\ A_p+ask(u_1)\ ;\ B_1+\cdots+ask(u_q)\ ;\ B_q+gp_1\ ;\ C_1+\cdots+gp_r\ ;\ C_r$$

where $gp_1, \ldots, gp_r$ are graphical primitives. Let us first establish that there is no alternative guarded by a $tell(t_i)$ operation. Indeed, if this was the case, then

$$D = \langle \mathscr{C}(AB) \mid \emptyset \rangle \longrightarrow \langle A_i \mid \{t_i\} \rangle$$

would be a valid computation prefix of $\mathscr{C}(AB)$. As $\mathscr{O}_f(AB) = \{(\emptyset, \delta^-)\}$, this prefix should deadlock afterwards. However, as $\mathscr{C}(AB + [tell(a)]) = \mathscr{C}(AB) + \mathscr{C}([tell(a)])$, the computation step D is also a valid computation prefix of $\mathscr{C}(AB + [tell(a)])$. Hence, $\mathscr{C}(AB + [tell(a)])$ admits a failing computation which, by property P_3, contradicts the fact that $\mathscr{O}_f(AB + [tell(a)]) = \{(\{a\}, \delta^+)\}$. The proof of the absence of an alternative guarded by a graphical primitive gp_i proceeds similarly.

Let us now establish that none of the u_i's belong to $\{a_1, \cdots, a_m\} \cup \{b_1, \cdots, b_n\}$. Indeed, if $u_j \in \{a_1, \cdots, a_m\}$ for some $j \in \{1, \cdots, q\}$, then, as $\mathscr{C}([tell(a)] \,;\, AB) = \mathscr{C}([tell(a)]) \,;\, \mathscr{C}(AB)$, the derivation

$$\begin{aligned} D' = \langle \mathscr{C}([tell(a)] \,;\, AB) \mid \emptyset \rangle \longrightarrow \cdots \longrightarrow & \langle \mathscr{C}(AB) \mid \{a_1, \cdots, a_m\} \rangle \\ & \longrightarrow \langle B_j \mid \{a_1, \cdots, a_m\} \rangle \end{aligned}$$

is a valid computation prefix of $\mathscr{C}([tell(a)] \,;\, AB)$. However, by applying rule (T),

$$\langle [tell(a)] \,;\, AB \mid \emptyset \rangle \longrightarrow \langle AB \mid \{a\} \rangle \not\longrightarrow$$

By Property P_3, it follows that D' can only be continued by failing suffixes. However, thanks to the fact that $\mathscr{C}([tell(a)] \,;\, (AB + [ask(a)])) = \mathscr{C}([tell(a)]) \,;\, (\mathscr{C}(AB) + \mathscr{C}([ask(a)]))$ the prefix D' induces the following computation prefix D'' for $\mathscr{C}([tell(a)] \,;\, (AB + [ask(a)]))$

$$\begin{aligned} D'' = \langle \mathscr{C}([tell(a)] \,;\, (AB + [ask(a)])) \mid \emptyset \rangle \longrightarrow \cdots \\ \longrightarrow \langle \mathscr{C}(AB) + \mathscr{C}([ask(a)]) \mid \{a_1, \cdots, a_m\} \rangle \\ \longrightarrow \langle B_j \mid \{a_1, \cdots, a_m\} \rangle. \end{aligned}$$

which can only be continued by failing suffixes whereas $[tell(a)] \,;\, (AB + [ask(a)])$ only admits a successful computation.

The proof proceeds similarly in the case $u_j \in \{b_1, \cdots, b_n\}$ for some $j \in \{1, \cdots, q\}$ by then considering $[tell(b)] \,;\, AB$ and $[tell(b)] \,;\, (AB + [ask(b)])$.

STEP 3: combining the first two steps to produce a contradiction. The u_i's are thus forced not to belong to $\{a_1, \cdots, a_m\} \cup \{b_1, \cdots, b_n\}$. However, this induces a contradiction. To that end, let us first observe that $\mathscr{C}(AB)$ cannot do any step on the store $\{a_1, \cdots, a_m, b_1, \cdots, b_n\}$ since none of the $ask(u_i)$ primitives can do a step. As a result,

$$\langle AB \mid \{a_1, \cdots, a_m, b_1, \cdots, b_n\} \rangle \not\longrightarrow$$

Now, by compositionality of the coder with respect to the sequential composition (property P_2), $\mathscr{C}([tell(a)] \,;\, [tell(b)] \,;\, AB) = \mathscr{C}([tell(a)]) \,;\, \mathscr{C}([tell(b)]) \,;\, \mathscr{C}(AB)$, and consequently the following derivation is valid:

$$\langle \mathscr{C}([tell(a)] \,;\, [tell(b)] \,;\, AB) \mid \emptyset \rangle \longrightarrow \cdots \longrightarrow \langle AB \mid \{a_1, \cdots, a_m, b_1, \cdots, b_n\} \rangle$$

and yields a failing computation for $\mathscr{C}([tell(a)] \,;\, [tell(b)] \,;\, AB)$. However, as easily checked, $[tell(a)] \,;\, [tell(b)] \,;\, AB$ has only one successful computation. □

Using similar arguments as in [7], it is possible to extend the previous proof so as to establish the following results.

Proposition 5

1. $\mathscr{L}_g(get,tell) \not\leq \mathscr{L}_r(get,tell)$
2. $\mathscr{L}_g(ask,get,tell) \not\leq \mathscr{L}_r(ask,get,tell)$
3. $\mathscr{L}_g(ask,nask,get,tell) \not\leq \mathscr{L}_r(ask,nask,get,tell)$

5 Performance

Let us now illustrate the gain of efficiency during model-checking obtained by the guarded list construct. To that end, we shall subsequently compare the performance of the Scan and Anemone breath-first search model checker on various examples of the rush hour puzzle coded, on the one hand, without the guarded list construct, and, on the other hand, with the guarded list construct.

As described in the previous sections, the rush hour puzzle can be formulated as a coordination problem by considering cars and trucks as autonomous agents which have to coordinate on the basis of free places. The complete code is available at [4]. Besides sets, maps and widget definitions, it is basically composed of generic procedures for coding horizontal cars and trucks as well as vertical cars and trucks. Specific cars and trucks are then obtained by instantiating colors and places and are put in parallel.

The code for the cars and trucks follows the pattern of the code presented in page 6. Basically, under some conditions, each car and truck amounts to (i) obtaining a free place to move through the execution of a `get` primitive, (ii) then to operating the movement graphically through the execution of a `move` primitive and (iii) finally to freeing the place previously occupied by means of the execution of a `tell` primitive. As an example, the following code is a snippet refining the code of page 6.

```
get(free(pred(r),c));
move(truck_img(c),pred(r),c);
tell(free(succ(succ(r)),c))
```

The problem is solved when the *out* si-term is put on the store, which leads to checking that the property $\#out = 1$ can be reached. As easily checked, the hypotheses of Proposition 1 are verified so that we can replace the above code snippet by the following:

```
[ get(free(pred(r),c)) ->
      move(truck_img(c),pred(r),c),
      tell(free(succ(succ(r)),c)) ]
```

This code is indeed an F-preserving contraction for the formulae $F = (\#out = 1)$.

By performing this transformation, one gains per vehicle the computation of two stores on four, which induces the hope of a gain of performance of 2^n if n is the number of vehicles in parallel. To verify the actual gain of performance, we have model checked the two codes (one with guarded list and the other without guarded list) on the examples of Table 1. They are inspired by cards of the real game and, in view of the above hope, are taken by progressively adding vehicles. The last column in Table 1 gives a brief description of the considered game. The V and H prefixes refer to a vehicle put vertically or horizontally, while the coordinates are those of the rows (counted from top to bottom) and columns (counted from left to right).

Table 2 reports on the data obtained on a portable computer Lenovo x64 bits, running Windows 10 with 16 GB of memory. The first column refers to the test case, the second and the third columns give the time in milliseconds necessary for model checking, the fourth column the time ratio and the last column

Case	Nb's cars/trucks	Game
1	2	VPurpleTruck(2,4), HRedCar(3,2)
2	3	VPurpleTruck(2,1), HRedCar(3,2), HGreenCar(1,1)
3	4	VPurpleTruck(2,1), HRedCar(3,2), HGreenCar(1,1), VOrangeCar(5,1)
4	5	VPurpleTruck(2,1), HRedCar(3,2), HGreenCar(1,1), VOrangeCar(5,1), VBlueTruck(2,4)
5	6	VPurpleTruck(2,1), HRedCar(3,2), HGreenCar(1,1), VOrangeCar(5,1), VBlueTruck(2,4), HGreenTruck(6,3)
6	7	VPurpleTruck(2,1), HRedCar(3,2), HGreenCar(1,1), VOrangeCar(5,1), VBlueTruck(2,4), HGreenTruck(6,3), VYellowTruck(1,6)

Table 1: Test cases

Case	Without GL	With GL	Gain	Expected gain
1	2630 ms (2s)	298 ms (0s)	8.82	4
2	64341 ms (64s |1m)	355 ms (0s)	181	8
3	60339 ms (60s |1m)	770 ms (1s)	78	16
4	495578 ms (496s | 8m)	1032 ms (1s)	480	32
5	3271343 ms (3271s | 55m)	4100 ms (4s)	797	64
6	$\geq$ 10h	4862322 (1h35m)	≥ 6	128

Table 2: Performance results

the hoped gain according to 2^n where n is the number of vehicles in the game. As can be seen from this table, guarded lists lead to a real performance gain and even a greater performance than expected[3]. This can be explained by the fact that the Scan and Anemone model checker relies on non-optimized structures like sequential lists and basically evaluates dynamically the transition system during the model-checking phase. It is also interesting to observe that the exponential behavior resulting from the interleaving of behaviors is kept to a reasonable cost for the first five cases with guarded lists, while it starts exploding from the fourth case without guarded lists. The interested reader may redo the campaign of tests by using the material available at [4].

6 Related work

Although, to the best of our knowledge, it has not been exploited by coordination languages, the idea of forcing statements to be executed without interruption is not new. In [15] Dijkstra has introduced guarded commands, which are statements of the form of $G \rightarrow S$ that atomically executes statement S provided the condition G is evaluated to true. They are mostly combined in repetitive constructs of the

[3] In the last case, we stopped the model-checker after 10 hours of run

form

$$\begin{array}{ll} \textbf{do} & G_0 \rightarrow S_0 \\ \Box & G_1 \rightarrow S_1 \\ & \cdots \\ \Box & G_n \rightarrow S_n \\ \textbf{od} & \end{array}$$

which repetitively selects one of the executable guarded commands until none of them are executable. A non-deterministic choice is operated in the selection of the guarded commands in case several of them can be executed. Later Abrial has used guarded commands in the Event-B method [1]. Such a construct is also at the core of the guarded Horn clause framework proposed by Ueda in [35] to introduce parallelism in logic programming. There Horn clauses are rewritten in the following form

$$H \leftarrow G_1, \cdots, G_m | B_1, \cdots, B_n$$

with H, G_1, ..., G_m, B_1, ..., B_n being atoms. The classical SLD-resolution used to reduce an atom is modified as follows. Assume A is the atom to be reduced. All the clauses whose head H is unifiable with A have their guard $G_1, \cdots, G_m$ evaluated. The first one which succeeds determines the clause that is used, the other being simply discarded. To avoid mismatching instantiations of variables, the evaluation of any G_i is suspended if it can only succeed by binding variables. Finally, several pieces of work have tried to incorporate transactions and atomic constructs in "classical" process algebras, like CCS. For instance, A2CCS [20] proposes to refine complex actions into sequences of elementary ones by modelling atomic behaviors at two levels, with so-called high-level actions being decomposed into atomic sequences of low-level actions. To enforce isolation, atomic sequences are required to go into a special invisible state during all their execution. In fact, sequences of elementary actions are executed sequentially, without interleaving with other actions, as though in a critical section. RCCS [10] is another process algebra incorporating distributed backtracking to handle transactions inside CCS. The main idea is that, in RCCS, each process has access to a log of its synchronization history and may always wind back to a previous state. A similar idea of log is used in AtCCS [2]. There, during the evaluation of an atomic block, actions are recorded in a private log and have no effects outside the scope of the transaction until it is committed. An explicit termination action "end" is used to signal that a transaction is finished and should be committed. States are used in addition to model the evaluation of expressions and can be viewed as tuples put or retrieved from shared spaces in coordination languages. When a transaction has reached commitment and if the local state meets the global one, then all actions present in the log are performed at the same time and the transaction is closed. Otherwise the transaction is aborted.

Our guarded list construct share similarities with these pieces of work. A major difference is however that we restrict the guard to a single primitive to be evaluated. This eases the implementation since, once the primitive has been successfully evaluated, the remaining primitives can be executed in a row without using distributed backtracking as in RCCS, private spaces as in AtCCS for speculative computations and checks for compatibility between local and global environments. Intricate suspensions inherent in guarded Horn clauses are also avoided. Nevertheless, under this restriction, the combination with the non deterministic choice operator $+$ allows to achieve computations similar to the repetitive statements of guarded commands. With respect to these pieces of work, our contribution is also to focus on model checking and to propose a refinement strategy that allows to transform programs by introducing the guarded list construct. An expressiveness study is also proposed in this paper and not in these pieces of work.

Limiting the state explosion problem in model checking by limiting interleaving is similar in spirit with the partial-order reduction introduced in [19, 30, 32, 36]. Realizing that n independent parallel

transitions result in $n!$ different orderings and 2^n different states, the idea is to select a representative composed of $n+1$ states. Indeed, as the transitions are independent, properties need only to be verified on a possible ordering. This technique has been employed in many research efforts for model checking asynchronous systems. However, these efforts aim at designing more efficient algorithms on optimized automata. The approach taken here is different. We do not change our algorithm for model checking, but rather introduce a new construct as well as considerations on refinements to transform programs into more efficient programs.

7 Conclusion

In the aim of improving the performance of the model checking tool introduced in the workbenches Scan [22] and Anemone[23], thÄl's article has introduced a new construct, named guarded list. It has been proved to yield an increase of expressiveness to Linda-like languages, while indeed bringing an increase of efficiency during the model checking phase. In order to pave the way to transform programs by safely introducing the guarded list construct, we have also proposed a notion of refinement and have characterized situations in which one can safely replace a sequence of primitives by a guarded list of primitives.

Our work opens several paths for future research. As regards the expressiveness study, we have used the approach proposed in [6] for a few sublanguages. This naturally leads to deepen the study to include all the sublanguages and to compare them with the L_{MR} and L_{CS} families of languages studied in [6]. Moreover this approach is only one of the possible approaches to compare languages. It would be for instance interesting to verify whether the absolute approach promoted by Zavattaro et al in [8] would change the expressiveness hierarchy of languages. Moreover, expressiveness studies based on bisimulations and fully abstract semantics such as reported in [33] are also worth exploring. As regards model-checking, the algorithm embodied in the Scan and Anemone workbenches is quite elementary and calls for improvements. In that line of research, it would be interesting to study how state collapsing and pruning techniques used for checking large distributed systems may improve the performance of the model checker.

8 Acknowledgment

The authors thank the University of Namur for its support. They also thank the Walloon Region for partial support through the Ariac project (convention 210235) and the CyberExcellence project (convention 2110186). Moreover they are grateful to the anonymous reviewers for their comments on earlier versions of this work.

References

[1] J.-R. Abrial (2010): *Modeling in Event-B - System and Software Engineering*. Cambridge University Press.

[2] L. Acciai, M. Boreale & S. Dal-Zilio (2007): *A Concurrent Calculus with Atomic Transactions*. In R. De Nicola, editor: *Proceedings of the 16th European Symposium on Programming Languages and Systems (ESOP)*, *Lecture Notes in Computer Science* 4421, Springer, pp. 48–63.

[3] J.C.M. Baeten & W.P. Weijland (1990): *Process Algebra*. *Cambridge tracts in Theoretical Computer Science* 18, Cambridge University Press.

[4] M. Barkallah & J.-M. Jacquet (2020): *Model-checking the Rush Hour Bach Program*. Available at `https://staff.info.unamur.be/mbarkall/ICE_2023` or `https://staff.info.unamur.be/jmj/ICE_2023`. Created on May 30th 2023.

[5] F.S. de Boer & C. Palamidessi (1994): *Embedding as a Tool for Language Comparison*. *Information and Computation* 108(1), pp. 128–157.

[6] A. Brogi & J.-M. Jacquet (1998): *On the Expressiveness of Linda-like Concurrent Languages*. *Electronical Notes in Theoretical Computer Science* 16(2), pp. 61–82.

[7] A. Brogi & J.-M. Jacquet (2003): *On the Expressiveness of Coordination via Shared Dataspaces*. *Science of Computer Programming* 46(1-2), pp. 71–98.

[8] N. Busi, R. Gorrieri & G. Zavattaro (2000): *On the Expressiveness of Linda Coordination Primitives*. *Information and Computation* 156(1-2), pp. 90–121.

[9] N. Carriero & D. Gelernter (1989): *Linda in Context*. *Communications of the ACM* 32(4), pp. 444–458.

[10] V. Danos & J. Krivine (2005): *Transactions in RCCS*. In M. Abadi & L. de Alfaro, editors: *Proceedings of the 16th International Conference on Concurrency Theory*, *Lecture Notes in Computer Science* 3653, Springer, pp. 398–412.

[11] D. Darquennes, J.-M. Jacquet & I. Linden (2013): *On Density in Coordiantion Languages*. In C. Canal & M. Villari, editors: *CCIS 393, Advances in Service-Oriented and Cloud Computing, ESOCC 2013, Proceedings of Foclasa Workshop*, Springer, Malaga, Spain, pp. 189–203.

[12] D. Darquennes, J.-M. Jacquet & I. Linden (2013): *On the Introduction of Density in Tuple-Space Coordination Languages*. In: *Science of Computer Programming*, Springer.

[13] D. Darquennes, J.-M. Jacquet & I. Linden (2015): *On Distributed Density in Tuple-based Coordination Languages*. In J. Cámara & J. Proença, editors: *Proceedings 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems*, *EPTCS* 175, Springer, Rome, Italy, pp. 36–53.

[14] D. Darquennes, J.-M. Jacquet & I. Linden (2018): *On Multiplicities in Tuple-Based Coordination Languages: The Bach Family of Languages and Its Expressiveness Study*. In G. Di Marzo Serugendo & M. Loreti, editors: *Proceedings of the 20th International Conference on Coordination Models and Languages*, *Lecture Notes in Computer Science* 10852, Springer, pp. 81–109.

[15] E.W. Dijkstra (1975): *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*. *Communication of the ACM* 18(8), pp. 453–457.

[16] E. Allen Emerson (1990): *Temporal and Modal Logic*. In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, Elsevier, pp. 995–1072.

[17] M. Felleisen (1990): *On the Expressive Power of Programming Languages*. In N. Jones, editor: *Proceedings European Symposium on Programming*, *Lecture Notes in Computer Science* 432, Springer-Verlag, pp. 134–151.

[18] D. Gelernter & N. Carriero (1992): *Coordination Languages and Their Significance*. *Communications of the ACM* 35(2), pp. 97–107.

[19] P. Godefroid & P. Wolper (1991): *Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties*. In K.G. Larsen & A. Skou, editors: *Proceedings of the 3rd International Workshop on Computer Aided Verification*, *Lecture Notes in Computer Science* 575, Springer, pp. 332–342.

[20] R. Gorrieri, S. Marchetti & U. Montanari (1990): *A2CCS: Atomic Actions for CCS*. *Theoretical Computer Science* 72(2&3), pp. 203–223.

[21] C.A.R. Hoare (1985): *Communicating Sequential Processes*. Prentice-Hall.

[22] J.-M. Jacquet & M. Barkallah (2019): *Scan: A Simple Coordination Workbench*. In H. Riis Nielson & E. Tuosto, editors: *Proceedings of the 21st International Conference on Coordination Models and Languages*, *Lecture Notes in Computer Science* 11533, Springer, pp. 75–91.

[23] J.-M. Jacquet & M. Barkallah (2021): *Anemone: A workbench for the Multi-Bach Coordination Language*. *Science of Computer Programming* 202, p. 102579.

[24] J.-M. Jacquet, K. De Bosschere & A. Brogi (2000): *On Timed Coordination Languages*. In A. Porto & G.-C. Roman, editors: *Proc. 4th International Conference on Coordination Languages and Models*, *Lecture Notes in Computer Science* 1906, Springer, pp. 81–98.

[25] J.-M. Jacquet & I. Linden (2007): *Coordinating Context-aware Applications in Mobile Ad-hoc Networks*. In T. Braun, D. Konstantas, S. Mascolo & M. Wulff, editors: *Proceedings of the first ERCIM workshop on eMobility*, The University of Bern, pp. 107–118.

[26] J.-M. Jacquet & I. Linden (2009): *Fully Abstract Models and Refinements as Tools to Compare Agents in Timed Coordination Languages*. *Theoretical Computer Science* 410(2-3), pp. 221–253.

[27] I. Linden & J.-M. Jacquet (2004): *On the Expressiveness of Absolute-Time Coordination Languages*. In R. De Nicola, G.L. Ferrari & G. Meredith, editors: *Proc. 6th International Conference on Coordination Models and Languages*, *Lecture Notes in Computer Science* 2949, Springer, pp. 232–247.

[28] I. Linden & J.-M. Jacquet (2007): *On the Expressiveness of Timed Coordination via Shared Dataspaces*. *Electronical Notes in Theoretical Computer Science* 180(2), pp. 71–89.

[29] I. Linden, J.-M. Jacquet, K. De Bosschere & A. Brogi (2004): *On the Expressiveness of Relative-Timed Coordination Models*. *Electronical Notes in Theoretical Computer Science* 97, pp. 125–153.

[30] K.L. McMillan (1992): *Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits*. In G. von Bochmann & D.K. Probst, editors: *Proceedings of the Fourth International Workshop on Computer Aided Verification*, *Lecture Notes in Computer Science* 663, Springer, pp. 164–177.

[31] R. Milner (1989): *Communication and Concurrency*. PHI Series in computer science, Prentice Hall.

[32] D.A. Peled (1993): *All from One, One for All: on Model Checking Using Representatives*. In C. Courcoubetis, editor: *Proceedings of the 5th International Conference on Computer Aided Verification*, *Lecture Notes in Computer Science* 697, Springer, pp. 409–423.

[33] K. Peters (2019): *Comparing Process Calculi Using Encodings*. In J. Pérez & J. Rot, editors: *Proceedings of the Combined Workshops on Expressiveness in Concurrency and Structural Operational Semantics, (EXPRESS/SOS)*, *EPTCS* 300, pp. 19–38.

[34] E.Y. Shapiro (1992): *Embeddings among Concurrent Programming Languages*. In W.R. Cleaveland, editor: *Proceedings of CONCUR'92*, Springer-Verlag, pp. 486–503.

[35] K. Ueda (1985): *Guarded Horn Clauses*. In E. Wada, editor: *Proceedings of the 4th Conference on Logic Programming*, *Lecture Notes in Computer Science* 221, Springer, pp. 168–179.

[36] A. Valmari (1996): *The State Explosion Problem*. In W. Reisig & G. Rozenberg, editors: *Lectures on Petri Nets I: Basic Models*, *Lecture Notes in Computer Science* 1491, Springer, pp. 429–528.

Proofs about Network Communication: For Humans and Machines

Wolfgang Jeltsch
Well-Typed
London, England
wolfgang@well-typed.com

Javier Díaz
Atix Labs (a Globant Division)
Buenos Aires, Argentina
javier.diaz@globant.com

Many concurrent and distributed systems are safety-critical and therefore have to provide a high degree of assurance. Important properties of such systems are frequently proved on the specification level, but implementations typically deviate from specifications for practical reasons. Machine-checked proofs of bisimilarity statements are often useful for guaranteeing that properties of specifications carry over to implementations. In this paper, we present a way of conducting such proofs with a focus on network communication. The proofs resulting from our approach are not just machine-checked but also intelligible for humans.

1 Introduction

Concurrent and distributed systems are difficult to design and implement, and their correctness is hard to ensure. However, many such systems are safety-critical and therefore have to provide a high degree of assurance. Machine-checked proofs can greatly help to meet this demand. A particular application area of them is the verification of design refinements. A specification may undergo a series of refinement steps to account for practical limitations, ultimately resulting in an implementation. Proving that these refinement steps preserve important properties of the system is vital for assuring the implementation's correctness.

Our current research program focuses on applying design refinement verification to the blockchain consensus protocols of the Ouroboros family [2, 6, 9]. For conducting machine-checked proofs, we use the Isabelle proof assistant together with a custom process calculus, called the Þ-calculus. As a first step, we have proved [8] that direct broadcast, which the protocol specifications assume as the means of data distribution, is behaviorally equivalent to broadcast via multicast, which implementations of the protocols use. For our proof, we have used a domain-specific language for describing network communication, which is embedded in the Þ-calculus.

A weakness of this existing broadcast equivalence proof is that it is not grounded in a formal semantics of the communication language but based on the assumption that certain lower-level bisimilarity statements hold. In this paper, we present a way of proving such bisimilarity statements such that the resulting proofs are machine-checked and intelligible. Concretely, we make the following contributions:

- We present a transition system semantics for the Þ-calculus and derive a transition system semantics for the communication language from it.
- We walk in detail through the proof of a lemma from which several fundamental bisimilarity statements about communication language processes can be derived. The proof of this lemma exemplifies a general way of conducting bisimulation proofs in a concise and human-friendly yet machine-checked fashion. Central to this approach is the combination of the Isabelle/Isar

Preliminary Report. Final version to appear in:
ICE 2023

proof language, a formalized algebra of "up to" methods, Isabelle's *coinduction* proof method, and higher-order abstract syntax.

The formal broadcast equivalence proof and its prerequisites can be obtained from the following sources:

- `https://github.com/input-output-hk/equivalence-reasoner`
- `https://github.com/input-output-hk/transition-systems`
- `https://github.com/input-output-hk/thorn-calculus`
- `https://github.com/input-output-hk/network-equivalences/pull/23`

2 The Þ-Calculus

Our language for describing communication networks is embedded in the Þ-calculus (pronounced "thorn calculus"). The Þ-calculus is a general-purpose process calculus, which we have devised as a tool for convenient development of machine-checked proofs about concurrent and distributed systems. The Þ-calculus in turn is embedded in Isabelle/HOL. We use higher-order abstract syntax (HOAS) for this embedding, since this allows us to have the object language (the Þ-calculus) only deal with the key features of process calculi, which are concurrency and communication, while shifting the treatment of local names, data, computation, conditional execution, and repetition to the meta-language (Isabelle/HOL).

The Þ-calculus strongly resembles the asynchronous π-calculus [7]. Processes communicate via asynchronous channels, which can be global or created locally. Channels are first-class and can therefore be transmitted through other channels, thus making them visible outside their original scopes. This is the mobility feature pioneered by the (synchronous) π-calculus. However, mobility plays only a marginal role in this paper, since it is not exploited by the communication language.

Definition 1 (Syntax of Þ-calculus processes)**.** *The syntax of Þ-calculus processes is given by the following BNF rule, where a denotes channels, x denotes values, p and q denote processes, and P denotes functions from channels or from values to processes, depending on the context:*

$$\mathit{Process} ::= \mathbf{0} \mid a \lhd x \mid a \rhd x.\, P\, x \mid p \parallel q \mid \nu a.\, P\, a$$

The processes generated by the different alternatives of this BNF rule are called the stop process, senders, receivers, parallel compositions, and restrictions, respectively. Parallel composition is right-associative and has lowest precedence; the other constructs have highest precedence.

Informally, the semantics of the Þ-calculus is characterized by the following behavior of processes:

- The stop process $\mathbf{0}$ does nothing.
- A sender $a \lhd x$ sends value x to channel a.
- A receiver $a \rhd x.\, P\, x$ receives a value x from channel a and continues like $P\, x$.
- A parallel composition $p \parallel q$ performs p and q in parallel.
- A restriction $\nu a.\, P\, a$ introduces a local channel a and behaves like $P\, a$.

Formally, the semantics is defined as a labeled transition system. Since mobility is not essential to the topics of this paper and is at the same time complex to handle, we present only a simplified version of the semantics that ignores mobility.[1]

[1] We refer the reader to the accompanying Isabelle code for the full semantics.

$$\frac{}{a \lhd x \xrightarrow{a \lhd x} \mathbf{0}} \ (\lhd) \qquad \frac{}{a \rhd x.\, P\, x \xrightarrow{a \rhd x} P\, x} \ (\rhd)$$

$$\frac{p \xrightarrow{a \lhd x} p' \quad q \xrightarrow{a \rhd x} q'}{p \parallel q \xrightarrow{\tau} p' \parallel q'} \ (\tau_{\rightarrow}) \qquad \frac{p \xrightarrow{a \rhd x} p' \quad q \xrightarrow{a \lhd x} q'}{p \parallel q \xrightarrow{\tau} p' \parallel q'} \ (\tau_{\leftarrow})$$

$$\frac{p \xrightarrow{\alpha} p'}{p \parallel q \xrightarrow{\alpha} p' \parallel q} \ (\parallel_1) \qquad \frac{q \xrightarrow{\alpha} q'}{p \parallel q \xrightarrow{\alpha} p \parallel q'} \ (\parallel_2) \qquad \frac{\forall a.\, P\, a \xrightarrow{\alpha} Q\, a}{\nu a.\, P\, a \xrightarrow{\alpha} \nu a.\, Q\, a} \ (\nu)$$

Figure 1: No-mobility versions of the transition rules of the Þ-calculus

Definition 2 (Syntax of Þ-calculus actions)**.** *The syntax of Þ-calculus actions is given by the following BNF rule, where a denotes channels and x denotes values:*

$$\mathit{Action} ::= a \lhd x \mid a \rhd x \mid \tau$$

The actions generated by the different alternatives of this BNF rule are called sending actions, receiving actions, and the internal-transfer action, respectively.

The intuitive meanings of the different actions are as follows:

- A sending action $a \lhd x$ means sending value x to channel a.
- A receiving action $a \rhd x$ means receiving value x from channel a.
- The internal-transfer action τ means transferring some value through some channel.

Definition 3 (Semantics of the Þ-calculus)**.** *The semantics of the Þ-calculus is given by the transition relation $\rightarrow \subseteq \mathit{Process} \times \mathit{Action} \times \mathit{Process}$ that is defined by the introduction rules in Figure 1.*

As usual, $p \xrightarrow{\alpha} q$ intuitively means that process p can perform action α and then continue like process q.

Definition 4 (Strong and weak bisimilarity of Þ-calculus processes)**.** *The relations $\sim \subseteq \mathit{Process} \times \mathit{Process}$ and $\approx \subseteq \mathit{Process} \times \mathit{Process}$ denote strong and weak bisimilarity derived from $\rightarrow$ in the usual way.*

Strong and weak bisimilarity possess various properties common for process calculi, for which proofs can be found in the accompanying Isabelle code.

Lemma 1 (Inclusion of strong bisimilarity in weak bisimilarity)**.** *Strongly bisimilar processes are also weakly bisimilar; formally, $\sim \subseteq \approx$.*

Lemma 2 (Congruence properties of bisimilarities)**.** *Strong and weak bisimilarity are congruence relations with respect to parallel composition and restriction; that is, they are equivalence relations, and the following propositions hold:*

$$p_1 \sim p_2 \wedge q_1 \sim q_2 \rightarrow p_1 \parallel q_1 \sim p_2 \parallel q_2 \tag{1}$$

$$p_1 \approx p_2 \wedge q_1 \approx q_2 \rightarrow p_1 \parallel q_1 \approx p_2 \parallel q_2 \tag{2}$$

$$(\forall a.\, P_1\, a \sim P_2\, a) \rightarrow \nu a.\, P_1\, a \sim \nu a.\, P_2\, a \tag{3}$$

$$(\forall a.\, P_1\, a \approx P_2\, a) \rightarrow \nu a.\, P_1\, a \approx \nu a.\, P_2\, a \tag{4}$$

Lemma 3 (Fundamental bisimilarity properties)**.** *The following strong bisimilarity properties hold:*

$$\mathbf{0} \parallel p \sim p \tag{5}$$
$$p \parallel \mathbf{0} \sim p \tag{6}$$
$$(p \parallel q) \parallel r \sim p \parallel (q \parallel r) \tag{7}$$
$$p \parallel q \sim q \parallel p \tag{8}$$
$$\nu a.\, \nu b.\, P\, a\, b \sim \nu b.\, \nu a.\, P\, a\, b \tag{9}$$
$$\nu a.\, p \sim p \tag{10}$$

Unlike the asynchronous π-calculus, the Þ-calculus does not contain a construct for guarded recursion, and it also does not contain a replication construct as found in the synchronous π-calculus. This is because the use of HOAS allows us to resort to the recursion features of the host language and in particular to build infinite processes, since the type of processes is coinductive. We could use this possibility to define guarded recursion and replication on top of the Þ-calculus and also directly to construct processes involving repetition. However, we introduce a guarded replication construct instead, which we use to realize any repetition.

Definition 5 (Repeating receivers)**.** *Processes $a \triangleright^{\infty} x.\, P\, x$, where a denotes channels, x denotes values, and P denotes functions from values to processes, are defined as follows:*

$$a \triangleright^{\infty} x.\, P\, x = a \triangleright x.\, (P\, x \parallel a \triangleright^{\infty} x.\, P\, x) \tag{11}$$

Such processes are called repeating receivers. The precedence of $\triangleright^{\infty}$ is the same as the one of $\triangleright$.

As can be seen from Equation 11, a repeating receiver $a \triangleright^{\infty} x.\, P\, x$ repeatedly receives values x from channel a and after each receipt initiates the execution of $P\, x$.

Lemma 4 (Transitions from repeating receivers)**.** *The only transitions possible from repeating receivers are of the form $a \triangleright^{\infty} x.\, P\, x \xrightarrow{a \triangleright x} P\, x \parallel a \triangleright^{\infty} x.\, P\, x$.*

Proof. According to Equation 11, repeating receivers are receivers of a special kind. The only transition rule that introduces transitions from receivers is $\triangleright$, and applying this rule to repeating receivers leads to transitions of the above-mentioned form. □

3 The Communication Language

The communication language is a process calculus specifically designed for describing communication networks. It differs from the Þ-calculus in that it does not allow for arbitrary sending and receiving but instead provides constructs for forwarding, removing, and duplicating values in channels. These constructs are more high-level than the Þ-calculus constructs they replace. They are also more limiting but still permit the communication language to express data flow in a network. By staying within the confines of the communication language, our network-related specifications and proofs tend to be well structured and comprehensible.

Definition 6 (Syntax of communication language processes)**.** *The syntax of communication language processes is given by the following BNF rule, where a, b, and b_i denote channels, p and q denote processes, and P denotes functions from channels to processes:*

$$\mathit{Process} ::= \mathbf{0} \mid a \Rightarrow [b_1, \ldots, b_n] \mid a \rightarrow b \mid a \leftrightarrow b \mid ¤^{?} a \mid ¤^{+} a \mid ¤^{*} a \mid p \parallel q \mid \nu a.\, P\, a$$

The processes generated by the different alternatives of this BNF rule are called the stop process, distributors, unidirectional bridges, bidirectional bridges, losers, duplicators, duplosers, parallel compositions, and restrictions, respectively. Parallel composition is right-associative and has lowest precedence; the other constructs have highest precedence.

The stop process, parallel compositions, and restrictions behave like they do in the Þ-calculus. The behavior of the other communication language constructs is informally characterized as follows:

- A distributor $a \Rightarrow [b_1, \ldots, b_n]$ continuously forwards values from channel a to all channels b_i.
- A unidirectional bridge $a \rightarrow b$ continuously forwards values from channel a to channel b.
- A bidirectional bridge $a \leftrightarrow b$ continuously forwards values from channel a to channel b and from channel b to channel a.
- A loser $¤^? a$ continuously removes values from channel a.
- A duplicator $¤^+ a$ continuously duplicates values in channel a.
- A duploser $¤^* a$ continuously removes values from and duplicates values in channel a.

Example 1 (Reliable anycast with three receivers)**.** *Consider a reliable anycast connection between a sender and three receivers, the latter being numbered from* 1 *to* 3*. Assume that the sender is equipped with a buffer for packets to be sent and each receiver is equipped with a buffer for packets received. If we model the sender's buffer by a channel s and the buffer of each receiver i by a channel r_i, this anycast connection can be modeled by the following process:*

$$\nu t.\, (s \rightarrow t \parallel t \rightarrow r_1 \parallel t \rightarrow r_2 \parallel t \rightarrow r_3)$$

Note that values in the local channel t model packets in transit.

Example 2 (Unreliable broadcast with three receivers)**.** *Consider a broadcast connection between a sender and three receivers that is unreliable in the sense that packets may be lost or duplicated. If channels s, r_1, r_2, and r_3 model send and receive buffers like in Example 1, this broadcast connection can be modeled by the following process:*

$$\nu t.\, (s \rightarrow t \parallel ¤^* t \parallel t \rightarrow r_1 \parallel t \rightarrow r_2 \parallel t \rightarrow r_3)$$

This process models indeed a broadcast connection, not an anycast connection, because due to duplication a single value sent to t may be forwarded to different channels r_i.

Definition 7 (Embedding of the communication language in the Þ-calculus)**.** *The communication language is a DSL embedded in the Þ-calculus. The stop process, parallel composition, and restriction are directly taken from the Þ-calculus, and the other communication language constructs are derived from Þ-calculus constructs and the repeating receiver construct as follows:*

$$a \Rightarrow [b_1, \ldots, b_n] = a \triangleright^{\infty} x.\, (b_1 \triangleleft x \parallel \ldots \parallel b_n \triangleleft x \parallel \mathbf{0}) \tag{12}$$

$$a \rightarrow b = a \Rightarrow [b] \tag{13}$$

$$a \leftrightarrow b = a \rightarrow b \parallel b \rightarrow a \tag{14}$$

$$¤^? a = a \Rightarrow [] \tag{15}$$

$$¤^+ a = a \Rightarrow [a, a] \tag{16}$$

$$¤^* a = ¤^? a \parallel ¤^+ a \tag{17}$$

$$\frac{}{a \lhd x \xrightarrow{a \lhd x} \mathbf{0}} \quad (\lhd)$$

$$\frac{}{a \Rightarrow [b_1, \ldots, b_n] \xrightarrow{a \rhd x} (b_1 \lhd x \parallel \ldots \parallel b_n \lhd x \parallel \mathbf{0}) \parallel a \Rightarrow [b_1, \ldots, b_n]} \quad (\Rightarrow)$$

$$\frac{p \xrightarrow{a \lhd x} p' \quad q \xrightarrow{a \rhd x} q'}{p \parallel q \xrightarrow{\tau} p' \parallel q'} \quad (\tau_{\rightarrow}) \qquad \frac{p \xrightarrow{a \rhd x} p' \quad q \xrightarrow{a \lhd x} q'}{p \parallel q \xrightarrow{\tau} p' \parallel q'} \quad (\tau_{\leftarrow})$$

$$\frac{p \xrightarrow{\alpha} p'}{p \parallel q \xrightarrow{\alpha} p' \parallel q} \quad (\parallel_1) \qquad \frac{q \xrightarrow{\alpha} q'}{p \parallel q \xrightarrow{\alpha} p \parallel q'} \quad (\parallel_2) \qquad \frac{\forall a.\, P\, a \xrightarrow{\alpha} Q\, a}{\nu a.\, P\, a \xrightarrow{\alpha} \nu a.\, Q\, a} \quad (\nu)$$

Figure 2: No-mobility versions of the transition rules of the extended communication language

Note that among the derivations in Definition 7 only the one of distributors directly refers to constructs outside the communication language; all other derivations refer to communication language constructs only. Therefore, we consider $\rightarrow$, $\leftrightarrow$, $¤^?$, $¤^+$, and $¤^*$ as merely providing syntactic sugar and discuss only the communication language fragment formed by $\mathbf{0}$, $\Rightarrow$, $\parallel$, and ν in the remainder of this section.

Since the communication language is embedded in the Þ-calculus, we can derive a formal semantics for it from the formal semantics of the Þ-calculus.[2] For dealing with the constructs inherited from the Þ-calculus, we reuse the corresponding transition rules, which are $\tau_{\rightarrow}$, $\tau_{\leftarrow}$, $\parallel_1$, $\parallel_2$, and ν. For dealing with distributors, which are receivers of a particular shape, we specialize the $\rhd$-rule appropriately, resulting in a new rule $\Rightarrow$. Transitions from distributors with at least one target channel result in processes that contain senders. Therefore, our transition system must be able to cope with the additional presence of senders in processes. We reuse the $\lhd$-rule from the Þ-calculus for this purpose. We call the communication language extended with senders the extended communication language.

Definition 8 (Syntax of processes of the extended communication language)**.** *The syntax of processes of the extended communication language is given by the following BNF rule, where a, b, and b_i denote channels, x denotes values, p and q denote processes, and P denotes functions from channels to processes:*

$$\mathit{Process} ::= \mathbf{0} \mid a \lhd x \mid a \Rightarrow [b_1, \ldots, b_n] \mid a \rightarrow b \mid a \leftrightarrow b \mid ¤^? a \mid ¤^+ a \mid ¤^* a \mid p \parallel q \mid \nu a.\, P\, a$$

The processes generated by the different alternatives of this BNF rule are called the stop process, senders, distributors, unidirectional bridges, bidirectional bridges, losers, duplicators, duplosers, parallel compositions, and restrictions, respectively. Parallel composition is right-associative and has lowest precedence; the other constructs have highest precedence.

Proposition 1 (Semantics of the extended communication language)**.** *The restriction of the transition relation $\rightarrow$ of the Þ-calculus to processes of the extended communication language is generated by the introduction rules in Figure 2.*

Lemma 5 (Strong and weak bisimilarity of processes of the extended communication language)**.** *The strong and weak bisimilarity relations derived from the transition relation described in Figure 2 arise from*

[2] Also this semantics ignores mobility, because we derive it from the no-mobility version of the Þ-calculus semantics. However, unlike with the Þ-calculus, the gap between the semantics presented here and the full semantics is minimal, since the absence of arbitrary sending makes it impossible to send local channels to the environment. In fact, the only additional feature of the full semantics is that it accounts for the possibility of distributors *receiving* previously unknown channels from the environment.

restricting the bisimilarity relations $\sim$ and $\approx$ of the Þ-calculus to processes of the extended communication language.

Proof. The only difference between the transition rules in Figures 1 and 2 is that the former include $\triangleright$ where the latter include $\Rightarrow$. However, $\Rightarrow$ is just $\triangleright$ restricted to those situations where the source process has the shape of a distributor. Therefore, simulation in the extended communication language can be performed according to the Þ-calculus semantics and only in this way. As a result, processes are strongly or weakly bisimilar according to the semantics of the extended communication language exactly if they are bisimilar (strongly or weakly, respectively) according to the semantics of the Þ-calculus. □

Corollary 1. *Lemmas 1 to 3 carry over to the extended communication language.*

4 A Proof of Idempotency of Repeating Receivers

As mentioned in Section 1, the proof of equivalence of direct broadcast and broadcast via multicast as presented in our previous work [8] relies on certain lower-level bisimilarity statements. Meanwhile, we have developed proofs for most of these statements.[3] Some of these proofs merely reduce the respective bisimilarity statements to more basic bisimilarity statements, but the proofs of the fundamental statements refer directly to the transition system semantics of the Þ-calculus and the communication language. These latter proofs are bisimulation proofs in the style that we advocate in this paper.

To illustrate this style, let us turn our attention to a group of idempotency laws. First note that all communication language constructs not inherited from the Þ-calculus are idempotent up to strong bisimilarity with respect to parallel composition, which is vital for our broadcast equivalence proof.

Lemma 6 (Idempotency of genuine communication language constructs)**.** *The following idempotency properties hold:*

$$a \Rightarrow bs \parallel a \Rightarrow bs \sim a \Rightarrow bs \tag{18}$$

$$a \rightarrow b \parallel a \rightarrow b \sim a \rightarrow b \tag{19}$$

$$a \leftrightarrow b \parallel a \leftrightarrow b \sim a \leftrightarrow b \tag{20}$$

$$¤^? a \parallel ¤^? a \sim ¤^? a \tag{21}$$

$$¤^+ a \parallel ¤^+ a \sim ¤^+ a \tag{22}$$

$$¤^* a \parallel ¤^* a \sim ¤^* a \tag{23}$$

The proofs of these idempotency properties are part of the accompanying Isabelle code. They reduce these properties to a fundamental idempotency law, which is idempotency of repeating receivers.

Lemma 7 (Idempotency of repeating receivers)**.** *The following idempotency property holds:*

$$a \triangleright^\infty x.\, P\, x \parallel a \triangleright^\infty x.\, P\, x \sim a \triangleright^\infty x.\, P\, x \tag{24}$$

It is this idempotency law that we use as our example for demonstrating our style of bisimulation proofs that are concise and human-friendly yet machine-checked. However, before we turn to the Isabelle/HOL proof that exhibits this style, we provide a semi-formal proof of this law.

Semi-formal proof of Lemma 7. We prove the idempotency of repeating receivers by bisimulation up to strong bisimilarity and context.

[3] These proofs can be found in the accompanying Isabelle code.

Forward simulation. Assume that $a \triangleright^\infty x.\, Px \parallel a \triangleright^\infty x.\, Px \xrightarrow{\alpha} s$ for arbitrary but fixed α and s. Looking at Figure 1, we can see that only rules $\tau_{\rightarrow}$, $\tau_{\leftarrow}$, $\parallel_1$, and $\parallel_2$ can in principle introduce this transition, given that its source process is a parallel composition.

Rules $\tau_{\rightarrow}$ and $\tau_{\leftarrow}$. Introducing the above transition using either of these rules requires a $\triangleleft$-transition from $a \triangleright^\infty x.\, Px$, which is not possible according to Lemma 4.

Rule $\parallel_1$. For introducing the above transition using this rule, there must be a process q such that the following statements hold:

$$a \triangleright^\infty x.\, Px \xrightarrow{\alpha} q \tag{i}$$

$$s = q \parallel a \triangleright^\infty x.\, Px \tag{ii}$$

Based on Lemma 4, statement (i) implies that there is an x for which the following propositions are true:

$$\alpha = a \triangleright x \tag{iii}$$

$$q = Px \parallel a \triangleright^\infty x.\, Px \tag{iv}$$

From (ii) and (iv), we can deduce the following:

$$s = (Px \parallel a \triangleright^\infty x.\, Px) \parallel a \triangleright^\infty x.\, Px \tag{v}$$

Because of (iii) and (v), the transition we have started with has the following concrete shape:

$$a \triangleright^\infty x.\, Px \parallel a \triangleright^\infty x.\, Px \xrightarrow{a \triangleright x} (Px \parallel a \triangleright^\infty x.\, Px) \parallel a \triangleright^\infty x.\, Px \tag{vi}$$

We simulate this transition with the transition $a \triangleright^\infty x.\, Px \xrightarrow{a \triangleright x} Px \parallel a \triangleright^\infty x.\, Px$, whose existence follows from (i), (iii), and (iv). The target processes of these two transitions are the processes we have started with up to strong bisimilarity and context. To see why, observe that the first target process can be transformed into a bisimilar one as follows, employing Bisimilarity 7:

$$(Px \parallel a \triangleright^\infty x.\, Px) \parallel a \triangleright^\infty x.\, Px \sim Px \parallel (a \triangleright^\infty x.\, Px \parallel a \triangleright^\infty x.\, Px) \tag{vii}$$

Removing the common context $Px \parallel [\cdot]$ from the result of this transformation and the target process of the simulating transition yields $a \triangleright^\infty x.\, Px \parallel a \triangleright^\infty x.\, Px$ and $a \triangleright^\infty x.\, Px$.

Rule $\parallel_2$. This rule can be handled analogously to rule $\parallel_1$.

Backward simulation. Assume that $a \triangleright^\infty x.\, Px \xrightarrow{\alpha} s$ for arbitrary but fixed α and s. Lemma 4 tells us that there is an x such that $\alpha = a \triangleright x$ and $s = Px \parallel a \triangleright^\infty x.\, Px$, from which we can deduce that the transition we have started with is concretely $a \triangleright^\infty x.\, Px \xrightarrow{a \triangleright x} Px \parallel a \triangleright^\infty x.\, Px$. By applying rule $\parallel_1$, we can turn this transition into transition (vi), which we use as the simulating transition. The target processes of the original and the simulating transition are the processes we have started with up to strong bisimilarity and context, for essentially the same reasons as in the case of forward simulation of transitions generated by rule $\parallel_1$. □

Figure 3 presents the formal proof of Lemma 7. To not bother the reader with technicalities, this presentation omits the subproofs that justify the atomic reasoning steps. These subproofs are only short, straightforward applications of lemmas and proof methods. The complete proof can be found in the accompanying Isabelle code. Note that the formal proof, also as shown here, refers to the full semantics and thus has to deal with mobility.

To aid understanding of the formal proof, let us point out a few things:

lemma *repeated_receive_idempotency*:
  shows $a \triangleright^\infty x.\, P\,x \parallel a \triangleright^\infty x.\, P\,x \sim a \triangleright^\infty x.\, P\,x$
proof (*coinduction rule*: *up_to_rule* [**where** $\mathcal{F} = [\sim] \frown \mathcal{M}$])
  case (*forward_simulation* α s)
  then show *?case*
  proof *cases*
    case (*parallel_left_io* η a' n x q)
    from $\langle a \triangleright^\infty x.\, P\,x \xrightarrow{IO\,\eta\,a'\,n\,x} q\rangle$ **obtain** t **where** $q = t \parallel (a \triangleright^\infty x.\, P\,x) \gg\!\!= \mathit{suffix}\,n$
      ⟨*proof*⟩
    with $\langle a \triangleright^\infty x.\, P\,x \xrightarrow{IO\,\eta\,a'\,n\,x} q\rangle$ **have** $a \triangleright^\infty x.\, P\,x \xrightarrow{IO\,\eta\,a'\,n\,x} t \parallel (a \triangleright^\infty x.\, P\,x) \gg\!\!= \mathit{suffix}\,n$
      ⟨*proof*⟩
    moreover have $(t \parallel r \gg\!\!= \mathit{suffix}\,n) \parallel r \gg\!\!= \mathit{suffix}\,n \sim t \parallel (r \parallel r) \gg\!\!= \mathit{suffix}\,n$ **for** r
      ⟨*proof*⟩
    ultimately show *?thesis*
      ⟨*proof*⟩
  next
    case (*parallel_right_io* η a' n x q)
    from $\langle a \triangleright^\infty x.\, P\,x \xrightarrow{IO\,\eta\,a'\,n\,x} q\rangle$ **obtain** t **where** $q = t \parallel (a \triangleright^\infty x.\, P\,x) \gg\!\!= \mathit{suffix}\,n$
      ⟨*proof*⟩
    with $\langle a \triangleright^\infty x.\, P\,x \xrightarrow{IO\,\eta\,a'\,n\,x} q\rangle$ **have** $a \triangleright^\infty x.\, P\,x \xrightarrow{IO\,\eta\,a'\,n\,x} t \parallel (a \triangleright^\infty x.\, P\,x) \gg\!\!= \mathit{suffix}\,n$
      ⟨*proof*⟩
    moreover have $r \gg\!\!= \mathit{suffix}\,n \parallel (t \parallel r \gg\!\!= \mathit{suffix}\,n) \sim t \parallel (r \parallel r) \gg\!\!= \mathit{suffix}\,n$ **for** r
      ⟨*proof*⟩
    ultimately show *?thesis*
      ⟨*proof*⟩
  qed (*blast elim*: *transition_from_repeated_receive*)+
next
  case (*backward_simulation* α s)
  from $\langle a \triangleright^\infty x.\, P\,x \xrightarrow{\alpha} s\rangle$ **obtain** n **and** x **where** $\alpha = a \triangleright_n x$ **and** $s = \mathit{post\_receive}\,n\,x\,P \parallel (a \triangleright^\infty x.\, P\,x) \gg\!\!= \mathit{suffix}\,n$
    ⟨*proof*⟩
  with $\langle a \triangleright^\infty x.\, P\,x \xrightarrow{\alpha} s\rangle$ **have** $a \triangleright^\infty x.\, P\,x \xrightarrow{a \triangleright_n x} \mathit{post\_receive}\,n\,x\,P \parallel (a \triangleright^\infty x.\, P\,x) \gg\!\!= \mathit{suffix}\,n$
    ⟨*proof*⟩
  then have $a \triangleright^\infty x.\, P\,x \parallel a \triangleright^\infty x.\, P\,x \xrightarrow{a \triangleright_n x} (\mathit{post\_receive}\,n\,x\,P \parallel (a \triangleright^\infty x.\, P\,x) \gg\!\!= \mathit{suffix}\,n) \parallel (a \triangleright^\infty x.\, P\,x) \gg\!\!= \mathit{suffix}\,n$
    ⟨*proof*⟩
  moreover have $(t \parallel r \gg\!\!= \mathit{suffix}\,n) \parallel r \gg\!\!= \mathit{suffix}\,n \sim t \parallel (r \parallel r) \gg\!\!= \mathit{suffix}\,n$ **for** r
    ⟨*proof*⟩
  ultimately show *?case*
    ⟨*proof*⟩
qed *respectful*

Figure 3: Formal proof of idempotency of repeating receivers

- The initial proof method uses the term $[\sim] \frown \mathcal{M}$ to specify "up to strong bisimilarity and context" as the "up to" method to use. To guarantee that the provided term specifies an "up to" method that is sound, we have to prove that it fulfills a certain condition. We do that by invoking the automated proof method *respectful* at the end of the proof.
- The part on forward simulation mentions actions of the form $IO\,\eta\,a\,n\,x$. Such actions can be sending or receiving actions. For reasons having to do with mobility, there are separate versions of $\|_1$ and $\|_2$ for sending and receiving actions on the one hand and the internal-transfer action on the other. The cases *parallel_left_io* and *parallel_right_io* are only about sending and receiving, not about internal transfer.
- There are no explicit proof steps for showing that the original transition of a forward simulation cannot be introduced using $\tau_{\rightarrow}$ or $\tau_{\leftarrow}$. We have that automatically shown by the proof method (*blast elim*: *transition_from_repeated_receive*)+ at the end of the forward simulation part. This proof method additionally shows that said transition cannot be introduced using the internal-transfer versions of $\|_1$ and $\|_2$ mentioned in the previous item.
- Mobility makes it possible to receive previously unknown channels from the environment. To deal with this possibility, some tweaks are necessary, namely adding $\gg \mathit{suffix}\,n$ in a few places, switching to a more powerful kind of receiving action, $\triangleright_n$, and replacing $P\,x$ by $\mathit{post\_receive}\,n\,x\,P$. A deeper discussion of these tweaks would be outside the scope of this paper.

As can be seen, the formal proof is quite similar to the semi-formal one, which we consider a strength of our work. It is generally more compact, but the handling of forward simulation of transitions generated by rule $\|_2$ had to be spelled out, where the semi-formal proof could just state that it is analogous to what was done for rule $\|_1$.

5 Bisimulation Proofs for Humans and Machines

The semi-formal proof of Lemma 7 is geared toward human readers, and its style has been chosen accordingly. The formal proof, by following the semi-formal proof rather closely, retains this human-friendly style to a large extend but is machine-checked at the same time. This achievement rests on the combination of several tools:

The Isabelle/Isar proof language. Isabelle/Isar [15] is a structured, declarative proof language that incorporates elements of mathematical prose. With these characteristics, Isar proofs differ notably from proof terms as well as tactics-based proof scripts, with the result of being better understandable by humans. Despite its human-friendliness, Isar comes with a precise semantics, and the correctness of Isar proofs can be checked using the Isabelle proof assistant.

The use of Isar is crucial for having the formal proof largely reflect the semi-formal proof. The block structure achieved by employing **proof**, **case**, **next**, and **qed** resembles the overall structure of the semi-formal proof, in particular the nesting of subproofs and the distinction between forward and backward simulation as well as between different introduction rules. At the bottom layer, intermediate facts are explicitly stated and later accessed using Isar's flexible means for fact referencing. Other, minor, features of Isar serve to further narrow the gap between the formal and the semi-formal proof.

A formalized algebra of "up to" methods. Both the semi-formal and the formal proof have to cope with the fact that transitions from $a \triangleright^{\infty} x.\, P\,x \parallel a \triangleright^{\infty} x.\, P\,x$ and $a \triangleright^{\infty} x.\, P\,x$ do not result in these

processes again but only in processes that can be derived from them by adding a common context and performing a strong bisimilarity transformation. However, this is not a problem, because employing the "up to strong bisimilarity and context" method bridges this gap.

A bisimulation proof that does not employ "up to" methods would be much more complex. Such a proof would have to show that bisimulation is also possible for the above-mentioned target processes and recursively for any processes that arise from bisimulation of previously considered processes. In the end, instead of dealing with repeating receivers only, the proof would have to deal with all processes of the form $u_1 \parallel \ldots \parallel u_n \parallel a \rhd^\infty x.\, P\,x$ where a process u_i is either a sender or the stop process. Since the processes to be proved bisimilar contain a total of three repeating receivers, this would result in an enormous amount of boilerplate that would obscure the key arguments of the proof. Furthermore, such a proof would be hard to develop in the first place.

In order to prevent such issues, we have implemented an algebra of "up to" methods that are guaranteed to be sound, using Isabelle/HOL. This implementation enables developers of formal bisimulation proofs to construct custom "up to" methods that fit the specific bisimilarity statements to prove. In the proof of idempotency of repeating receivers, we use the "up to" method $[\sim] \frown \mathcal{M}$. This method is built from the primitive methods $\mathcal{M}$ and $[\sim]$. $\mathcal{M}$ requires target processes to be source processes up to context[4], and $[\sim]$ requires target processes to be strongly bisimilar, independently of source processes. The operator $\frown$ serves to combine the two. Note that $[\sim] \frown \mathcal{M}$ allows only the first process to deviate by strong bisimilarity, which is the one for which we need this possibility; full "up to strong bisimilarity and context" is denoted by $[\sim] \frown \mathcal{M} \frown [\sim]$.

The *coinduction* proof method. Isabelle's *coinduction* proof method [5] makes it possible to conduct coinductive proofs using the **proof–case–next–qed** style exemplified by our formal proof of Lemma 7. Isabelle/HOL supports coinductive definitions of data types and predicates, and in its default mode the *coinduction* method enables reasoning along the coinductive structure of the data types and predicates so defined. In the case of bisimilarity, which is a coinductively defined predicate, this leads to plain bisimulation proofs, those that do not employ "up to" methods.

However, the *coinduction* method can also work with user-provided coinduction rules, which can be lemmas derived from the coinduction rules induced by coinductive data type and predicate definitions. This allows us to use the *coinduction* method for bisimulation proofs that apply "up to" methods. For employing a concrete "up to" method, we can instantiate the generic lemma *up_to_rule* for this "up to" method and provide the resulting fact as the coinduction rule to use to the *coinduction* proof method.

A feature of the *coinduction* method that helps making proofs concise is the automatic derivation of bisimulation relations. As indicated in the previous item, bisimulation relations often have to cover more than just the processes to be proved bisimilar if "up to" methods are not used, since target processes typically deviate from source processes. However, in most bisimulation proofs that do use "up to" methods, including our proof of idempotency of repeating receivers, this issue does not arise, and the bisimulation relation of choice is the one that just covers the processes whose bisimilarity is to be shown. The *coinduction* method derives this relation from the proof goal and automatically shows the trivial statement that the processes to be proved bisimilar are in this relation.[5] As a result, the proof can concentrate on the actual bisimulation.

[4] Actually up to mutation, which is more general than up to context.

[5] This is what distinguishes it from the *coinduct* method [15, Subsection 6.5.2], which requires the user to specify the bisimulation relation and prove that the processes to be proved bisimilar are in this relation.

Higher-order abstract syntax. Higher-order abstract syntax (HOAS) [10] is a technique of embedding an object language in a higher-order host language where name binding in the object language is expressed using functions of the host language. When not using HOAS, formal proofs that involve binding constructs tend to be littered with boilerplate for dealing with issues like name capturing and freshness conditions. By employing HOAS, this problem can be prevented. As we have seen in Section 4, also our use of HOAS necessitates additional handling of technicalities as soon as mobility is taken into account. However, the corresponding amount of extra code tends to be low compared to the amount of extra code necessary with non-HOAS approaches, including those that make use of nominal logic, which is generally boilerplate-reducing.[6]

The use of HOAS makes it possible to construct exotic terms, that is, terms where functions representing binding yield subterms whose structure depends on the arguments of these functions. In the case of receivers, we consider this a feature, as it allows us to handle computation and conditional execution within the meta-language. However, in the case of restrictions, exotic terms may become an issue when treating mobility naïvely; for example, Bisimilarities 1 and 2 may not hold anymore. The typical solution to such problems is to restrict the calculus in question to terms that are not exotic. The solution of the Þ-calculus, however, is different: exotic terms can be constructed freely, but the transition system semantics does not allow transitions from exotic restrictions. Only the full transition system, which can be found in the accompanying Isabelle code, has this feature of preventing such transitions. To achieve it, the transition system has to maintain lists of channels introduced by restrictions, and this results in the need for the additional tweaks present in the formal proof.

6 Related Work

Various domain-specific languages for modeling communication networks and reasoning about them are discussed in the literature. One of them is NetKAT [1], a network programming language based on Kleene algebra with tests (KAT) that features a complete deductive system and a PSPACE decision procedure. Unlike our communication language, NetKAT lacks restriction and, being a sequential language, also parallel composition. On the other hand, it allows for packet inspection and modification. Another example of a network communication DSL is Nettle [14], a language for programming OpenFlow networks that is embedded in Haskell and based on the principles of functional reactive programming. Like with NetKAT, packet inspection and modification is also possible with Nettle.

Process calculi for describing and verifying communication networks have been an active area of research. For example, the ω-calculus [12] is a process calculus devised to formally reason about mobile ad-hoc networks (MANETs). It is a conservative extension of the π-calculus that has built-in support for unicast and broadcast communication as well as location-based scoping. We have designed the Þ-calculus as a general-purpose process calculus and have thus avoided the inclusion of application-specific features like support for broadcast communication. That said, such features can be implemented on top of the Þ-calculus, as the definition of the communication language as an embedded DSL and Examples 1 and 2 show.

Several well-known process calculi have been formalized by Bengtson and colleagues in Isabelle/HOL,

[6] For example, the complete implementation of "up to" methods for the Þ-calculus is less than half the size of the implementation of "up to" methods for ψ-calculi [11], which also uses Isabelle/HOL and employs the Nominal Isabelle framework [13]. This considerable difference in code size may also be due to ψ-calculi explicitly handling computation and conditional execution, but the reason that the Þ-calculus does not have to explicitly deal with these features is also because of its use of HOAS.

in particular the π-calculus [3] and ψ-calculi [4]. Unlike our formalization of the Þ-calculus, those formalizations do not use HOAS but Nominal Isabelle [13] for dealing with name binding. It appears that this makes them more complex than the Þ-calculus formalization, although one has to consider that they use version 1 of Nominal Isabelle, not the improved version 2. Furthermore, the formalizations by Bengtson et al. suffer from considerable repetition, for example in their handling of strong and weak bisimilarity. The Þ-calculus formalization, on the other hand, makes more use of abstractions and achieves more code reuse this way. Another, albeit minor, advantage of the Þ-calculus formalization is its use of the *coinduction* proof method. The above-mentioned formalizations of the π-calculus and ψ-calculi use the less powerful *coinduct* method, resulting in more boilerplate code. Finally, "up to" methods help to avoid repetitive, technical proof code on a large scale, which we leverage in the Þ-calculus formalization and the developments built on it, using our formalized algebra of "up to" methods. The formalizations by Bengtson et al. also make use of "up to" methods, but the authors have only proved the soundness of a few specific methods. That said, Åman Pohjola and Parrow have developed a framework for "up to" methods for ψ-calculi [11].

7 Conclusion

We have presented a transition system semantics for the Þ-calculus, which is a general-purpose process calculus embedded in Isabelle/HOL, and derived from it a transition system semantics for a custom network communication language, which is embedded in the Þ-calculus. Building on this foundation and based on an example related to network communication, we have shown a way of conducting bisimulation proofs such that they become concise and human-friendly, while being machine-checked at the same time. Our proving style stems from combining the Isabelle/Isar proof language, an algebra of "up to" methods formalized in Isabelle/HOL, Isabelle's *coinduction* proof method, and higher-order abstract syntax.

8 Ongoing and Future Work

As mentioned in Section 4, we have proved most of the lower-level bisimilarity statements on which our broadcast equivalence proof [8] relies. At the moment, we are completing the last proofs of such statements.

In accordance with our research program mentioned in Section 1, we plan to verify further design refinement steps that the consensus protocols of the Ouroboros family have undergone. The refinement step we want to tackle next is the replacement of whole-chain distribution with a protocol for updating chains incrementally. Furthermore, we want to add some missing bits, in particular documentation, to the formalization of the Þ-calculus and the algebra of "up to" methods and submit both formalizations to Isabelle's Archive of Formal Proofs (AFP)[7].

Acknowledgements

This work was funded by Input Output. We are thankful to Input Output for giving us the opportunity to work on numerous interesting topics, including the one described in this paper. Furthermore, we want to thank the anonymous reviewers and James Chapman for their various suggestions for improvement of this paper.

[7] See `https://www.isa-afp.org/`.

References

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger & David Walker (2014): *NetKAT: Semantic Foundations for Networks*. In: *Proceedings of the* 41*st ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, ACM, New York, pp. 113–126, doi:10.1145/2535838.2535862.

[2] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell & Vassilis Zikas (2018): *Ouroboros Genesis: Composable Proof-of-Stake Blockchains with Dynamic Availability*. In: *Proceedings of the* 2018 *ACM SIGSAC Conference on Computer and Communications Security*, ACM, New York, pp. 913–930, doi:10.1145/3243734.3243848.

[3] Jesper Bengtson & Joachim Parrow (2009): *Formalising the π-Calculus Using Nominal Logic*. *Logical Methods in Computer Science* 5(2), pp. 1–36, doi:10.2168/LMCS-5(2:16)2009.

[4] Jesper Bengtson, Joachim Parrow & Tjark Weber (2016): *Psi-Calculi in Isabelle*. *Journal of Automated Reasoning* 56(1), pp. 1–47, doi:10.1007/s10817-015-9336-2.

[5] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu & Dmitriy Traytel (2014): *Truly Modular (Co)datatypes for Isabelle/HOL*. In Gerwin Klein & Ruben Gamboa, editors: *Interactive Theorem Proving*, *Lecture Notes in Computer Science* 8558, Springer, Berlin/Heidelberg, Germany, pp. 93–110, doi:10.1007/978-3-319-08970-6_7.

[6] Bernardo David, Peter Gaži, Aggelos Kiayias & Alexander Russell (2018): *Ouroboros Praos: An Adaptively-Secure, Semi-Synchronous Proof-of-Stake Blockchain*. In Jesper Buus Nielsen & Vincent Rijmen, editors: *Advances in Cryptology – EUROCRYPT* 2018, *Lecture Notes in Computer Science* 10821, Springer, Berlin/Heidelberg, Germany, pp. 66–98, doi:10.1007/978-3-319-78375-8_3.

[7] Kohei Honda & Mario Tokoro (1991): *An Object Calculus for Asynchronous Communication*. In Pierre America, editor: *ECOOP* '91 *European Conference on Object-Oriented Programming*, *Lecture Notes in Computer Science* 512, Springer, Berlin/Heidelberg, Germany, pp. 133–147, doi:10.1007/BFb0057019.

[8] Wolfgang Jeltsch & Javier Díaz (2022): *Correctness of Broadcast via Multicast: Graphically and Formally*. *Electronic Proceedings in Theoretical Computer Science* 369, pp. 37–50, doi:10.4204/EPTCS.369.

[9] Aggelos Kiayias, Alexander Russell, Bernardo David & Roman Oliynykov (2017): *Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol*. In Jonathan Katz & Hovav Shacham, editors: *Advances in Cryptology – CRYPTO* 2017, *Lecture Notes in Computer Science* 10401, Springer, Berlin/Heidelberg, Germany, pp. 357–388, doi:10.1007/978-3-319-63688-7_12.

[10] Frank Pfenning & Conal Elliott (1988): *Higher-Order Abstract Syntax*. In: *Proceedings of the ACM SIGPLAN* 1988 *Conference on Programming Language Design and Implementation*, ACM, New York, pp. 199–208, doi:10.1145/53990.54010.

[11] Johannes Åman Pohjola & Joachim Parrow (2016): *Bisimulation Up-to Techniques for Psi-Calculi*. In: *Proceedings of the* 5*th ACM SIGPLAN Conference on Certified Programs and Proofs*, ACM, New York, pp. 142–153, doi:10.1145/2854065.2854080.

[12] Anu Singh, C. R. Ramakrishnan & Scott A. Smolka (2010): *A Process Calculus for Mobile Ad Hoc Networks*. *Science of Computer Programming* 75(6), pp. 440–469, doi:10.1016/j.scico.2009.07.008.

[13] Christian Urban (2008): *Nominal Techniques in Isabelle/HOL*. *Journal of Automated Reasoning* 40(4), pp. 327–356, doi:10.1007/s10817-008-9097-2.

[14] Andreas Voellmy & Paul Hudak (2011): *Nettle: Taking the Sting Out of Programming Network Routers*. In Ricardo Rocha & John Launchbury, editors: *Practical Aspects of Declarative Languages*, *Lecture Notes in Computer Science* 6539, Springer, Berlin/Heidelberg, Germany, pp. 235–249, doi:10.1007/978-3-642-18378-2_19.

[15] Makarius Wenzel (2022): *The Isabelle/Isar Reference Manual*. `https://isabelle.in.tum.de/dist/Isabelle2022/doc/isar-ref.pdf`.

Comprehensive Specification and Formal Analysis of Attestation Mechanisms in Confidential Computing

Muhammad Usama Sardar[1], Thomas Fossati[2] and Simon Frost[2]

[1]TU Dresden
[2]Arm Ltd.

Confidential Computing (CC) using hardware-based Trusted Execution Environments (TEEs) has emerged as a promising solution for protecting sensitive data in all forms. One of the fundamental characteristics of such TEEs is remote attestation [4] which provides mechanisms for securely measuring and reporting the state of the remote platform and computing environment to a user. We present a novel approach combining TEE-agnostic attestation architecture and formal analysis enabling comprehensive and rigorous security analysis of attestation mechanisms in CC. We demonstrate the application of our approach for three prominent industrial representatives, namely Arm Confidential Compute Architecture (CCA) [1, 2] in architecture lead solutions, Intel Trust Domain Extensions (TDX) [5] in vendor solutions, and Secure CONtainer Environment (SCONE) [3] in frameworks. For each of these solutions, we provide a comprehensive specification of all phases of the attestation mechanism in confidential computing, namely provisioning, initialization, and attestation protocol. Our approach reveals design and security issues in Intel TDX and SCONE attestation. The work is currently under submission at another venue with formal proceedings [6].

References

[1] Arm Ltd.: Arm Realm Management Extension (RME) System Architecture. Tech. rep. (2021-2022), https://developer.arm.com/documentation/den0129

[2] Arm Ltd.: Realm Management Monitor specification. Tech. rep. (2021-2022), https://developer.arm.com/documentation/den0137

[3] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'keeffe, D., Stillwell, M.L., et al.: SCONE: Secure linux containers with Intel SGX. In: USENIX Symposium on Operating Systems Design and Implementation. pp. 689–703 (2016), https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[4] Guttman, J.D., Ramsdell, J.D.: Understanding Attestation: Analyzing Protocols that Use Quotes. In: Security and Trust Management, pp. 89–106 (2019), http://link.springer.com/10.1007/978-3-030-31511-5_6

[5] Intel: Intel ® Trust Domain Extensions (aug 2021), https://cdrdv2.intel.com/v1/dl/getContent/690419

[6] Sardar, M.U., Fossati, T., Frost, S.: SoK: Attestation in Confidential Computing (2023), https://www.researchgate.net/publication/367284929_SoK_Attestation_in_Confidential_Computing

Type qualifier inference and code synthesis for a better data-centric synchronisation experience

Ana Almeida Matos
Instituto de Telecomunicações
Instituto Superior Técnico
University of Lisbon
ana.matos@tecnico.ulisboa.pt

Jan Cederquist
Instituto de Telecomunicações
Instituto Superior Técnico
University of Lisbon
jan.cederquist@tecnico.ulisboa.pt

Marco Giunti
NOVA LINCS
marco.giunti@gmail.com

João Matos
Instituto Superior Técnico
University of Lisbon
joao.r.matos@tecnico.ulisboa.pt

Hervé Paulino
NOVA LINCS
NOVA School of Science and Technology
herve.paulino@fct.unl.pt

António Ravara
NOVA LINCS
NOVA School of Science and Technology
aravara@fct.unl.pt

Despite the advantages of Data-Centric Synchronisation (DCS), a high-level declarative approach that abstracts away from the actual concurrency control mechanism(s) by means of data atomicity declarations, its practical use in existing solutions is hindered by verbose coding requirements and the lack of support for interfaces required in most object-oriented programs. AtomiS is a new DCS model that requires only an *atomicity specification* in interfaces and fields, and automatizes generation of specification consistent code that covers the multiple method variants that cope with differently qualified parameters. The model is rigorously defined for a type-sound programming language, a type qualifier inference (atomicites) based on a new type directed constraint solving algorithm, and a transpilation methodology that, when the analysis described is successful, produces fully qualified code, with all atomicities sorted out. In this paper we present the foundations for the AtomiS analysis stage, developed over OOlong, and formal guarantees that the generated program is well-typed and that it corresponds behaviourally to the original one. The proofs are mechanised in Coq. The ultimate goal of AtomiS is to guarantee the absence of atomicity violations. We formulate an abstract semantic property capturing that critical result, which depends on the posterior lock injection stage.

1 Introduction

The de-facto standard when programming concurrent applications is to use shared memory and control interference by identifying the critical sections and managing access to them via synchronisation mechanisms like locks, monitors, or semaphores. This identification of regions of code that should be executed in mutual exclusion has proven time and again to be quite difficult.

Data-Centric Synchronisation (DCS) is a high-level declarative approach that shifts reasoning about concurrency restrictions from control structures to data declaration, abstracting away from the actual concurrency control mechanism(s) in use. In short, one simply needs to identify critical resources to protect from interference, instead of code portions. Despite its advantages, the practical use of DCS is

Submitted to:
ICE 2023

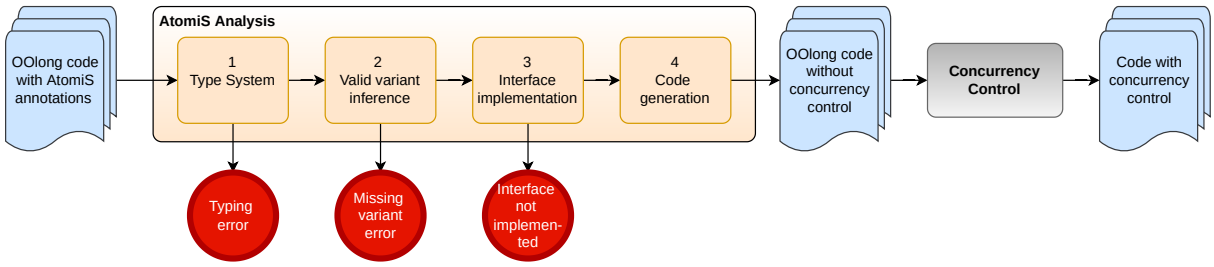

Figure 1: ATOMIS compilation stages

hindered by the fact that it may require many annotations and/or multiple implementations of the same method to cope with differently qualified parameters. To overcome these limitations, we have developed ATOMIS, a new DCS model based on a rigorously defined type-sound programming language.

Programming with ATOMIS requires only (`atomic`)-qualifying types of parameters and return values in interface definitions, and of fields in class definitions. From this *atomicity specification*, a static analysis infers the atomicity constraints that are local to each method, considering valid only the method variants that are consistent with the specification, and performs code generation for all valid variants of each method. The generated code then undergo automatic injection of concurrency control primitives, by plugging into the models pipeline the desired automatic technique. The entire model has been implemented for Java, using a lock-injection algorithm that is inspired in Autolocker [14], and which guarantees the expected thread safety properties of absence of atomicity violations, of deadlocks and of data races.

Why do we propose an oral communication? The main contribution of our work, presented in detail in a technical report[1] and summarised herein, is the *formal foundations of* ATOMIS, a new DCS language-based approach which requires only atomicity specifications, inferring statically the atomicity concerns local to each method. The code generated includes atomic versions of the source code's types and, for all classes, the code of all valid method variants. Crucially, our approach relies on type qualifiers for specifying the target resources for the concurrency control mechanism.

As usual for type qualifiers, atomicity annotations do not have an impact in the operational semantics of the language, though they do represent a desired semantic property that excludes behaviour involving the access to protected resources. Since the change (mainly loss) in behaviour introduced by the concurrency control are to be introduced in a subsequent stage, we do not define a semantics for it. Instead, we define an abstract semantic property that should be enforced.

We are working on a clarification of the semantics of the atomic qualifiers in the form of a safety property that excludes programs containing traces that violate exclusivity of atomic accesses within a unit of work. A valid mechanism for lock injection must then be sound with respect to this property. The main goal of our communication is to seek feedback from the community regarding the approach we are pursuing.

Roadmap. If our oral communication proposal is accepted, its technical content is as follows.

1. The presentation of a rigorous formalisation of the ATOMIS analysis in a type sound-core language OOlong (in §3), with formal guarantees of type and behavioural soundness (in §4). A mechanised proof in Coq has been developed for the former [2], and for the latter it is underway.

[1] `https://drive.proton.me/urls/1EFCXGE14W#iNe2lXovYmkO`

[2] `https://zenodo.org/record/6382015` and `https://zenodo.org/record/6346649`

2. A type qualifier inference methodology that enables to infer complete atomicity qualifiers from an interface and field-based specification, where the two possible qualifiers (atomic and non-atomic) are not related by subtyping. This methodology also determines the method variants for all possible parameter and return type qualifiers that are valid, i.e., are consistent with the specification (in §3).

3. A program synthesis methodology, based on the solved atomicity and valid variant inference, that produces fully qualified code from all classes and corresponding valid method variants (in §3).

4. A formulation of an abstract semantic property excluding atomicity violations, based on the notion of *unit of work* [7] and of *exclusivity of atomic units of work* (in §2), which can be made concrete for a specific lock inference algorithm, and provide the bases for ensuring thread safety.

2 The AtomiS Model

AtomiS is a generic data-centric synchronisation model applicable to any concurrent language with shared state. The data-centrality comes from the addition of the **atomic** type qualifier to variables whose values are shared across multiple concurrent execution flows, similar to what is found in C++ [12]. We do this using an **atomic** annotation, applicable to any type.

Mutable values assigned to variables with atomic type are referred to as *atomic values*. Atomic values may only be manipulated within the scope of a *unit of work* [7], which in AtomiS translates to blocks of instructions (such as method bodies) that access at least one atomic value. Atomic values accessible from the same unit of work implicitly share consistency requirements. The goal of AtomiS is to ensure that all accesses to atomic values within each unit of work are, in effect, a single atomic operation.

Atomicity Specification. Concurrency restrictions are specified in the types found in interface and class definitions. In the case of interfaces, each method specifies what combinations of atomicities are supported in its parameter and return type, with sequences of elements $(q_1) \rightarrow q_2$, where q_1 denotes the atomicity of the parameter and q_2 the atomicity of the return type. The type in class instantiation may also be atomic-qualified.

Consider the following interface for lists of atomic objects. We want to specify that the methods support the insertion and retrieval of atomic values.

```
interface ListAtomic {
 add(element : Object) : Unit [(atomic) → atomic, (atomic) → non_atomic]
 get(pos : Integer) : Object [(non_atomic) → atomic, (atomic) → atomic]
 equals(other : List) : Boolean [(non_atomic) → non_atomic, (non_atomic) → atomic, (atomic) → non_atomic,
    (atomic) → atomic]
}
```

`add` takes an atomic parameter. `get` takes either atomic or non-atomic `Integers` and returns a value of atomic type. Lastly, in `equals`, we want to be able to compare the contents of the current list with any other, atomic or not, and the result must be assignable to variables of both atomic and regular types, so all combinations must be supported.

Regarding class implementations, class fields may be accessed by multiple methods which may, in turn, be executed by multiple threads. We thus require field atomicity to be specified in the source code. The following implementation of `ListOfAtomic` supports concurrent access to list nodes, so they are specified as atomic. The nodes are implemented as instances of class `NodeOfAtomic`, which stores atomic objects.

```
1 class NodeOfAtomic {
2  atomic Object value;
3  NodeOfAtomic next, prev;
4  }
```

```
class ConcurrentListOfAtomic implements ListOfAtomic {
 atomic NodeOfAtomic head, tail;
 void add(T element) { ... }
 ...
}
```

Listing 1: A concurrent list example.

Atomic Types and Method Variants. Atomic-qualified (*atomic*, for short) and regular types define two different type trees, and thus are not convertible into each other. Equivalently, if a class *C* extends a class *D* then **atomic** *C* will extend **atomic** *D*. As a result, an object with atomic type cannot be cast into a regular type, and, by this way, give rise to uncontrollable atomicity violations.

Concerning atomic class types, a decision must be made about the atomicity of unqualified fields of the same type of the hosting class, such as field `next` from class `NodeOfAtomic` in Listing 1. The atomicity of such fields may be inherited from the class itself, preserving the type equality between the field and the hosting class, or simply remain unaltered, breaking this equality. Currently, we chose to preserve the equality. Ergo, in Listing 1, the type of `head.next` is **atomic** `NodeOfAtomic`, allowing for the `NodeOfAtomic` type to be used both in the implementation of both concurrent and regular lists of atomic objects.

The atomicity of parameters types must be matched, for each method call, with the atomicity of the types of the values assigned to them. The atomicity of the types of a method's local variables and the return type of that method may depend on the atomicity of the types of the values that have been (or will be later) assigned to a field or to a parameter. A method may hence have several different signatures, what we refer to as *method variants*, corresponding to combinations of atomicity qualifiers for the object itself (**this**), the method's parameters and the return type. For each variant, the atomicity of all non-qualified local variables is inferred during the compilation process, having as base the atomicity of fields and the given atomicity combination (more details in §3). Accordingly, some variants may not be type-safe.

Although our approach shares affinities with the *context-sensitive field-based* type qualifier inference presented in [10], in our work it is necessary to separate regular from atomic types, rather than defining a partial-order between them in order to prevent atomicity violations.

Unit of Work. The role of the atomicity specification in ATOMIS is to inform the compilation process, with the ultimate aim of enforcing exclusive access to atomic values. The Analysis stage thus clarifies *which values are to be treated atomically*, and in the Concurrency Control stage a mechanism is introduced to enforce the property over units of work. We say that a thread is executing within a *unit of work* when it is executing instructions in between the first and the last access that is performed by a method call on an atomic value. The work in this paper does not comprise this latter stage, including the algorithm for enforcing units of work. We can however define the property abstractly.

Let *unit of work context* of an executing thread refer to the set of atomic values that correspond to the unit within which the thread is executing, or the empty set if that is not the case. Unit of work context is defined as expected for thread collections, as the union of unit of work contexts of all threads. We must ensure that thread collections respect atomicity of units of work throughout their computation. We are specifically interested in whether the units of work of atomic values are entered *exclusively* by a single thread at all times. We say that a thread collection guarantees *exclusivity of units of work* with respect to a set of atomic values if it is single-threaded, or if the unit of work contexts of all threads are disjoint (their intersection is empty) and each thread has also exclusivity of units of work with respect to the same set. We can then define the operational property of a thread collection respecting exclusivity of atomic units of work as simply to require that exclusivity be preserved by the execution of programs in the language.

3 ATOMIS Analysis

The purpose of ATOMIS' static analysis is to infer all atomic-qualifiers of the program's types, and to consistently synthesise the code for the valid variants of every method.

ATOMIS-OOlong. The correctness of the produced program is crucial for ensuring the thread safety properties of the lock-injection stage that follows. To formally prove the soundness of the analysis we build on OOlong [2], a principled object-oriented multi-threaded programming calculus with minimal syntax, a formal static and dynamic semantics, and a computer-aided proof of type soundness, on which our mechanisation builds. ATOMIS-OOlong extends OOlong to support the declaration of atomic fields and the instantiation of atomic objects, via the **atomic** keyword, and signature annotations, which are the only constructs to express concurrency restrictions. As the output of the analysis is an Oolong program, we omit the presentation of its semantics here. [3]

Analysis stages. The analysis of an ATOMIS-OOlong program P_{orig} produces an OOlong program (without locks) with safe code (*i.e.*, type-safe and preventing atomicity violations), that determines the atomicity qualification of every type in P_{orig}. For each type in P_{orig}, the generated code includes its regular and atomic versions. Moreover, the class types have their original methods unfolded into the corresponding valid variants, and all method calls are explicitly resolved into a valid variant. The complete analysis of an original program is a partial function, denoted AtomiSAnalysis, that comprises four stages (Fig. 1).

Stage 1 - Check Well-formedness. The analysis requires underlying OOlong programs to be well-typed. The code, stripped from all **atomic** and interface signature annotations, is submitted to an instrumented version of the original OOlong type system [2]: the original verification includes program well-formedness and the matching of the types of the various components of the program.

Stage 2 - Solve Atomicities and Validity of Method Variants. This is a three step process that outputs the *solution* – a model of a constraint system over *variant* and *atomicity variables*, where *validity values* are assigned to variant variables and *atomicity values* are assigned to atomicity variables.

First, a **type-based generation of atomicity-related information** produces a map that, for each method of the source program, provides constraints on the atomicity of the method's local variables, and the set of the method variants that the method will call in its execution. Constraints are defined over *atomicity variables*, which represent yet unresolved atomicities – such as of variables, of method parameters, of the result of method calls, or of objects – and *validity variables* – enabling to express atomicity requirements for a variant to be valid.

Next, the generated atomicity information is used to **build a global constraint system** with the conditions for the inference to be successful. The system comprises a validity constraint per every possible variant of each method, and for each call (denoted by a variant variable). The resulting constraints are conjugated to ensure satisfiability of all calls. Furthermore, the validity of different method variants needs to be checked for satisfiability of the restrictions imposed by the method's body, including calls, plus those imposed (by the variant) on the atomicity of the object, parameter and return value.

Last, the solver is called on **the global constraint system** which only uses equality, implication, conjunction and disjunction operations over two sets of binary values, *i.e.*, a Boolean Satisfiability problem. The model resulting from the system's satisfiability assigns Boolean values to all validity and atomicity variables, producing the aforementioned solution.

[3] The interested reader can check Section 4, namely Figures 8 and 9, of OOlong main reference [2].

Stage 3 - Check Interface Implementation. An ATOMIS-OOlong interface may feature signature annotations to explicitly convey the atomicity of method parameters and return types. This stage aims at guaranteeing that the set of valid variants computed for each class C includes the signatures that result from the parsing of the interface I implemented by C. To that end, this stage receives the solution – to retrieve the signatures of the valid variants of the class' methods – and the source code – to retrieve the signatures originally defined in the program's classes and interfaces. Having both sets of signatures, the stage simply checks if the former includes the latter.

Stage 4 - Generate Code. This stage consists of a syntax directed code transformation that replaces: 1 - original method signatures in interfaces by the ones resulting from the parsing of the atomicity annotations; 2 - method definitions in classes by the variants considered valid in the solution generated by Stage 2; 3 - method names by the right variant in all method calls; and 4 - types and qualifiers by new types from a set that is implicitly partitioned into atomic and no-atomically qualified types. The resulting code has thus no ambiguities with regard to variable atomicity.

4 Soundness

The AtomiSAnalysis process guarantees type and behavioural soundness, *i.e.*, that (when successful) it preserves typeability and atomicity annotations and produces a program behaviourally equivalent to the original one. Definitions and results are presented informally here.[4] Type soundness of the automatically generated program has been proved in Coq.[5]

Type Soundness. The results are stated for programs P_{orig}, *i.e.*, ATOMIS-OOlong programs with atomicity annotations, for which the analysis succeeds and produces a final OOlong program P^{+}.

Preservation of Base Types: States that if P_{orig}, stripped from atomicity annotations, is typeable with type t, then the final program P^{+} is also typeable with a compound of an atomicity ν and type t.

Consistency of Types with Atomicity Annotations: Further guarantees that the field and signature's types of the final program P^{+} are consistent with those in P_{orig}, i.e, that types given to fields (respectively, to methods) in P^{+} are a compound of the atomicity and types of the fields (respectively, of the methods) given to fields (respectively, to methods) with the corresponding name in corresponding classes of P_{orig}.

Note that the Progress and Preservation results that hold for the OOlong language and type system ensure that *the output of the analysis never goes wrong* in what regards both base types and atomicities.

Behavioural Soundness. Our approach performs a program transformation that consistently fleshes out the atomicity qualifiers of every type in the program, while unfolding classes and methods according to their determined qualified types. We have proven that the ATOMIS Analysis (up until but not including lock injection) does not affect the original behaviour, *i.e.*, that the final program does everything the original one does, and nothing more, according to a notion of indistinguishability that is based on a bisimulation[6]. To this end we: (1) define syntactic correspondences mapping types and method names occurring in the interfaces and classes of the final program, to those from which they originated in the initial code; and (2) focus on *heap correspondences* between two heapswith the same domain, and assign, in corresponding classes, the same field map and lock status to all references.

[4] Their full and precise formulations are provided in the Technical Report (2023); please see the supplementary material.

[5] AtomiS-Coq proof of type preservation (2022), `https://zenodo.org/record/6346649`, `https://zenodo.org/record/6382015`

[6] Theorem 14 in the Technical Report in the supplementary material.

We design a bisimulation relation to relate thread collections that derive from programs between which there is a syntactic correspondence, and which preserve a heap correspondence throughout all possible execution paths. Our operational soundness result establishes that *when a program undergoes the* ATOMIS *Analysis, the original and the final programs are bisimilar, i.e., exhibit the same behaviour with respect to the heap correspondence*.

5 Related Work

DCS in shared memory programming. *Atomic Sets* [7, 18] is a reference work in the area. Variables holding values with consistency properties must be placed in an *atomic* set. Sets may have multiple units of work, which can be explicitly augmented to account for multiple method parameters. Likewise, alias annotations enable the union of sets from distinct classes at object creation. Although a seminal work, Atomic Sets annotations may hamper reasoning and are error-prone, as some may be easily forgotten. AWJS [13] combines Atomic Sets with work-stealing-based task parallelism in the Java language. Other works that perform automatic inference of atomic sets include: AJ-lite [11], a lighter version that assumes a single atomic set per Java class and is only applicable to libraries and not entire programs; [6], which is able to automatically infer most atomic sets from patterns recognised in execution traces, although sensitive to the quality of the input traces, and may generate more annotations than necessary.

Ceze *at al.* [4] associate variables with consistency proprerties to a colour, defining a consistency domain. However, concurrency control is not centralised on data declaration. Code annotations have to be added to the methods' implementations. Moreover, atomic-region-like control-centric concurrency control is needed to handle *high-level data races* [1] in composite operations. In [3], some of the same authors proposed hardware support for data-centric synchronisation.

RC^3 [17] is a DCS model that uses a single keyword (**atomic**), being a source of inspiration for ATOMIS. Most of the concerns delegated on the programmer in AJ are shifted to static analyses. However, the methods' implementations are atomicity-aware, requiring the duplication of code to account for the atomicities of the parameters. As in all other DCS solutions, there is no support for interfaces.

Type qualifier inference and Type-directed code synthesis. Generic frameworks and tools for type qualifier inference have been proposed, *e.g.* CQUAL [8, 9] and CLARITY [5] for C, and JQual [10] for Java, and their usage experimented for different purposes (see the latter for a review). The ATOMIS model performs a field-based, context-sensitive, flow-insensitive, analysis, as JQual, but for specifically inferring atomicity type qualifiers to achieve strong concurrency control guarantees. Imposing program-wide atomicity constraints on how data is manipulated, in a flow-insensitive approach in the presence of alias appears to be incompatible with enabling subtyping between the atomic and non-atomic qualifiers. Indeed, the same object should not be treated simultaneously as atomic and non-atomic by different parts of the program. Automatic generation of method variants by the ATOMIS model provides flexibility compatible with the methods' code, up to the conservative nature of the constraint generation.

Our code generation approach bears connections with Osera's [16] constraint-based type-directed program synthesis technique for producing polymorphic code from types and *examples*. The main differences with respect to our approach are that [16] targets a functional programming language, and that ATOMIS's code generation is based on an original program.

6 Conclusions

We propose herein the formal foundations of a sound new model of data-centric synchronisation, that only requires the annotation of interfaces and of atomic class fields. The approach has two main phases — an analysis, which, when successful, infers missing annotations and produces type-safe atomicity-related variants of each method; followed by a lock injection phase, to manage concurrency and prevent interferences. In this paper we present the formalisation of the analysis phase, for a core object calculus (OOlong), showing that the generated code is type-safe and behaviourally corresponds to the original one. We provide a computer-verified proof that the resulting generated program is type-safe, using the Coq proof assistant. We are developing a mechanised proof of ATOMIS's behavioural soundness.

Up to the formalised stage, the work is largely agnostic to the semantics of atomic annotations. It is in subsequent stages that the program structures that actually protect atomic resources are injected into the code, thus shaping the semantics and properties of the final program. We have nevertheless proposed an abstract formulation for the crucial property of mutual exclusion that is based on the notion of unit of work. While the computation of units of work and lock inference has been implemented, we are working on its formalisation, and aim to prove its soundness.

Acknowledgement. This work is supported by NOVA LINCS (UIDB/04516/2020) with the financial support of FCT.IP.

References

[1] Cyrille Artho, Klaus Havelund & Armin Biere (2003): *High-Level Data Races*. In Pedro T. Isaías, Florence Sèdes, Juan Carlos Augusto & Ulrich Ultes-Nitsche, editors: *New Technologies for Information Systems, Proceedings of the 3rd International Workshop on New Developments in Digital Libraries, NDDL 2003, and the 1st International Workshop on Validation and Verification of Software for Enterprise Information Systems, VVEIS 2003, In conjunction with ICEIS 2003, Angers, France, April 2003*, ICEIS Press, pp. 82–93.

[2] Elias Castegren & Tobias Wrigstad (2019): *OOlong: A Concurrent Object Calculus for Extensibility and Reuse*. *SIGAPP Appl. Comput. Rev.* 18(4), pp. 47–60, doi:10.1145/3307624.3307629.

[3] Luis Ceze, Pablo Montesinos, Christoph von Praun & Josep Torrellas (2007): *Colorama: Architectural Support for Data-Centric Synchronization*. In: *13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*, IEEE Computer Society, pp. 133–144, doi:10.1109/HPCA.2007.346192. Available at `https://ieeexplore.ieee.org/xpl/conhome/4147635/proceeding`.

[4] Luis Ceze, Christoph von Praun, Calin Cascaval, Pablo Montesinos & Josep Torrellas (2008): *Concurrency control with data coloring*. In Emery D. Berger & Brad Chen, editors: *Proceedings of the 2008 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08), Seattle, Washington, USA, March 2, 2008*, ACM, pp. 6–10, doi:10.1145/1353522.1353525.

[5] Brian Chin, Shane Markstrum, Todd Millstein & Jens Palsberg (2006): *Inference of User-Defined Type Qualifiers and Qualifier Rules*. In: *IN PROC. ESOP*, pp. 264–278.

[6] Peter Dinges, Minas Charalambides & Gul Agha (2013): *Automated inference of atomic sets for safe concurrent execution*. In Stephen N. Freund & Corina S. Pasareanu, editors: *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '13, Seattle, WA, USA, June 20, 2013*, ACM, pp. 1–8, doi:10.1145/2462029.2462030. Available at `http://dl.acm.org/citation.cfm?id=2462029`.

[7] Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri & Jan Vitek (2012): *A data-centric approach to synchronization*. *ACM Trans. Program. Lang. Syst.* 34(1), pp. 4:1–4:48, doi:10.1145/2160910.2160913.

[8] Jeffrey S. Foster, Manuel Fähndrich & Alexander Aiken (1999): *A Theory of Type Qualifiers*. In Barbara G. Ryder & Benjamin G. Zorn, editors: *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*, ACM, pp. 192–203, doi:10.1145/301618.301665.

[9] Jeffrey S. Foster, Robert Johnson, John Kodumal & Alex Aiken (2006): *Flow-insensitive type qualifiers*. *ACM Trans. Program. Lang. Syst.* 28(6), pp. 1035–1087, doi:10.1145/1186635.

[10] David Greenfieldboyce & Jeffrey S. Foster (2007): *Type qualifier inference for java*. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes & Guy L. Steele Jr., editors: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, ACM, pp. 321–336, doi:10.1145/1297027.1297051.

[11] Wei Huang & Ana Milanova (2012): *Inferring AJ Types for Concurrent Libraries*. 19th International Workshop on Foundations of Object-Oriented Languages, FOOL 2012, Tucson, AZ, USA; October 22, 2012.

[12] (2011): *C++ 11 standard*. `https://www.iso.org/standard/50372.html`. Section 6.7.2.4 - Atomic type specifiers.

[13] Vivek Kumar, Julian Dolby & Stephen M. Blackburn (2016): *Integrating Asynchronous Task Parallelism and Data-centric Atomicity*. In Walter Binder & Petr Tuma, editors: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*, ACM, pp. 7:1–7:10, doi:10.1145/2972206.2972214.

[14] Bill McCloskey, Feng Zhou, David Gay & Eric A. Brewer (2006): *Autolocker: synchronization inference for atomic sections*. In Morrisett & Jones [15], pp. 346–358, doi:10.1145/1111037.1111068. Available at `http://dl.acm.org/citation.cfm?id=1111037`.

[15] J. Gregory Morrisett & Simon L. Peyton Jones, editors (2006): *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. ACM. Available at `http://dl.acm.org/citation.cfm?id=1111037`.

[16] Peter-Michael Osera (2019): *Constraint-based type-directed program synthesis*. In David Darais & Jeremy Gibbons, editors: *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2019, Berlin, Germany, August 18, 2019*, ACM, pp. 64–76, doi:10.1145/3331554.3342608.

[17] Hervé Paulino, Daniel Parreira, Nuno Delgado, António Ravara & Ana Gualdina Almeida Matos (2016): *From atomic variables to data-centric concurrency control*. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ACM, pp. 1806–1811, doi:10.1145/2851613.2851734.

[18] Mandana Vaziri, Frank Tip & Julian Dolby (2006): *Associating synchronization constraints with data in an object-oriented language*. In Morrisett & Jones [15], pp. 334–345, doi:10.1145/1111037.1111067. Available at `http://dl.acm.org/citation.cfm?id=1111037`.