EPTCS 58

Proceedings of the

10th International Workshop on the Foundations of Coordination Languages and Software Architectures

Aachen, Germany, 10th September, 2011

Edited by: Mohammad Reza Mousavi and Antonio Ravara

27th July 2011
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Foreword

Mohammad Reza Mousavi
Department of Computer Science, Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

António Ravara
Department of Informatics
Faculty of Sciences and Technology, New University of Lisbon

Welcome to the proceedings of FOCLASA 2011, the 10th International Workshop on the Foundations of Coordination Languages and Software Architectures. FOCLASA 2011 was held in Aachen, Germany on September 10th, 2011 as a satellite event of CONCUR 2011, the 22nd International Conference on Concurrency Theory.

The workshop provides a venue where researchers and practitioners could meet, exchange ideas, identify common problems, determine some of the key and fundamental issues related to coordination languages and software architectures, and explore together and disseminate solutions. Indeed, a number of hot research topics are currently sharing the common problem of combining concurrent, distributed, mobile and heterogeneous components, trying to harness the intrinsic complexity of the resulting systems. These include coordination, peer-to-peer systems, grid computing, web services, multi-agent systems, and component-based systems. Coordination languages and software architectures are recognized as fundamental approaches to tackle these issues, improving software productivity, enhancing maintainability, advocating modularity, promoting reusability, and leading to systems more tractable and more amenable to verification and global analysis.

This year, we received ten submissions involving 33 authors from 12 different countries. Papers underwent a rigorous review process, and all accepted papers received 4 review reports. After the review process, the international Program Committee of FOCLASA 2010 decided to select five papers for presentation during the workshop and inclusion in these proceedings. These papers tackle different issues that are currently central to our community, specification and reasoning frameworks parallel and concurrent systems, systems with linked data, resource-constrained and timed systems and data-flow coordination models. The workshop features an invited speech by Joe Armstrong from Ericsson, Sweden. The best papers of the workshop will be invited for a special issue in Science of Computer Programming (Elsevier). We would like to thank all the members of the program committee for their great work during the review process, the external reviewers for providing insightful review reports, the authors for submitting papers to the workshop, and the participants for attending the workshop in Aachen. All these people contribute to the success of the 2011 edition of FOCLASA.

Mohammad Reza Mousavi
Antonio Ravara

M.R. Mousavi and A. Ravara: FOCLASA 2011
EPTCS 58, 2011, pp. ii–iii, doi:10.4204/EPTCS.58.0

How Erlang views the world and what we have learned in the last 25 years of programming distributed systems.

Joe Armstrong
Ericsson Telecom AB, Stockholm, Sweden

Erlang views the world as a large collection of isolated communicating processes. These processes have no shared memory. During normal operation processes communicate by exchanging messages. Abnormal situations are handled through the exchange of signals. Signals and messages are fundamentally different. Erlang was originally designed for building “fault tolerant systems that run forever” which is the main reason for our insistence on process isolation. Erlang was invented long before the advent of cheap massive distributed systems and long before the multicore chip emerged. The first Erlang applications were in their nature distributed and built to run on clusters. Multicores are essentially “clusters on a chip” so what we learned from programming clusters carries naturally over to the world of multicores. This talk is about the lessons and pitfalls of building distributed systems in Erlang. What works and what doesn’t work and the possible reasons for this.

Biography Joe Armstrong is the principle inventor of the Erlang programming Language and coined the term ”Concurrency Oriented Programming”. He works for Ericsson where he developed Erlang and was chief architect of the Erlang/OTP system. Joe has a PhD in computer science from the Royal Institute of Technology in Stockholm (KTH), Sweden and is an expert in the construction of fault tolerant systems. He is the author of several books on Erlang.

M.R. Mousavi and A. Ravara: FOCLASA 2011
EPTCS 58, 2011, pp. iv–iv, doi:10.4204/EPTCS.58.0.1

Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca

Luca Aceto[1] Matteo Cimini[1] Anna Ingolfsdottir[1]
Arni Hermann Reynisson[1] Steinar Hugi Sigurdarson[1]
Marjan Sirjani[1,2]
[1]Reykjavik University, Iceland [2]University of Tehran, Iran

In this paper we propose an extension of the Rebeca language that can be used to model distributed and asynchronous systems with timing constraints. We provide the formal semantics of the language using Structural Operational Semantics, and show its expressiveness by means of examples. We developed a tool for automated translation from timed Rebeca to the Erlang language, which provides a first implementation of timed Rebeca. We can use the tool to set the parameters of timed Rebeca models, which represent the environment and component variables, and use McErlang to run multiple simulations for different settings. Timed Rebeca restricts the modeller to a pure asynchronous actor-based paradigm, where the structure of the model represents the service oriented architecture, while the computational model matches the network infrastructure. Simulation is shown to be an effective analysis support, specially where model checking faces almost immediate state explosion in an asynchronous setting.

1 Introduction

This paper presents an extension of the actor-based Rebeca language [22] that can be used to model distributed and asynchronous systems with timing constraints. This extension of Rebeca is motivated by the ubiquitous presence of real-time computing systems, whose behaviour depends crucially on timing as well as functional requirements.

A well-established paradigm for modelling the functional behaviour of distributed and asynchronous systems is the actor model. This model was originally introduced by Hewitt [8] as an agent-based language, and is a mathematical model of concurrent computation that treats *actors* as the universal primitives of concurrent computation [1]. In response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message it receives. Actors have encapsulated states and behaviour, and are capable of redirecting communication links through the exchange of actor identities. Different interpretations, dialects and extensions of actor models have been proposed in several domains and are claimed to be the most suitable model of computation for the dominating applications, such as multi-core programming and web services [9].

Reactive Objects Language, *Rebeca* [22], is an operational interpretation of the actor model with formal semantics and model-checking tools. Rebeca is designed to bridge the gap between formal methods and software engineers. The formal semantics of Rebeca is a solid basis for its formal verification. Compositional and modular verification, abstraction, symmetry and partial-order reduction have been investigated for verifying Rebeca models. The theory

M.R. Mousavi and A. Ravara: FOCLASA 2011
EPTCS 58, 2011, pp. 1–19, doi:10.4204/EPTCS.58.1

underlying these verification methods is already established and is embodied in verification tools [14, 21, 22]. With its simple, message-driven and object-based computational model, Java-like syntax, and a set of verification tools, Rebeca is an interesting and easy-to-learn model for practitioners.

Motivation and Contribution. Although actors are attracting more and more attention both in academia and industry, little has been done on timed actors and even less on analyzing timed actor-based models. In this work we present

- timed Rebeca by extending Rebeca with time constraints,
- the formal semantics of timed Rebeca using Structural Operational Semantics (SOS) [19],
- a tool for mapping timed Rebeca models to Erlang, and
- experimental results from the simulation of timed Rebeca models using McErlang [7].

The contribution of this work is offering a pure asynchronous actor-based modelling language with timing primitives and analysis support. Timed Rebeca can be used in a model-driven methodology in which the designer builds an abstract model where each component is a reactive object communicating through non-blocking asynchronous messages. The structure of the model can very well represent service oriented architectures, while the computational model matches the network infrastructure. Hence the model captures faithfully the behaviour of the system in a distributed and asynchronous world.

Comparison with other timed models. Comparing with the well-established timed models, like timed automata [2], TCCS [25], and real-time Maude [18], timed Rebeca offers an actor-based syntax and a built-in actor-based computational model, which restricts the style of modelling to an event-based concurrent object-based paradigm. Modelling time-related features in computational models has been studied for a long time [3, 2]; while we have no claims of improving the expressiveness of timed models, we believe that our model is highly usable due to its actor-based nature and Java-like syntax. The usability is due to the one to one correspondence between the entities of the real world and the objects in the model, and the events and actions of the real world and the computational model. Moreover, the syntax of the language is familiar for software engineers and practitioners.

Comparison with other timed actor models. We know of a few other timed actor-based modelling languages [20, 16, 4] that we will explain in more detail in the related work section. In [20] a central synchronizer acts like a coordinator and enforces the real-time and synchronization constraints (called interaction constraints). The language for the coordinated actors is briefly proposed in [16]; however, the main focus is having reusable real-time actors without hardwired interaction constraints. The constraints declared within the central synchronizer in this line of work can be seen as the required global properties of a timed Rebeca model. We capture the architecture and configuration of a system via a timed Rebeca model and then we can check whether the global constraints are satisfied. The language primitives that we use to extend Rebeca are consistent with the proposal in [16]. The primitives proposed in [4] are different from ours; they introduced an *await* primitive where we keep the asynchronous nature of the model.

Analysis support. In order to analyze timed Rebeca models, we developed a tool to facilitate their simulation. In a parallel project [11], a mapping from timed Rebeca to timed automata is developed and UPPAAL [24] is used for model checking. The asynchronous nature of Rebeca models causes state explosion while model checking even for small models. One solution is using a modular approach like in [12]. Here, we selected an alternative solution as a complementary tool for analysis. Using our tool we can translate a timed Rebeca model to Erlang [6], set the parameters which represent the environment and component variables, and run McErlang [7] to simulate the model. The tool allows us to change the settings of different timing parameters and rerun the simulation in order to investigate different scenarios, find potential bugs and problems, and optimize the model by manipulating the settings. The parameters can be timing constraints on the local computations (e.g., deadlines for accomplishing a requested service), computation time for providing a service, and frequency of a periodic event. Parameters can also represent network configurations and delays. In our experiments we could find timing problems that caused missing a deadline, or an unstable state in the system.

The formal semantics presented in this paper is the basis for the correct mapping from timed Rebeca to Erlang. The detailed mapping, and the tool together with some examples can be found at [10].

Our choice to use the actor-based programming language Erlang is also based on the idea of covering the whole life cycle of the system in future, and of providing a refinement step for implementing the code from our timed Rebeca model.

2 Related Work

Different approaches are used in designing formal modelling languages for real-time systems. The model of timed automata, introduced by Alur and Dill [2], has established itself as a classic formalism for modelling real-time systems. The theory of timed automata is a timed extension of automata theory, using clock constraints on both locations and transitions. In many other cases the proposed modelling languages for real-time systems are extensions of existing languages with real-time concepts—see, for example, TCCS [25] and Real-time Maude [18].

A real-time actor model, RT-synchronizer, is proposed in [20], where a centralized synchronizer is responsible for enforcing real-time relations between events. Actors are extended with timing assumptions, and the functional behaviours of actors and the timing constraints on patterns of actor invocation are separated. The semantics for the timed actor-based language is given in [16]. Two positive real-valued constants, called *release time* and *deadline*, are added to the *send* statement and are considered as the earliest and latest time when the message can be invoked relative to the time that the method executing the send is invoked. In Timed Rebeca, we have the constructs *after* and *deadline*, which are representing the same concepts, respectively, except that they are relative to the time that the message (itself) is sent. So, it more directly reflects the computation architecture including the network delays. In our language, it is also possible to consider a time *delay* in the execution of a computation where in [16] it is possible to specify an upper bound on the execution time of a method. While RT-synchronizer is an abstraction mechanism for the declarative specification of timing constraints over groups of actors, our model allows us to work at a lower level of abstraction. Using timed Rebeca, a modeller can easily capture the functional features of a system, together with the timing constraints for both computation and network latencies, and analyze the model from various

points of view.

There is also some work on schedulability analysis of actors [17], but this is not applied on a real-time actor language. Time constraints are considered separately. Recently, there have been some studies on schedulability analysis for Rebeca models [13]. This work is based on mapping Rebeca models to timed automata and using UPPAAL to check the schedulability of the resulting models. Deadlines are defined for accomplishing a service and each task spends a certain amount of time for execution. In the above-mentioned papers, modelling of time is not incorporated in the Rebeca language.

Creol is a concurrent object-oriented language with an operational semantics written in an actor-based style, and supported by a language interpreter in the Maude system. In [5], Creol is extended by adding best-case and worst-case execution time for each statement, and a deadline for each method call. In addition, an object is assigned a scheduling strategy to resolve the nondeterminism in selecting from the enabled processes. This work is along the same lines as the one presented in [13] and the focus is on schedulability analysis, which is carried out in a modular way in two steps: first one models an individual object and its behavioural interface as timed automata, and then one uses UPPAAL to check the schedulability considering the specified execution times and the deadlines. In this work, network delays are not considered, and the execution time is weaved together with the statements in a fine-grained way.

In [4] a timed version of Creol is presented in which the only additional syntax is read-only access to the global clock, plus adding a data-type *Time* together with its accompanying operators to the language. Timed behaviour is modelled by manipulating the *Time* variables and via the *await* statement in the language.

3 Timed Rebeca

A Rebeca model consists of a set of *reactive classes* and the *main* program in which we declare reactive objects, or rebecs, as instances of *reactive classes*. A reactive class has an argument of type integer, which denotes the length of its message queue. The body of the reactive class includes the declaration for its *known rebecs*, variables, and methods (also called message servers). Each method body consists of the declaration of local variables and a sequence of statements, which can be assignments, *if* statements, rebec creation (using the keyword *new*), and method calls. Method calls are sending asynchronous messages to other rebecs (or to self) to invoke the corresponding message server (method). Message passing is fair, and messages addressed to a rebec are stored in its message queue. The computation takes place by taking the message from the front of the message queue and executing the corresponding message server [22].

Timing features in an asynchronous and distributed setting. To decide on the timing primitives to be added to the Rebeca syntax, we first considered the different timing features that a modeller might need to address in a message-based, asynchronous and distributed setting. These features (like the computation time, or periodic events) can be common in any setting.

1. **Computation time**: the time needed for a computation to take place.
2. **Message delivery time**: the time needed for a message to travel between two objects, that depends on the network delay (and possibly other parameters).

3. **Message expiration**: the time within which a message is still valid. The message can be a request or a reply to a request (a request being served).
4. **Periods of occurrences of events**: the time periods for periodic events.

We introduce an extension of Rebeca with real-time primitives to be able to address the above-mentioned timing features. In timed Rebeca model, each rebec has its own local clock, which can be considered as synchronized distributed clocks[1]. Methods are still executed atomically, but we can model passing of time while executing a method. Instead of a message queue for each rebec, we have a bag containing the messages that are sent. The timing primitives that are added to the syntax of Rebeca are *delay*, *now*, *deadline* and *after*. Figure 1 shows the grammar for Timed Rebeca. The *delay* statement models the passing of time for a rebec during execution of a method (computation time), and *now* returns the local time of the rebec. The keywords *after* and *deadline* can only be used in conjunction with a method call. Each rebec knows about its local time and can put *deadline* on the messages that are sent declaring that the message will not be valid after the deadline (modelling the message expiration). The *after* primitive, attached to a message, can be used to declare a constraint on the earliest time at which the message can be served (taken from the message bag by the receiver rebec). The modeller may use these constraints for various purposes, such as modelling the network delay or modelling a periodic event.

The messages that are sent are put in the message bag together with their time tag and *deadline* tag. The scheduler decides which message is to be executed next based on the time tags of the messages. The time tag of a message is the value of *now* when the message was sent, with the value of the argument of the *after* added to it when the message is augmented with an *after*. The intuition is that a message cannot be taken (served) before the time that the time tag determines.

The progress of time is modeled locally by the delay statement. Each delay statement within a method body increases the value of the local time (variable *now*) of the respective rebec by the amount of its argument. When we reach a *call* statement (sending a message), we put that message in the message bag augmented with a time tag. The local time of a rebec can also be increased when we take a message from the bag to execute the corresponding method.

The scheduler takes a message from the message bag, executes the corresponding message server atomically, and then takes another message. Every time the scheduler takes a message for execution, it chooses a message with the least time tag. Before the execution of the corresponding method starts, the local time (*now*) of the receiver rebec is set to the maximum value between its current time and the time tag of the message. The current local time of each rebec is the value of *now*. This value is frozen when the method execution ends until the next method of the same rebec is taken for execution.

The arguments of *after* and *delay* are relative values, but when the corresponding messages are put in the message bag their tags are absolute values, which are computed by adding the relative values of the arguments to the value of the variable *now* of the sender rebec (where the messages are sent). To summarize, Timed Rebeca extends Rebeca with the following four constructs.

- **Delay**: *delay(t)*, where t is a positive natural number, will increase the value of the local clock of the respective rebec by the amount t.

[1] In this paper we do not address the problem of distributed clock synchronization; several options and protocols for establishing clock synchronization in a distributed system are discussed in the literature, including [23].

$$
\begin{aligned}
\mathit{Model} &::= \mathit{EnvVar}^* \ \mathit{Class}^* \ \mathit{Main} \qquad \mathit{EnvVar} ::= \textbf{env}\ T\ \langle v \rangle^+;\\
\mathit{Main} &::= \textbf{main}\ \{\ \mathit{InstanceDcl}^*\ \} \qquad \mathit{InstanceDcl} ::= C\ r(\langle r \rangle^*) : (\langle c \rangle^*);\\
\mathit{Class} &::= \textbf{reactiveclass}\ C\ \{\ \mathit{KnownRebecs}\ \mathit{Vars}\ \mathit{MsgSrv}^*\ \}\\
\mathit{KnownRebecs} &::= \textbf{knownrebecs}\ \{\ \mathit{VarDcl}^*\ \}\ \mathit{Vars} ::= \textbf{statevars}\ \{\ \mathit{VarDcl}^*\ \}\ \mathit{VarDcl} ::= T\ \langle v \rangle^+;\\
\mathit{MsgSrv} &::= \textbf{msgsrv}\ M(\langle T\ v \rangle^*)\ \{\ \mathit{Stmt}^*\ \}\\
\mathit{Stmt} &::= v = e;\ |\ r = \mathit{new}\ C(\langle e \rangle^*);\ |\ \mathit{Call};\ |\ \mathit{if}\ (e)\ \mathit{MSt}\ [\mathit{else}\ \mathit{MSt}]\ |\ \textbf{delay}(t);\ |\ \textbf{now}();\\
\mathit{Call} &::= r.M(\langle e \rangle^*)\ [\textbf{after}(t)]\ [\textbf{deadline}(t)]\\
\mathit{MSt} &::= \{\ \mathit{Stmt}^*\ \}\ |\ \mathit{Stmt}
\end{aligned}
$$

Figure 1: Abstract syntax of Timed Rebeca. Angle brackets ⟨...⟩ are used as meta parenthesis, superscript + for repetition more than once, superscript ∗ for repetition zero or more times, whereas using ⟨...⟩ with repetition denotes a comma separated list. Brackets [...] show being optional. Identifiers *C*, *T*, *M*, *v*, *c*, and *r* denote class, type, method, variable, constant, and rebec names, respectively; and *e* denotes an (arithmetic, boolean or nondetermistic choice) expression.

- **Now**: *now()* returns the time of the local clock of the rebec from which it is called.
- **Deadline**: *r.m() deadline(t)*, where *r* denotes a rebec name, *m* denotes a method name of *r* and *t* is a natural number, means that the message *m* is sent to the rebec *r* and is put in the message bag. After *t* units of time the message is not valid any more and is purged from the bag. Deadlines are used to model message expirations (timeouts).
- **After**: *r.m() after(t)*, where *r* denotes a rebec name, *m* denotes a method name of *r* and *t* is a natural number, means that the message *m* is sent to the rebec *r* and is put in the message bag. The message cannot be taken from the bag before *t* time units have passed. After statements can be used to model network delays in delivering a message to the destination, and also periodic events.

Ticket Service Example We use a ticket service as a running example throughout the article. Listing 1 shows this example written in Timed Rebeca. The ticket service model consists of two reactive classes: *Agent* and *TicketService*. Two rebecs, *ts*1 and *ts*2, are instantiated from the reactive class *TicketService*, and one rebec *a* is instantiated from the reactive class *Agent*. The agent *a* is initialized by sending a message *findTicket* to itself in which a message *requestTicket* is sent to the ticket service *ts*1 or *ts*2 based on the parameter passed to *findTicket*. The deadline for the message *requestTicket* to be served is *requestDeadline* time units. Then, after *checkIssuedPeriod* time units the agent will check if it has received a reply to its request by sending a *checkTicket* message to itself, modelling a periodic event. There is no receive statement in Rebeca, and all the computation is modeled via asynchronous message passing, so, we need a periodic check. The *attemptCount* variable helps the agent to keep track of the ticket service rebec that the request is sent to. The *token* variable allows the agent to keep track of which incoming *ticketIssued* message is a reply to a valid request. When any of the ticket service rebecs receives the *requestTicket* message, it will issue the ticket after *serviceTime1* or *serviceTime2* time units, which is modelled by sending *ticketIssued* to the agent with the *token* as parameter. The

expression ?(*serviceTime*1,*serviceTime*2) denotes a nondeterministic choice between *serviceTime*1 and *serviceTime*2 in the *assignment* statement. Depending on the chosen value, the ticket service may or may not be on time for its reply.

```
env int requestDeadline, checkIssuedPeriod, retryRequestPeriod, newRequestPeriod, serviceTime1,
    serviceTime2;

reactiveclass Agent {
  knownrebecs { TicketService ts1; TicketService ts2; }
  statevars { int attemptCount; boolean ticketIssued; int token; }
  msgsrv initial() { self.findTicket(ts1); } // initialize system, check 1st ticket service
  msgsrv findTicket(TicketService ts) {
    attemptCount += 1; token += 1;
    ts.requestTicket(token) deadline(requestDeadline); // send request to the TicketService
    self.checkTicket() after(checkIssuedPeriod); // check if the request is replied
  }
  msgsrv ticketIssued(int tok) { if (token == tok) { ticketIssued = true; } }
  msgsrv checkTicket() {
    if (!ticketIssued && attemptCount == 1) {        // no ticket from 1st service,
      self.findTicket(ts2);                          // try the second TicketService
    } else if (!ticketIssued && attemptCount == 2) { // no ticket from 2nd service,
      self.retry() after(retryRequestPeriod);        // restart from the first TicketService
    } else if (ticketIssued) {                       // the second TicketService replied,
      ticketIssued = false;
      self.retry() after(newRequestPeriod);          // new request by a customer
    }
  }
  msgsrv retry() {
    attemptCount = 0; self.findTicket(ts1);     // restart from the first TicketService
  }
}

reactiveclass TicketService {
  knownrebecs { Agent a; }
  msgsrv initial() { }
  msgsrv requestTicket(int token) {
    int wait = ?(serviceTime1,serviceTime2);    // the ticket service sends the reply
    delay(wait);                                // after a non-determinstic delay of
    a.ticketIssued(token);                      // either serviceTime1 or serviceTime2
  }
}

main {
  Agent a(ts1, ts2):();               // instantiate agent, with two known rebecs
  TicketService ts1(a):();            // instantiate 1st and 2nd ticket services, with
  TicketService ts2(a):();            // the agent as their known rebecs
}
```

Listing 1: A Timed Rebeca model of the ticket service example

3.1 Structural Operational Semantics for Timed Rebeca

In this section we provide an SOS semantics for Timed Rebeca in the style of Plotkin [19]. The behaviour of Timed Rebeca programs is described by means of the transition relation $\rightarrow$ that describes the evolution of the system.

The states of the system are pairs (Env, B), where Env is a finite set of environments and B is a bag of messages. For each rebec A of the program there is an environment σ_A contained in Env, that is a function that maps variables to their values. The environment σ_A represents the private store of the rebec A. Besides the user-defined variables, environments also contain

the value for the special variables *self*, the name of the rebec, *now*, the current time, and *sender*, which keeps track of the rebec that invoked the method that is currently being executed. The environment σ_A also maps every method name of A to its body.

The bag contains an unordered collection of messages. Each message is a tuple of the form $(A_i, m(\overline{v}), A_j, TT, DL)$. Intuitively, such a tuple says that at time TT the sender A_j sent the message to the rebec A_i asking it to execute its method m with actual parameters $\overline{v}$. Moreover this message expires at time DL.

The system transition relation $\rightarrow$ is defined by the rule *scheduler*:

$$(scheduler)\ \frac{(\sigma_{A_i}(m), \sigma_{A_i}[\text{now} = \max(TT, \sigma_{A_i}(now)),\ [\overline{arg} = \overline{v}],\ sender = A_j],\ Env,\ B) \xrightarrow{\tau} (\sigma'_{A_i},\ Env', B')}{(\{\sigma_{A_i}\} \cup Env, \{(A_i, m(\overline{v}), A_j, TT, DL)\} \cup B) \rightarrow (\{\sigma'_{A_i}\} \cup Env',\ B')} \qquad C$$

where the condition C is defined as follows: σ_{A_i} is not contained in Env, and $(A_i, m(\overline{v}), A_j, TT, DL) \notin B$, and $\sigma_{A_i}(now) \leq DL$, and $TT \leq \min(B)$. The *scheduler* rule allows the system to progress by picking up messages from the bag and executing the corresponding methods. The third side condition of the rule, namely $\sigma_{A_i}(now) \leq DL$, checks whether the selected message carries an expired deadline, in which case the condition is not satisfied and the message cannot be picked. The last side condition is the predicate $TT \leq min(B)$, which shows that the time tag TT of the selected message has been the smallest time tag of all the messages for all the rebecs A_i in the bag B. The premise executes the method m, as described by the transition relation $\xrightarrow{\tau}$, which will be defined below. The method body is looked up in the environment of A_i and is executed in the environment of A_i modified as follows: (1) The variable *sender* is set to the sender of the message. (2) In executing the method m, the formal parameters $\overline{arg}$ are set to the values of the actual parameters $\overline{v}$. Methods of arity n are supposed to have $arg_1, arg_2, \ldots, arg_n$ as formal parameters. This is without loss of generality since such a change of variable names can be performed in a pre-processing step for any program. (3) The variable *now* is set to the maximum between the current time of the rebec and the time tag of the selected message.

The execution of the methods of rebec A_i may change the private store of the rebec A_i, the bag B by adding messages to it and the list of environments by creating new rebecs through *new* statements. Once a method is executed to completion, the resulting bag and list of environments are used to continue the progress of the whole system.

The transition relation $\xrightarrow{\tau}$ describes the execution of methods in the style of natural semantics [15]. (See Figure 2 for selected rules. The full set of rules may be found in Appendix A.) Since in this kind of semantics the whole computation of a method is performed in a single step, this choice perfectly reflects the atomic execution of methods underlying the semantics of the Rebeca language. The general form of this type of transition is $(S, \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B')$. A single step of $\xrightarrow{\tau}$ consumes all the code S and provides the value resulting from its execution. Carrying the bag B is important because new messages may be added to it during the execution of a statement S. Also Env is required because *new* statements create new rebecs and may therefore add new environments to it. In the semantics, the local environment σ is separated from the environment list Env for the sake of clarity. The result of the execution of the method thus amounts to the modified private store σ', the new list of environments Env' and the new bag B'.

The rules for assignment, conditional statement and sequential composition are standard. The rules for the timing primitives deserve some explanation.

$$(msg)\ (varname.m(\overline{v})\ after(d)\ deadline(DL), \sigma, Env, B) \xrightarrow{\tau} (\sigma, Env, \{(\sigma(varname), m(eval(\overline{v}, \sigma)), \sigma(self), \sigma(now) + d, \sigma(now) + DL)\} \cup B)$$

$$(delay)\ (delay(d), \sigma, Env, B) \xrightarrow{\tau} (\sigma[now = \sigma(now) + d], Env, B)$$

$$(create)\ (varname = new\ O(\overline{v}), \sigma, Env, B) \xrightarrow{\tau} (\sigma[varname = A], \{\sigma_A[now = \sigma(now),\ self = A]\} \cup Env, \{(A, initial(eval(\overline{v}, \sigma)), \sigma(self), \sigma(now), +\infty)\} \cup B)$$

Figure 2: Selected Method-Execution Transition Rules. In rule *create*, the rebec name A should not appear in the range of the environment σ. The function *eval* evaluates expressions in a given environment in the expected way. In each rule, we assume that σ is not contained in *Env*.

- Rule *msg* describes the effect of method invocation statements. For the sake of brevity, we limit ourselves to presenting the rule for method invocation statements that involve both the *after* and *deadline* keywords. The semantics of instances of that statement without those keywords can be handled as special cases of that rule by setting the argument of *after* to zero and that of *deadline* to $+\infty$, meaning that the message never expires. Method invocation statements put a new message in the bag, taking care of properly setting its fields. In particular the time tag for the message is the current local time, which is the value of the variable *now*, plus the number d that is the parameter of the *after* keyword.
- Delay statements change the private variable *now* for the considered rebec.

Finally, the creation of new rebecs is handled by the rule *create*. A fresh name A is used to identify the newly created rebec and is assigned to *varname*. A new environment σ_A is added to the list of environments. At creation time, σ_A is set to have its method names associated to their code. A message is put in the bag in order to execute the *initial* method of the newly created rebec.

4 Mapping from Timed Rebeca to Erlang

In this section, we present a translation from the fragment of Timed Rebeca without rebec creation to Erlang (for an extended explanation and a more formal description see [10]). The motivation for translating Timed Rebeca models to Erlang code is to be able to use McErlang [7] to run experiments on the models. This translation also yields a first implementation of Timed Rebeca.

McErlang is a model-checking tool written in Erlang to verify distributed programs written in Erlang. It supports Erlang datatypes, process communication, fault detection and fault tolerance and the Open Telecom Platform (OTP) library, which is used by most Erlang programs. The verification methods range from complete state-based exploration to simulation, with specifications written as LTL formulae or hand-coded runtime monitors. This paper focuses on simulation since model checking with real-time semantics is not yet offered by McErlang. Note, however, that our translation opens the possibility of model checking (untimed) Rebeca models using McErlang, which is not the subject of this paper.

```
1 receive
2   Pattern1 when Guard1 -> Expr1;
3   Pattern2 when Guard2 -> Expr2;
4   ...
5 after
6   Time -> Expr
7 end
```

Listing 2: Syntax of a receive with timeout.

Erlang Primer Erlang is a dynamically-typed general-purpose programming language, which was designed for the implementation of distributed, real-time and fault-tolerant applications. Originally, Erlang was mostly used for telephony applications such as switches. Its concurrency model is based on the actor model.

Erlang has few concurrency and timing primitives:

- `Pid = spawn(Fun)` creates a new process that evaluates the given function `Fun` in parallel with the process that invoked spawn.
- `Pid !Msg` sends the given message `Msg` to the process with the identifier `Pid`.
- **`receive ... end`** receives a message that has been sent to a process; message discrimination is based on pattern matching.
- **`after`** is used in conjunction with a **`receive`** and is followed by a timeout block as shown in Listing 2, after the specified time (deadline for receiving the required pattern) the process executes the timeout block
- `erlang:now()` returns the current time of the process

When a process reaches a **`receive`** expression it looks in the queue and takes a message that matches the pattern if the corresponding guard is true. A guard is a boolean expression, which can include the variables of the same process. The process looks in the queue each time a message arrives until the timeout occurs.

Mapping The abstract syntax for a fragment of Erlang that is required to present the translation is shown in Figure 3. Table 1 offers an overview of how a construct in one language relates to one in the other. We discuss the general principles behind our translation in more detail below.

Reactive classes are translated into three functions, each representing a possible behaviour of an Erlang process: 1) the process waits to get references to known rebecs, 2) the process reads the initial message from the queue and executes it, 3) the process reads messages from the queue and executes them. Once processes reach the last function they enter a loop. Erlang pseudocode for the reactive class *TicketService* in the Rebeca model in Listing 1 is shown in Listing 3.

A message server is translated into a match expression (see Figure 3), which is used inside **`receive ... end`**. In Listing 3, `requestTicket` is the pattern that is matched on, and the body of the message server is mapped to the corresponding expression.

Message send is implemented depending on whether **`after`** is used. If there is no **`after`**, the message is sent like a regular message using the `!` operator, as shown on line 4 in Listing

$$
\begin{aligned}
&\mathit{Program} ::= \mathit{Function}^* \quad \mathit{Function} ::= v(\mathit{Pattern}^*) \rightarrow e \\
&\mathit{Expr} ::= e_1\ op_e\ e_2 \mid e(\langle e \rangle^*) \mid e_1\ !\ e_2 \mid e_1\ ,\ e_2 \mid \mathit{Pattern} = e \mid \textbf{case}\ e\ \textbf{of}\ \mathit{Match}\ \textbf{end} \mid \textbf{receive}\ \mathit{Match}\ \textbf{end} \\
&\quad \mid \textbf{receive}\ \mathit{Match}\ \textbf{after}\ \mathit{Time}\ \rightarrow\ e\ \textbf{end} \mid \textbf{if}\ \langle \mathit{Match} \rangle^* \textbf{end} \mid \mathit{BasicValue} \mid v \mid \{\langle e \rangle^*\} \mid [\langle e \rangle^*] \\
&\mathit{Match} ::= \mathit{Pattern}\ \textbf{when}\ \mathit{Guard} \rightarrow e \\
&\mathit{Pattern} ::= v \mid \mathit{BasicValue} \mid \{\langle \mathit{Pattern} \rangle^*\} \mid [\langle \mathit{Pattern} \rangle^*] \quad \mathit{Time} ::= \text{int} \\
&\mathit{Value} ::= \mathit{BasicValue} \mid \{\langle \mathit{Value} \rangle^*\} \mid [\langle \mathit{Value} \rangle^*] \quad \mathit{BasicValue} ::= \text{atom} \mid \text{number} \mid \text{pid} \mid \text{fid} \\
&\mathit{Guard} ::= g_1\ op_g\ g_2 \mid \mathit{BasicValue} \mid v \mid g(\langle g \rangle^*) \mid \{\langle g \rangle^*\} \mid [\langle g \rangle^*]
\end{aligned}
$$

Figure 3: Abstract syntax of a relevant subset of Erlang. Angle brackets ⟨...⟩ are used as meta parenthesis, superscript + for repetition more than once, superscript * for repetition zero or more times, whereas using ⟨...⟩ with repetition denotes a comma separated list. Identifiers v, p and g denote variable names, patterns and guards, respectively, and e denotes an expression. Note that {} and [] are parts of the syntax of Erlang representing tuples and lists, respectively.

Timed Rebeca		**Erlang**
Model	→	A set of processes
Reactive classes	→	A process whose behaviour consists of three functions
Known rebecs	→	Record of variables
State variables	→	Record of variables
Message server	→	A match in a receive expression
Local variables	→	Record of variables
Message send	→	Message send expression
Message send w/after	→	Message send expression in the timeout block of a receive with an empty pattern, the timeout block is always executed, sending the message after the specified time
Message send w/deadline	→	Message send expression with the deadline as parameter
Delay statement	→	Empty receive with a timeout
Now expression	→	System time
Assignment	→	Record update
If statement	→	If expression
Nondeterministic selection	→	Random selection in Erlang

Table 1: Structure of the mapping from Timed Rebeca to Erlang.

```
ticketService() ->
  receive
    % wait for a message with a set of known rebecs
    {Agent} ->
      % proceed to the next behaviour
      ticketService(#ticketService_knownrebecs{agent=Agent})
  end.
ticketService(KnownRebecs) ->
  receive
    % wait for the 'initial' message
    initial ->
      % process message 'initial' and proceed to the next behaviour
      ticketService(KnownRebecs, #ticketService_statevars{})
  end.
ticketService(KnownRebecs, StateVars) ->
  receive
    % wait for each message servers
    requestTicket ->
      % process message 'requestTicket' and loop
      ticketService(KnownRebecs, StateVars)
  end.
```

Listing 3: Pseudo Erlang code capturing the behaviour of the ticketService process.

```
1 Sender = self(),
2 spawn(fun() ->
3   receive after 15 ->
4     TicketService ! {{Sender, now(), inf}, requestTicket}
5   end
6 end)
```

Listing 4: Example of a message send after 15 time units in Erlang.

4. However, if the keyword **after** is present a new process is spawned which sleeps for the specified amount of time before sending the message as described before. Setting a deadline for the delivery of a message is possible by changing the value `inf`, which denotes no deadline (as shown on line 3 in Listing 4), to an absolute point in time. Messages are tagged with the time at which they were sent. For the simulation we use the system clock to find out the current time by calling the Erlang function **now**().

Moreover, since message servers can reply to the sender of the message, we need to take care of setting the sender as part of the message as seen on line 4 in Listing 4.

As there is no pattern to match with, the *delay* statement is implemented as a receive consisting of just a timeout that makes the process wait for a certain amount of time. For example, *delay*(10) is translated to **receive after** 10 ->**ok end**.

The *deadline* of each message is checked right before the body of the message server is executed. The current time is compared with the deadline of the message to see if the deadline has expired and, if so, the message is purged.

Request deadline	Check issued period	Retry request period	New request period	Service time 1	Service time 2	Result
2	1	1	1	3,4	7	Not issued
2	2	1	1	4	7	Not issued
2	2	1	1	3	7	Ticket issued

Table 2: Experimental simulation results for ticket service.

5 Simulation of Timed Rebeca Using McErlang

In this section, we present experimental results for two case studies. The first case study is the ticket service model displayed in Listing 1 and the second is a model of a sensor network. In each case we run a simulation for ten times, and for each case for 30 minutes or until a runtime monitor fails, which means that an erroneous state has been reached. The simulations are run in a setting in which a time unit is 1000 ms. The experiment platform is Macbook 2.0GHz Intel Core 2 Duo - Aluminum 4GB memory Mac OS X, 10.6.6, and Erlang R13B04.

Ticket Service The ticket service model is described in Section 3. For each simulation, we change one of the following parameters: the amount of time that is allowed to pass before a request is processed, the time that passes before agent checks if he has been issued a ticket, the amount of time that passes before agent tries the next ticket service if he did not receive a ticket, the amount of time that passes before agent restarts the ticket requests in case neither ticket service issued a ticket and two different service times, which are non-deterministically chosen as delay time in a ticket service and model the processing time for a request. Table 5 shows different settings of those parameters for which the ticket services never issue a ticket to the agent because of tight deadlines, as well as settings for which a ticket is issued during a simulation of the model.

Sensor Network We model a simple sensor network using Timed Rebeca. (See Listing 5 in Appendix B for the complete description of the model.) A distributed sensor network is set up to monitor levels of toxic gasses. The sensor rebecs (`sensor0` and `sensor1`), announce the measured value to the admin node (`admin` rebec) in the network. If the admin node receives reports of dangerous gas levels, it immediately notifies the scientist (`scientist` rebec) on the scene about it. If the scientist does not acknowledge the notification within a given time frame, the admin node sends a request to the rescue team (`rescue` rebec) to look for the scientist. The rescue team has a limited amount of time units to reach the scientist and save him.

The rebecs `sensor0` and `sensor1` will periodically read the gas-level measurement, modelled as a non-deterministic selection between `GAS_LOW` and `GAS_HIGH`, and send their values to `admin`. The `admin` continually checks, and acts upon, the sensor values it has received. When the `admin` node receives a report of a reading that is life threatening for the `scientist` (`GAS_HIGH`), it notifies him and waits for a limited amount of time units for an acknowledgement. The `rescue` rebec represents a rescue team that is sent off, should the `scientist` not acknowledge the message from the `admin` in time. We model the response speed of the rescue team with a non-deterministic delay of 0 or 1 time units. The `admin` keeps track of the deadlines for the `scientist` and the `rescue` team as follows:

Network delay	Admin period	Sensor 0 period	Sensor 1 period	Scientist deadline	Rescue deadline	Result
1	4	2	3	2	3	Mission failed
1	4	2	3	2	4	Mission success
2	1	1	1	4	5,6,7	Mission failed
2	4	1	1	4	7	Mission success

Table 3: Experimental simulation results for sensor network.

- the `scientist` must acknowledge that he is aware of a dangerous gas-level reading before *scientistDeadline* time units have passed;
- the `rescue` team must have reached the `scientist` within *rescueDeadline* time units.

Otherwise we consider the mission failed.

The model can be parameterized over the values of network delay, `admin` sensor-read period, `sensor0` read period, `sensor1` read period, scientist reply deadline and rescue-team reply deadline, as shown in Table 5. In that table, we can see two different cases in which we go from mission failure to mission success between simulations. In the first scenario, we go from mission failure to success as we increase the rescue deadline, as expected. In the second scenario, we changed the parameters to model a faster sensor update and we observed mission failure. In this scenario, increasing the rescue deadline further (from 5 to 7) is insufficient. Upon closer inspection, we observe that our model fails to cope with the rapid sensor updates and admin responses because it enters an unstable state. The admin node initiates a new rescue mission while another is still ongoing, eventually resulting in mission failure. This reflects a design flaw in the model for frequent updates that can be solved by keeping track of an ongoing rescue mission in the model. Alternatively, increasing the value of `admin` sensor-read period above half the rescue deadline eliminates the flaw and the simulation is successful again.

6 Future Work

The work reported in this paper paves the way to several interesting avenues for future work. In particular, we have already started modelling larger real-world case studies and analyzing them using our tool. We plan to explore different approaches for model checking Timed Rebeca models. It is worth noting that the translation from Timed Rebeca to Erlang immediately opens the possibility of model checking untimed Rebeca models using McErlang. This adds yet another component to the verification toolbox for Rebeca, whose applicability needs to be analyzed via a series of benchmark examples. As mentioned in the paper, McErlang supports the notion of time only for simulation and not in model checking, and therefore cannot be used as is for model checking Timed Rebeca models. We plan to explore different ways in which McErlang can be used for model checking Timed Rebeca. One possible solution is to store the local time of each process and write a custom-made scheduler in McErlang that simulates the way the Timed Rebeca scheduler operates. The formal semantics for Timed Rebeca presented in this paper is also used in another parallel line of work [11]. The aim of that study is to map Timed Rebeca to timed automata [2] in order to use UPPAAL [24] for model checking Timed Rebeca models. The translation from Timed Rebeca to timed automata will be integrated in our tool suite. We are also working on a translation of Timed Rebeca into (Real-time) Maude. This

alternative translation would allow designers to use the analysis tools supported by Maude in the verification and validation of Timed Rebeca models. Our long-term goal is to have a tool suite for modelling, executing, simulating, and model checking asynchronous object-based systems using Timed Rebeca.

Acknowledgements The work on this paper has been partially supported by the projects "New Developments in Operational Semantics" (nr. 080039021), "Meta-theory of Algebraic Process Theories" (nr. 100014021) and "Timed Asynchronous Reactive Objects in Distributed Systems: TARO" (nr. 110020021) of the Icelandic Research Fund.

References

[1] G. Agha (1990): *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA.

[2] R. Alur & D Dill (1994): *A Theory of Timed Automata*. *Theoretical Computer Science* 126, pp. 183–235. doi:10.1016/0304-3975(94)90010-8.

[3] Henry Givens Baker (1978): *Actor Systems for Real-Time Computation*. Technical Report, MIT.

[4] Joakim Bjørk, Einar Broch Johnsen, Olaf Owe & Rudolf Schlatte (2010): *Lightweight Time Modeling in Timed Creol*. In: *RTRTS*, pp. 67–81. doi:10.4204/EPTCS.36.4.

[5] Frank S. de Boer, Tom Chothia & Mohammad Mahdi Jaghoori (2009): *Modular Schedulability Analysis of Concurrent Objects in Creol*. In: *FSEN*, pp. 212–227. doi:10.1007/978-3-642-11623-0_12.

[6] Erlang: *Erlang Programming Language Homepage*. Http://www.erlang.org.

[7] Lars-Åke Fredlund & Hans Svensson (2007): *McErlang: a model checker for a distributed functional programming language*. In: *ICFP*, pp. 125-136. doi:10.1145/1291151.1291171.

[8] C. Hewitt (1972): *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot*. MIT Artificial Intelligence Technical Report 258, Department of Computer Science, MIT.

[9] Carl Hewitt (2007): *What is Commitment? Physical, Organizational, and Social (Revised)*. In: *Proceedings of Coordination, Organizations, Institutions, and Norms in Agent Systems II*, Lecture Notes in Computer Science, Springer, pp. 293–307. doi:10.1007/978-3-540-74459-7_19.

[10] ICEROSE: *ICEROSE Homepage*. Http://en.ru.is/icerose/applying-formal-methods/projects/TARO.

[11] Mohammad Javad Izadi (2010): *An Actor-based Model for Modeling and Verification of Real-Time Systems - Master Thesis, University of Tehran, Iran*.

[12] M. M. Jaghoori, F.S. de Boer, T. Chothia & M. Sirjani (2007): *Task scheduling in Rebeca*. In: *Proc. Nordic Workshop on Programming Theory (NWPT'07)*. Extended abstract.

[13] M. M. Jaghoori, F.S. de Boer, T. Chothia & M. Sirjani (2009): *Schedulability of Asynchronous Real-Time Concurrent Objects*. *Logic and Algebraic Programming* 78(5), pp. 402–416. A preliminary version appeared in NWPT/FLACOS 2007 as an extended abstract. doi:10.1016/j.jlap.2009.02.009.

[14] Mohammad Mahdi Jaghoori, Marjan Sirjani, Mohammad Reza Mousavi, Ehsan Khamespanah & Ali Movaghar (2009): *Symmetry and Partial Order Reduction Techniques in Model Checking Rebeca*. *Acta Informaticae* 47(1), pp. 33–66. doi:10.1007/s00236-009-0111-x.

[15] Gilles Kahn (1987): *Natural Semantics*. In Franz-Josef Brandenburg, Guy Vidal-Naquet & Martin Wirsing, editors: *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, *Lecture Notes in Computer Science* 247, Springer-Verlag, pp. 22–39. doi:10.1007/BFb0039592.

[16] Brian Nielsen & Gul Agha: *Semantics for an actor-based real-time language*. In: *Proceedings of The Fourth International Workshop on Parallel and Distributed Real-Time Systems (WPDRS'96)*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[17] Libero Nigro & Francesco Pupo (2001): *Schedulability Analysis of Real Time Actor Systems Using Coloured Petri Nets*. In: *Proc. Concurrent Object-Oriented Programming and Petri Nets*, pp. 493–513. doi:10.1007/3-540-45397-0_21.

[18] Peter Csaba Ölveczky & José Meseguer (2002): *Specification of real-time and hybrid systems in rewriting logic*. *Theor. Comput. Sci.* 285(2), pp. 359–405. doi:10.1016/S0304-3975(01)00363-2.

[19] G. D. Plotkin (1981): *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark.

[20] Shangping Ren & Gul Agha (1995): *RT-synchronizer: Language Support for Real-Time Specifications in Distributed Systems*. In: *Workshop on Languages, Compilers and Tools for Real-Time Systems*, pp. 50–59. doi:10.1145/216636.216656.

[21] M. Sirjani, A. Movaghar, A. Shali & F.S. de Boer (2005): *Model Checking, Automated Abstraction, and Compositional Verification of Rebeca Models*. *Journal of Universal Computer Science* 11(6), pp. 1054–1082.

[22] M. Sirjani, A. Movaghar, A. Shali & F.S. de Boer (Dec. 2004): *Modeling and Verification of Reactive Systems using Rebeca*. *Fundamenta Informatica* 63(4), pp. 385–410.

[23] Andrew S. Tanenbaum & Maarten van Steen (2007): *Distributed systems - principles and paradigms (2. ed.)*. Pearson Education.

[24] UPPAAL: *UPPAAL Homepage*. Http://uppaal.com.

[25] Wang Yi (1991): *CCS + time = an interleaved model for real time systems*. In: *Proceedings of ICALP 1991, Lecture Notes in Computer Science* 510, Springer-Verlag, pp. 217–228. doi:10.1007/3-540-54233-7_136.

A Method-Execution Transition Rules

$$(msg)\ (varname.m(\overline{v})\ after(d)\ deadline(DL), \sigma, Env, B) \xrightarrow{\tau} (\sigma, Env, \{(\sigma(varname), m(eval(\overline{v}, \sigma)), \sigma(self), \sigma(now) + d, \sigma(now) + DL)\} \cup B)$$

$$(delay)\ (delay(d), \sigma, Env, B) \xrightarrow{\tau} (\sigma[now = \sigma(now) + d], Env, B)$$

$$(assign)\ (x = e, \sigma, Env, B) \xrightarrow{\tau} (\sigma[x = eval(e, \sigma)], Env, B)$$

$$(create)\ (varname = new\ O(\overline{v}), \sigma, Env, B) \xrightarrow{\tau} (\sigma[varname = A], \{\sigma_A[now = \sigma(now),\ self = A]\} \cup Env, \{(A, initial(eval(\overline{v}, \sigma)), \sigma(self)), \sigma(now), +\infty)\} \cup B)$$

$$(cond_1)\ \frac{eval(e, \sigma) = true \quad (S_1, \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B')}{(if\ (e)\ then\ S_1\ else\ S_2, \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B')}$$

$$(cond_2)\ \frac{eval(e, \sigma) = false \quad (S_2, \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B')}{(if\ (e)\ then\ S_1\ else\ S_2,\ \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B')}$$

$$(seq)\ \frac{(S_1, \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B'),\ (S_2, \sigma', Env', B') \xrightarrow{\tau} (\sigma'', Env'', B'')}{(S_1; S_2, \sigma, Env, B) \xrightarrow{\tau} (\sigma'', Env'', B'')}$$

Figure 4: The Method-Execution Transitions Rules. In rule *create*, the rebec name A should not appear in the range of the environment σ. The function *eval* evaluates expressions in a given environment in the expected way. In each rule, we assume that σ is not contained in *Env*.

B Rebeca Model for the Sensor Network

```
1  env int netDelay;
2  env int adminCheckDelay;
3  env int sensor0period;
4  env int sensor1period;
5  env int scientistDeadline;
6  env int rescueDeadline;
7
8  reactiveclass Sensor {
9      knownrebecs {
10         Admin admin;
11     }
12
13     statevars {
14         int period;
15     }
```

```
16
17      msgsrv initial(int myPeriod) {
18          period = myPeriod;
19          self.doReport();
20      }
21
22      msgsrv doReport() {
23          int value;
24          value = ?(2, 4); // 2=safe gas levels, 4=danger gas levels
25          admin.report(value) after(netDelay);
26          self.doReport() after(period);
27      }
28  }
29
30  reactiveclass Scientist {
31      knownrebecs {
32          Admin admin;
33      }
34
35      msgsrv initial() {}
36
37      msgsrv abortPlan() {
38          admin.ack() after(netDelay);
39      }
40  }
41
42  reactiveclass Rescue {
43      knownrebecs {
44          Admin admin;
45      }
46
47      msgsrv initial() {}
48
49      msgsrv go() {
50          int msgDeadline = now() + (rescueDeadline-netDelay);
51          int excessiveDelay = ?(0, 1); // unexpected obstacle might occur during rescue
52          delay(excessiveDelay);
53          admin.rescueReach() after(netDelay) deadline(msgDeadline);
54      }
55  }
56
57  reactiveclass Admin {
58      knownrebecs {
59          Sensor sensor0;
60          Sensor sensor1;
61          Scientist scientist;
62          Rescue rescue;
63      }
64
65      statevars {
66          boolean reported0;
67          boolean reported1;
68          int sensorValue0;
69          int sensorValue1;
70          boolean sensorFailure;
71          boolean scientistAck;
72          boolean scientistReached;
73          boolean scientistDead;
74      }
75
76      msgsrv initial() {
77          self.checkSensors();
78      }
79
80      msgsrv report(int value) {
```

```
81          if (sender == sensor0) {
82              reported0 = true;
83              sensorValue0 = value;
84          } else {
85              reported1 = true;
86              sensorValue1 = value;
87          }
88      }
89
90      msgsrv rescueReach() {
91          scientistReached = true;
92      }
93
94      msgsrv checkSensors() {
95          if (reported0) reported0 = false;
96          else sensorFailure = true;
97
98          if (reported1) reported1 = false;
99          else sensorFailure = true;
100
101         boolean danger = false;
102         if (sensorValue0 > 3) danger = true;
103         if (sensorValue1 > 3) danger = true;
104
105         if (danger) {
106             scientist.abortPlan() after(netDelay);
107             self.checkScientistAck() after(scientistDeadline); // deadline for the scientist to answer
108         }
109
110         self.checkSensors() after(adminCheckDelay);
111     }
112
113     msgsrv checkRescue() {
114         if (!scientistReached) {
115             scientistDead = true; // scientist is dead
116         } else {
117             scientistReached = false;
118         }
119     }
120
121     msgsrv ack() {
122         scientistAck = true;
123     }
124
125     msgsrv checkScientistAck() {
126         if (!scientistAck) {
127             rescue.go() after(netDelay);
128             self.checkRescue() after(rescueDeadline);
129         }
130         scientistAck = false;
131     }
132 }
133
134 main {
135     Sensor sensor0(admin):(sensor0period);
136     Sensor sensor1(admin):(sensor1period);
137     Scientist scientist(admin):();
138     Rescue rescue(admin):();
139     Admin admin(sensor0, sensor1, scientist, rescue):();
140 }
```

Listing 5: A Timed Rebeca model of the sensor network example

A Verified Algebra for Linked Data

Ross Horne and Vladimiro Sassone
Electronics and Computer Science, University of Southampton, United Kingdom
{rjh06r,vs}@ecs.soton.ac.uk

A foundation is investigated for the application of loosely structured data on the Web. This area is often referred to as Linked Data, due to the use of URIs in data to establish links. This work focuses on emerging W3C standards which specify query languages for Linked Data. The approach is to provide an abstract syntax to capture Linked Data structures and queries, which are then internalised in a process calculus. An operational semantics for the calculus specifies how queries, data and processes interact. A labelled transition system is shown to be sound with respect to the operational semantics. Bisimulation over the labelled transition system is used to verify an algebra over queries. The derived algebra is a contribution to the application domain. For instance, the algebra may be used to rewrite a query to optimise its distribution across a cluster of servers. The framework used to provide the operational semantics is powerful enough to model related calculi for the Web.

1 Introduction

The application of interest is a powerful emerging idea commonly referred to as the Web of Data [6]. The Web of Data marks a shift from publishing documents to publishing data. The Web is based on documents which contain links to other documents. The Web of Data is concerned with resources more general than documents. Data on the Web contains links to resources described in multiple data sources. In both the case of the Web and the Web of Data the links between documents and resources, respectively, are established by a standardised global naming system --- the URI. On the Web, URIs allow documents in distributed locations with distinct ownership to refer to each other. Similarly, in a Web of Data, URIs allow data in distributed locations with distinct ownership to refer to common resources.

Suppose that the URIs are not used as a standard naming system. In this case, each data source uses its own naming system. Typically, in this case each data source is disjoint, hence traditional database techniques may be applied. This is referred to as closed world system, since the boundaries of the data source are known. For instance, classical negation can be used to determine whether some data does not appear in a data source, and schemata can constrain the structure of data.

In contrast, the presence of URIs as a global naming system, enables an open world system. In an open world system a variety of protocols can be used to obtain data from multiple sources based on the URIs which appear. For instance, a request may be sent to a URI to directly obtain some data about that URI. Alternatively, services may be used to find data relevant to a URI. In this open world setting, there is no guarantee that mechanisms find all relevant data. There may always be data not known locally which refers to a resource; hence in general optimal query results cannot be obtained and classical negation cannot be applied. Another restriction in an open world system is that schemata which constrain data cannot be enforced globally.

A light semi-structured data format must be agreed for the Web of Data. The W3C recommends the Resource Description Framework (RDF) as a general format for presenting data [16]. RDF is based on triples which consist of a subject, predicate and object. The subject, predicate and object are all named by URIs. Each URI in a triple may represent resources in different locations, hence a triple links locations.

M.R. Mousavi and A. Ravara: FOCLASA 2011
EPTCS 58, 2011, pp. 20--33, doi:10.4204/EPTCS.58.2

Other semi-structured data formats contain URIs, such as feeds. RDF is intended as a minimal data format to which other formats can be lifted.

Assuming that Linked Data can be gathered, observations about Linked Data can be made. The W3C recommendation is to use SPARQL Queries to make such observations [24]. In this work, to model this scenario, both RDF Data and SPARQL Queries are internalised in a process calculus. The operational semantics of the process calculus specifies how queries and data interact, to realise the W3C recommendations. The operational semantics are realistic since there is no guarantee of maximal responses, only that responses are correct.

Two SPARQL Queries may be indistinguishable with respect to their operational behaviour. Such operationally equivalent queries are bisimilar. In this work, bisimulation is used to derive an algebra over SPARQL Queries. The algebra agrees with expected equivalences analogous to those uncovered by relational algebra and exposes some new equivalences. The derived algebra can be used to rewrite a query to a normal form. Normal forms are useful for optimisation purposes. A query can be optimised before being distributed over multiple data sources. Distribution of queries is a key challenge for enabling a Web of Data [11].

Section 2 presents a syntax and semantics for RDF triples, SPARQL queries and processes which internalise both triples and queries. Section 3 provides an alternative operational semantics using a labelled transition system. The labelled transition system is proven to be sound with respect to the reduction system. Section 4 introduces two notions of equivalence over the calculus, which correspond to the two operational semantics. Bisimulation for the labelled transition system is proven to be complete with respect to contextual equivalence for the reduction system. An algebra for queries is verified using bisimulation.

2 A syntax and semantics for the syndication calculus

The concrete syntax for both RDF and SPARQL Query are specified in W3C recommendations [16, 24]. Here an abstract syntax is presented to model the core features of the concrete syntax. This abstract syntax is easier to define than the concrete syntax, which is sugared to make programming easier.

The operational semantics of the calculus is specified as a reduction system. The syntax and rules of the reduction system borrow from a fragment of Linear Logic, extended with a continuation. Related work has investigated other approaches to using Linear Logic for both query languages and process calculi [17, 13, 3].

Note that the description of the syntax and reduction system is brief. A similar syntax and reduction system are extensively discussed in the thesis of the first author [14]. The main contribution of this paper is the bisimulation results for queries.

2.1 A syntax for RDF triples

An abstract syntax for triples conveys the RDF data format. The atoms of the syntax are names and literals. Names represent occurrences of URIs, which are represented by identifiers in italics, such as *John* or *knows*. Literals are basic data values, such as the strings `'Paul'` or `'77-3426'`. The definition of literals in the XML Schema Datatypes specification [4] is assumed. Variables $a, b \ldots$ and $x, y \ldots$ represent place holders for names and literals respectively.

A triple consists of three components: the subject, the predicate and the object, which is written (*subject predicate object*). The subject is related by the predicate to the object, similarly to simple sen-

$$
\begin{array}{rlr}
\phi ::= & \mathrm{I} & \text{true} \\
| & 0 & \text{false} \\
| & \phi \vee \phi & \text{or} \\
| & \phi \wedge \phi & \text{and} \\
| & \neg\phi & \text{not} \\
| & \ldots & \text{etc.}
\end{array}
\qquad
\begin{array}{rlr}
U ::= & C & \text{asked triple} \\
| & \phi & \text{filter} \\
| & U \oplus U & \text{choice} \\
| & U \otimes U & \text{tensor} \\
| & \bigvee a.U & \text{select name} \\
| & \bigvee x.U & \text{select literal} \\
| & *U & \text{iteration} \\
| & U\,;P & \text{then}
\end{array}
\qquad
\begin{array}{rlr}
P ::= & \bot & \text{nothing} \\
| & P \parr P & \text{par} \\
| & \bigwedge a.P & \text{blank node} \\
| & U & \text{query} \\
| & \underline{C} & \text{stored triple}
\end{array}
$$

Figure 1: The syntax of constraints (ϕ), queries (U) and processes (P), over triples (C).

tences in English of form subject-verb-object, where URIs and literals are used instead of words. The syntax ensures that literals can only appear as the object of a triple. The example below presents two RDF triples.

(b_4 *home starr.uk*) (b_4 *give_name* `'Ringo'`)

Predicates are names such as *home*. For instance, the first triple above means that a subject b_4 is related by predicate *home* to object *starr.uk*. The second triple above indicates that subject b_4 is related by predicate *given_name* to the literal `'Ringo'`.

2.2 A syntax for SPARQL queries

In this section an abstract syntax for queries, Fig. 1, represents the core features of SPARQL Query [24]. SPARQL Queries are used to read from RDF triples. Synchronisation constructs allow substantial queries to be expressed. The syntax of processes, also in Fig. 1, demonstrates how both queries and content can be internalised in a process calculus, which suggests a high level language for Linked Data, which uses query results. In this model, persistently stored triples are used to answer queries. A stored triple is indicated by an underscore.

Ask queries and multiplicative operators. The simplest 'ask' query provides a triple to be matched. There are three multiplicative operators: a tensor product ($\otimes$) for synchronously joining queries, a par operator ($\parr$) for composing processes in parallel and the operator then (;) for guarding a process with a query. The difference between tensor and par is that queries composed using tensor must happen simultaneously (in the same atomic step), whereas processes composed in parallel may be used in different atomic steps. Tensor is the implicit join of queries used in SPARQL. Then and par are part of a higher level language, where query results are immediately used. These operators are multiplicative since they control the sharing of resources.

The additive operators and select queries. There are three additive operators: choose ($\oplus$), select ($\bigvee$) and the blank node quantifier ($\bigwedge$). The choose operator presents a choice between two queries, hence models the SPARQL keyword `UNION`. The select operator is a quantifier which binds a variable. Select is used to model `SELECT` queries in SPARQL, which discover names and literals. The names and literals discovered can also be bound in a continuation process, hence value passing is modelled at a high level. Blank node quantifiers provide a model for blank nodes in RDF [16]. A blank node is a local name

$$P \parr \bot \equiv P \qquad P \parr Q \equiv Q \parr P \qquad P \parr (Q \parr R) \equiv (P \parr Q) \parr R$$

$$\bigwedge a.\bot \equiv \bot \qquad \bigwedge a.\bigwedge b.P \equiv \bigwedge b.\bigwedge a.P \qquad \bigwedge a.P \parr Q \equiv \bigwedge a.(P \parr Q) \quad a \notin \mathrm{fn}(P)$$

Figure 2: The structural congruence over processes.

where the scope of the blank node is indicated by the scope of the quantifier. Blank nodes allow further data structures to be represented in RDF, including XML.

Constraints and optional queries. A constraint may be used in a query. Constraints form a Boolean algebra of basic predicates, such as inequalities and regular expressions. The specification of constraints can be found under the keyword `FILTER` in the recommendation [24]. A choice between a query and true models an optional query in SPARQL, so the keyword `OPTIONAL` is defined as follows: $\texttt{OPTIONAL}\, U \triangleq U \oplus \mathrm{I}$.

Repeated queries and iteration. A common requirement of a query language is that more than one result can be obtained. Bounded multiple copies of queries can be synchronously posed, using queries with natural number exponents and finite sums. Exponents and sums are just abbreviations defined as follows.

$$U^0 \triangleq \mathrm{I} \qquad U^{n+1} \triangleq U \otimes U^n \qquad \Sigma_{n=0}^{0} U^n \triangleq \mathrm{I} \qquad \Sigma_{n=0}^{k+1} U^n \triangleq \Sigma_{n=0}^{k} U^n \oplus U^{k+1}$$

A natural number exponent n repeatedly applies the tensor product, so the query must be answered exactly n times. The sum with bound n allows the query to be answered between 0 and n times. Sums model the keyword `LIMIT`, such that $U\,\texttt{LIMIT}\,k \triangleq \Sigma_{n=0}^{k} U^n$.

Unbounded iteration of queries is indicated by an explicit operator ($*$), which allows zero or more copies of a query to be answered. Note that iteration differs from replication in common process calculi. All copies of an iterated query must be answered simultaneously using disjoint resources.

2.3 A reduction system for the calculus

The reduction system presents a concise operational semantics for the calculus. The reduction system is defined by a structural congruence and a relation over processes called the commitment relation. A further preorder over triples formalises key features of RDF Schema (RDFS [7]). RDFS is a light extension to RDF, which improves interoperability by resolving aliases between URIs.

The structural congruence ($\equiv$ in Fig. 2) is defined such that $(P, \parr, \bot)$ forms a commutative monoid. Alpha conversion can also be applied to blank node quantifiers. Furthermore, blank node quantifiers can be eliminated in the presence of nothing, commute and distribute over par. All reductions are considered up to structural congruence --- as standard in process calculi.

The commitment relation ($\rhd$ in Fig. 3) specifies atomic operational steps. The process on the left of the commitment relation, becomes the process on the right. A commitment is performed atomically.

Working with aliases for URIs is a key problem in Linked Data [2]. Aliases arise since different data sources use different URIs for similar purposes. For instance, in the context of a song, predicate *lyricist* may be more specific than predicate *creator* (see subPropertyOf in RDFS [7]). Similarly, $song_0$ and $song_1$ may be URIs for the same song (see sameAs in OWL [2]). Hence the aliases $lyricist \sqsubseteq creator$

$$\frac{C \sqsubseteq D}{\underline{C} \parr D \rhd \underline{C}} \qquad \frac{\vDash \phi}{\phi \rhd \bot} \qquad \frac{P \parr U \rhd Q}{P \parr (U \oplus V) \rhd Q} \qquad \frac{P \parr V \rhd Q}{P \parr (U \oplus V) \rhd Q} \qquad \frac{P \parr U \rhd P' \quad Q \parr V \rhd Q'}{P \parr Q \parr (U \otimes V) \rhd P' \parr Q'}$$

$$*U \rhd \bot \qquad \frac{P \parr U \rhd Q}{P \parr *U \rhd Q} \qquad \frac{P \parr (*U \otimes *U) \rhd Q}{P \parr *U \rhd Q} \qquad \frac{P \parr U\{^b/_a\} \rhd Q}{P \parr \bigvee a.U \rhd Q} \qquad \frac{P \parr U\{^v/_x\} \rhd Q}{P \parr \bigvee x.U \rhd Q}$$

$$\frac{P \parr U \rhd Q}{P \parr (U\,;R) \rhd Q \parr R} \qquad \frac{P \rhd P'}{P \parr Q \rhd P' \parr Q} \qquad \frac{P \parr Q \rhd P' \parr Q'}{P \parr \bigwedge a.Q \rhd P' \parr \bigwedge a.Q'}\ a \notin \mathrm{fn}\left(\ P, P', \beta\ \right)$$

Figure 3: Commitment rules: ask, filter, choose left, choose right, tensor, weakening, dereliction, contraction, select name, select literal, guard, context, and blank node (fn indicates the free names).

and $song_0 \sqsubseteq song_1$ may be assumed. The application specific set of alias assumptions is referred to as β. The transitive reflexive closure of β gives rise to a preorder ($\sqsubseteq$) over URIs.

The ask axiom, guard rule and alias assumptions. The following example demonstrates the interaction of an ask query with a continuation and a stored triple. The axiom 'ask' allows a query triple and a stored triple to interact. The stored triple remains available after the commitment. The axiom 'guard' makes the continuation process available after the commitment.

$$\underline{(song_0\ lyricist\ b_4)} \parr ((song_1\ creator\ b_4)\,;P) \rhd \underline{(song_0\ lyricist\ b_4)} \parr P$$

Above, the conditions for a match are relaxed by the preorder over triples ($\sqsubseteq$). The preorder over triples is the point-wise extension of the preorder over URIs introduced above.

The tensor and select rules. The following example demonstrates two synchronised queries, in the presence of two stored triples. The first query poses a pattern to match, while the second query selects a name with respect to a pattern.

$$\begin{array}{l}((b_2\ role\ singer) \otimes \bigvee b.((b\ role\ guitarist)\,;P)) \parr \\ \underline{(b_2\ role\ singer)} \parr \underline{(b_3\ role\ guitarist)}\end{array} \rhd \begin{array}{l}\underline{(b_2\ role\ singer)} \parr \\ \underline{(b_3\ role\ guitarist)} \parr P\{^{b_3}/_b\}\end{array}$$

In the above example, the 'tensor' rule divides the stored triples between the two parts of the query. On the left the 'select' rule is applied. The 'select' rule substitutes a suitable URI for the quantified name. The result is that a URI is passed to the continuation.

The choose rule. The following example demonstrates a choice between queries. The 'choose left' rule is used in this case.

$$\bigvee a.(((a\ knows\ b_2)\,;P) \oplus ((b_2\ knows\ a)\,;Q)) \parr \underline{(b_1\ knows\ b_2)} \rhd \underline{(b_1\ knows\ b_2)} \parr P\{^{b_1}/_a\}$$

The query result determines the continuation triggered.

Constraints in queries. The example query below selects a literal. The data literal appears in a triple and a constraint. The rules ensure that both a suitable triple appears and the constraint imposed holds.

$$\bigvee x.((|x| \leq 5) \otimes (b_1 \ \mathit{name}\ x)\,;P) \parr \underline{(b_1 \ \mathit{name}\ \texttt{`John'})} \triangleright \underline{(b_1 \ \mathit{name}\ \texttt{`John'})} \parr P\left\{{}^{\texttt{`John'}}/_{x}\right\}$$

The satisfaction relation for evaluating constraints $\models$, is left to the W3C recommendation [24]. Satisfaction is assumed to define a Boolean algebra of constraints.

The rules for iteration of queries. The example below demonstrates iteration used to answer two copies of the same query. Two iterated queries are answered using 'dereliction', which are combined using the conventional tensor rule. The 'contraction' rule then reduces the combined queries to a single query.

$$*\bigvee c.((c \ \mathit{is\ busy})\,;P) \parr \underline{(b_2 \ \mathit{is\ busy})} \parr \underline{(b_3 \ \mathit{is\ busy})} \triangleright \underline{(b_2 \ \mathit{is\ busy})} \parr \underline{(b_3 \ \mathit{is\ busy})} \parr P\left\{{}^{b_2}/_{c}\right\} \parr P\left\{{}^{b_3}/_{c}\right\}$$

A continuation for each result is triggered. Note the 'weakening' rule could be used to allow the query to be answered zero times.

Blank nodes as quantifiers. The example below demonstrates a query which discovers a blank node. The 'blank node' rule uses a temporary name to represent the blank node. The result is that the scope of the blank node quantifier is extended to include the continuation, which receives the blank node.

$$\begin{array}{l} \bigvee c.((c \ \mathit{creator}\ b_2)\,;U) \parr \\ \bigwedge a.\Big(\underline{(a \ \mathit{author}\ b_2)} \parr \underline{(a \ \mathit{status\ open})}\Big) \end{array} \triangleright \bigwedge a.\left(\begin{array}{l} U\{{}^{a}/_{c}\} \parr \\ \underline{(a \ \mathit{author}\ b_2)} \parr \underline{(a \ \mathit{status\ open})} \end{array} \right)$$

The alias $\mathit{author} \sqsubseteq \mathit{creator}$ is assumed above. The temporary name must not appear in the alias assumptions (β). The unused stored triple is idled.

Rules for an additive disjunction, tensor product, existential quantification, universal quantification and iteration, are borrowed from Linear Logic [9]. The sequent calculus is extended to indicate a continuation process, constraints extend the basic units with a Boolean algebra, and a preorder accommodates aliases over names.

3 A labelled transition system for the operational semantics

The operational semantics can be expressed as a labelled transition system. This provides an alternative operational semantics to the reduction system. This alternative semantics allows the behaviour of queries and data to be evaluated separately and then composed. Lemma 2 verifies that the labelled transition system and reduction system describe the same behaviour.

3.1 The purpose of labels

A labelled transition consists of two processes and a label. The first process is the process before the transition. The label is a constraint on the context in which a transition can take place. The second process is the resulting process after the transition.

The labels are formed from a commutative monoid over triples $(E, \otimes, \mathrm{I})$. A label indicates the inputs and outputs of a process. An input indicates that a process can proceed if it can receive the triples on the

$$\frac{C \sqsubseteq D}{D \xrightarrow{C} \bot} \qquad \frac{U \xrightarrow{E} Q}{U\,;P \xrightarrow{E} Q \parr P} \qquad \frac{U \xrightarrow{E} P \quad V \xrightarrow{F} Q}{U \otimes V \xrightarrow{E \otimes F} P \parr Q} \qquad \frac{U \xrightarrow{E} P}{U \oplus V \xrightarrow{E} P} \qquad \frac{V \xrightarrow{E} Q}{U \oplus V \xrightarrow{E} Q}$$

$$\frac{\vDash \phi}{\phi \xrightarrow{\mathrm{I}} \bot} \qquad \frac{U\{{}^{b}/_{a}\} \xrightarrow{E} Q}{\bigvee a.U \xrightarrow{E} Q} \qquad \frac{U\{{}^{v}/_{x}\} \xrightarrow{E} Q}{\bigvee x.U \xrightarrow{E} Q} \qquad *U \xrightarrow{\mathrm{I}} \bot \qquad \frac{U \xrightarrow{E} P}{*U \xrightarrow{E} P} \qquad \frac{*U \otimes *U \xrightarrow{E} P}{*U \xrightarrow{E} P}$$

Figure 4: Labelled transitions for queries: input triple, trigger guard, tensor, choose left, choose right, filter, select name, select literal, weakening, dereliction and contraction.

label from its context. An output indicates that a process outputs the triple on the label to its context. For instance, the query below inputs a triple; while the stored triple below outputs a triple.

$$(b_4\ \mathit{knows}\ b_3)\,;P \xrightarrow{(b_4\ \mathit{knows}\ b_3)} P \qquad\qquad \underline{(b_4\ \mathit{knows}\ b_3)} \xrightarrow{\overline{(b_4\ \mathit{knows}\ b_3)}} \underline{(b_4\ \mathit{knows}\ b_3)}$$

A relevant interpretation is that the first transition above is an action from the perspective of a client which resolves a query; whereas the second is an action from the perspective of a server that provides a triple. Two processes composed in parallel with matching inputs and outputs may interact. For instance, the above processes can be composed, resulting in the following transition. The unit label indicates an operational step without side effects.

$$(b_4\ \mathit{knows}\ b_3)\,;P \parr \underline{(b_4\ \mathit{knows}\ b_3)} \xrightarrow{\mathrm{I}} P \parr \underline{(b_4\ \mathit{knows}\ b_3)}$$

Output labels can also indicate extruded names. For instance, the example below extrudes the name a. The extruded names represent blank nodes where the scope of the blank node quantifier may be extended. This is similar to extrusion of new names in the π-calculus [22].

$$\bigwedge a.\underline{(a\ \mathit{has\ paper})} \parr \underline{(b_2\ \mathit{has\ stone})} \xrightarrow{a|\overline{(a\ \mathit{has\ paper})}} \underline{(a\ \mathit{has\ paper})} \parr \underline{(b_2\ \mathit{has\ stone})}$$

The commutative monoid rules can always be applied to reorder labels.

3.2 Labelled transitions for queries

The input transitions allow the behaviour of a query to be modelled independently. The rules for queries are presented in Fig. 4. The rules accumulate RDF triples on an input label, which represents contexts in which a query may be answered.

The 'input triple' rule poses the triple as an input on the label. The triple on the label may be strengthened by the preorder over triples. The 'trigger guard' rule allows a continuation process to be triggered exposing the continuation. The following example demonstrates a query consisting of a single triple and a continuation process, where the preorder $\mathit{colleague} \sqsubseteq \mathit{knows}$ is assumed.

$$(b_4\ \mathit{knows}\ b_3)\,;P \xrightarrow{(b_4\ \mathit{colleague}\ b_3)} P$$

Select quantifiers are resolved by anticipating the name or literal to input. For instance, the following labelled transition indicates that the query can be answered in a context where a name is chosen. The same name is passed to the continuation process.

$$\bigvee a.((b_4\ \mathit{knows}\ a)\,;P) \xrightarrow{(b_4\ \mathit{knows}\ b_3)} P\{{}^{b_3}/_{a}\}$$

$$\frac{C \sqsubseteq D}{\underline{C} \xrightarrow{\overline{D}} \underline{C}} \qquad \frac{P \xrightarrow{\alpha|\overline{E}} Q}{\bigwedge a.P \xrightarrow{\alpha+a|\overline{E}} Q}\ a \notin \mathrm{fn}(\beta) \qquad \frac{P \xrightarrow{\alpha|\overline{E}} Q}{\bigwedge a.P \xrightarrow{\alpha|\overline{E}} \bigwedge a.Q}\ a \notin \alpha \cup \mathrm{fn}(E)$$

$$\frac{P \xrightarrow{\alpha|\overline{E}} P'}{P \parr Q \xrightarrow{\alpha|\overline{E}} P' \parr Q}\ \alpha \cap \mathrm{fn}(Q) = \emptyset \qquad \frac{P \xrightarrow{\alpha_0|\overline{E}} P' \quad Q \xrightarrow{\alpha_1|\overline{F}} Q'}{P \parr Q \xrightarrow{\alpha_0+\alpha_1|\overline{E \otimes F}} P' \parr Q'}\ \begin{array}{l}\alpha_0 \cap \mathrm{fn}(Q) = \emptyset \\ \alpha_1 \cap \mathrm{fn}(P) = \emptyset\end{array}$$

$$\frac{P \xrightarrow{E \otimes F} P' \quad Q \xrightarrow{\alpha|\overline{F}} Q'}{P \parr Q \xrightarrow{E} \bigwedge \alpha.(P' \parr Q')}\ \alpha \cap (\mathrm{fn}(P) \cup \mathrm{fn}(E)) = \emptyset$$

Figure 5: Process rules: output triple, open, blank node context, par context, parallel outputs and close. The symmetric versions of the par context and close rule are also assumed.

Choices are resolved by anticipating the left or right branch. For instance, the following transition indicates the label and continuation which results from choosing the left branch.

$$((b_4\ \mathit{knows}\ b_2)\,;P) \oplus ((b_4\ \mathit{knows}\ b_3)\,;Q) \xrightarrow{(b_4\ \mathit{knows}\ b_2)} P$$

Tensor synchronises two queries, by composing their respective labels and continuations. For instance, the following query simultaneously inputs two triples. The continuations of both queries are triggered in parallel, with the appropriate substitutions.

$$\bigvee a.(((b_4\ \mathit{knows}\ a)\,;P) \otimes (\bigvee x.(a\ \mathit{name}\ x)\,;Q)) \xrightarrow{(b_4\ \mathit{knows}\ b_2) \otimes (b_2\ \mathit{name}\ \texttt{`John'})} P\left\{{}^{b_2}/_{a}\right\} \parr Q\left\{{}^{b_2,\texttt{`John'}}/_{a,x}\right\}$$

A constraint is disposed when it is satisfied. For instance, in the following query the length of a selected literal is constrained, but satisfied by the substitution.

$$\bigvee x.((b_2\ \mathit{name}\ x) \otimes (|x| \leq 5)\,;P) \xrightarrow{(b_2\ \mathit{name}\ \texttt{`John'})} P\left\{{}^{\texttt{`John'}}/_{x}\right\}$$

Iteration anticipates the number of copies of a query to pose using weakening, dereliction and contraction. For instance, two copies of the following query are posed using contraction and dereliction. The label indicates the two separate triples which are to be answered simultaneously. Both continuations are composed in parallel.

$$*\bigvee a.((b_4\ \mathit{knows}\ a)\,;P) \xrightarrow{(b_4\ \mathit{knows}\ b_2) \otimes (b_4\ \mathit{knows}\ b_3)} P\left\{{}^{b_2}/_{a}\right\} \parr P\left\{{}^{b_3}/_{a}\right\}$$

The rules of the labelled transition system are sufficient to model queries.

3.3 Labelled transitions for an RDF store

The behaviour of stored RDF triples can be modelled using output labels. The rules of output labels are presented in Fig. 5. The names extruded on the label are indicated by α, where + indicates disjoint union of names. The abbreviation $\bigwedge \alpha.P$ is used to indicate the quantification of all names in α.

Stored triples can output the triple on the label. The same triple appears in the continuation unchanged. The preorder over names may be used to weaken the output triple. Names are extruded on

the label using the 'open scope' rule. For instance, the following triple outputs a triple and extrudes the blank node, using the assumption $\mathit{colleague} \sqsubseteq \mathit{knows}$.

$$\bigwedge b_4.\underline{(b_4\ \mathit{colleague}\ b_3)} \xrightarrow{b_4|\overline{(b_4\ \mathit{knows}\ b_3)}} \underline{(b_4\ \mathit{colleague}\ b_3)}$$

Output labels composed in parallel can be combined. Extruded names on both labels must be disjoint to preserve the scope of blank nodes. For instance, the following transition simultaneously outputs two triples and extrudes three names.

$$\bigwedge b_4.\Big(\bigwedge b_2.\underline{(b_4\ \mathit{knows}\ b_2)} \parr \bigwedge b_3.\underline{(b_4\ \mathit{knows}\ b_3)}\Big) \xrightarrow{b_2,b_3,b_4|\overline{(b_4\ \mathit{knows}\ b_2)\otimes(b_4\ \mathit{knows}\ b_3)}} \underline{(b_4\ \mathit{knows}\ b_2)} \parr \underline{(b_4\ \mathit{knows}\ b_3)}$$

Two parallel processes may interact using the close rule. Close allows complementary inputs and outputs to be matched. Names extruded on the output label are introduced as quantifiers in the continuation. Any inputs not answered remain on the resulting label, to be answered later. For instance, the following iterated query is answered twice. One copy is answered by the available process and the other copy must be answered by the context for the transition to occur. In the continuation, the scope of the blank node is extended.

$$*\bigvee a.((b_4\ \mathit{knows}\ a)\,;P) \parr \bigwedge b_3.\underline{(b_4\ \mathit{knows}\ b_3)} \xrightarrow{(b_4\ \mathit{knows}\ b_2)} \bigwedge b_3.\Big(P\{^{b_2}/_a\} \parr P\{^{b_3}/_a\} \parr \underline{(b_4\ \mathit{knows}\ b_3)}\Big)$$

The context rule for parallel composition allows a process which does not contribute to an interaction to idle. Similarly, the context rule for blank node quantifiers allows a blank node to be ignored in a transition if it does not appear on the label.

3.4 Comparison of the two operational semantics

To justify the labelled transition system, the labelled transitions are compared to the reductions of the reduction system. If a unit labelled transition can be derived then the corresponding reduction can also be derived. The significance is that, given the independent perspectives of the query and the store in terms of labelled transitions, their combination satisfies the global perspective specified by the reduction system.

Scope extrusion presents technical difficulties. The following technical lemma reduces these difficulties, by eliminating scope extrusion. The proof demonstrates that combinations of opening names and closing names can be eliminated from a proof tree which uses an extruded name.

Lemma 1 (Elimination of extrusion)**.** *Suppose that a labelled transition proof uses name extrusion, but not in the conclusion. The same labelled transition, up to structural congruence, holds without any name extrusion.*

Note that full proofs for all theorems are provided in the thesis of the first author [14].

Every completed labelled transition can also be expressed as a reduction, Lemma 2. The proof works by transforming proof trees so that labels used in interactions are eliminated.

Lemma 2 (Elimination of labels)**.** $P \xrightarrow{\mathrm{I}} Q$ *if and only if* $P \triangleright Q$.

Thus the local perspective of the labelled transition system and the global perspective of the reduction system specify the same operational capabilities.

4 An algebra for the syndication calculus

In this section bisimulation is introduced as the natural notion of equivalence over the labelled transition system. Bisimulation is demonstrated to be sound with respect to equivalence in the reduction system. Thus every pair of bisimilar processes are equivalent with respect to the natural notion of equivalence over the reduction system. Bisimulation is then used to verify an algebra over queries and processes.

4.1 Bisimulation

Processes which are capable of the same observable behaviour can be regarded as equivalent. The observable behaviour of a process is given by the labels of the labelled transition system. Observational equivalence of processes is established using the technique of (strong) bisimulation, as follows.

Definition 1 (Bisimulation)**.** *Bisimulation, written $\sim$, is the greatest symmetric relation such that the following holds, for any label l. If $P \sim Q$ and $P \xrightarrow{l} P'$ then there exists some Q' such that $Q \xrightarrow{l} Q'$ and $P' \sim Q'$.*

The following verifies that bisimulation is a congruence --- a relation which holds in any context. It is necessary that bisimulation is a congruence for it to be used as an algebra. A context is a process with a place holder for some syntax.

Lemma 3 (Bisimulation is a congruence)**.** *If $P \sim Q$ and C is a context, then $CP \sim CQ$.*

An alternative notion of equivalence is defined using the reduction system. Contextual equivalence is used in related work to justify notions of bisimulation on the π-calculus and ambient calculus [15, 21].

Definition 2 (Contextual equivalence)**.** *Contextual equivalence, written $\simeq$, is the greatest symmetric, reduction closed, context closed relation. A relation $\mathcal{R}$ is reduction closed iff $P \mathcal{R} Q$ and $P \triangleright P'$ then there exists some Q' such that $Q \triangleright Q'$ and $P' \mathcal{R} Q'$. A relation $\mathcal{R}$ is context closed iff $P \mathcal{R} Q$ yields that $CP \mathcal{R} CQ$, for all contexts C.*

Bisimulation is sound with respect to contextual equivalence. Soundness is essential to justify the chosen notion of bisimulation.

Theorem 1 (Bisimulation is a contextual equivalence)**.** *If $P \sim Q$ then $P \simeq Q$.*

Proof. Reduction closure follows from Lemma 2 and context closure follows from Lemma 3. □

Soundness of bisimulation ensures that algebraic properties proven using bisimulation also hold for contextual equivalence. Bisimulation simplifies proofs in the following section. Note that completeness (contextual equivalence is a bisimulation) is not required for this work. Completeness can only be achieved in an extended version of the calculus.

4.2 Algebraic properties of queries

Using bisimulation as an equivalence, key properties of queries are established. This section amounts to a soundness proof of the algebraic properties established. Thus if any two process are equivalent according to the algebraic properties then they are bisimilar; and furthermore, by Theorem 1, they are contextually equivalent.

For the labelled transition system, structural congruence is not assumed, hence verified here. The proof for the distributivity of blank node quantifiers over par requires extensive case analysis. The case of associativity of par follows from distributivity of blank node quantifiers. Proofs are similar to the analogous bisimulations in the π-calculus [22].

Proposition 2. *The structural congruence (Fig.2) is a bisimulation. So,* $(P, \parr, \bot)$ *forms a commutative monoid. Blank node quantifiers annihilate with* $\bot$*, commute, and distribute over* $\parr$*.*

Bisimulation reveals some canonical algebraic properties of queries. Firstly, queries form an idempotent semiring. Semirings are ubiquitous in computer science. A notable feature of semirings is that the ideals of a semiring form a semiring.

Proposition 3. $(U, \otimes, \oplus, \mathrm{I}, 0)$ *is a commutative idempotent semiring. That is,* $(U, \otimes, \mathrm{I})$ *is a commutative monoid,* $(U, \oplus, 0)$ *is idempotent commutative monoid.* $\otimes$ *distributes over* $\oplus$ *and* 0 *annihilates with* $\otimes$*.*

Idempotent semirings have a natural preorder, given by $U \leq V$ iff $U \oplus V \sim V$. Hence queries have this natural preorder. An immediate consequence is that choice is a colimit, i.e. least upper bound, of two queries.

Proposition 4. *Choice is a colimit of its branches. That is,* $V \leq W$ *and* $U \leq W$*, if and only if* $V \oplus U \leq W$*.*

The preorder over queries can be used to optimise queries. If a query offers a choice between a query and a weaker query, with respect to the preorder, the stronger branch may be eliminated. For instance, in related work [23], is is claimed that U `OPTIONAL` (V `OPTIONAL` W) is not the same as (U `OPTIONAL` V) `OPTIONAL` W. Under the interpretation of `OPTIONAL` in the calculus it holds that $U \otimes ((V \otimes (W \oplus \mathrm{I})) \oplus \mathrm{I}) \leq U \otimes ((V \oplus \mathrm{I}) \otimes (W \oplus \mathrm{I}))$, by distributivity, commutativity and idempotency. So the first is a stronger query.

A single rule is sufficient to capture the algebra of the select quantifier. From this algebra common equalities can be derived. The derived rules are suitable for the optimisation technique of flattening nested selects used in relational algebra [8]. The proof of commutativity of quantifiers requires capture avoiding substitution to be assumed. The presence of the tensor in the rule is required to prove that $\bigvee a.U \otimes V \leq \bigvee a.(U \otimes V)$, when $a \notin \mathrm{fn}(V)$.

Proposition 5. *Selects are colimits of substitutions. So,* $U\left\{{}^{b}/_{a}\right\} \otimes V \leq W$ *for all* b*, if and only if* $\bigvee a.U \otimes V \leq W$*. Immediate consequences are that, select commutes, distributes over choice, is annihilated by true and distributes over tensor. Furthermore, alpha conversion of bound variables is verified.*

The following rules of regular algebra hold. The first of the rules is sufficient to demonstrate that $*V \otimes U$ is a fixed point of the (monotone) map $W \mapsto U \oplus (V \otimes W)$. The second rule demonstrates that $*V \otimes U$ is the least such fixed point. Historically, Redko demonstrated that no finite collection of equations could axiomatise iteration [25]. The formulation below, was proven to be complete by Kozen [19].

Proposition 6. *An iterated query expands as follows* $*U \sim \mathrm{I} \oplus (U \otimes *U)$*. Furthermore, if* $U \oplus (V \otimes W) \leq W$ *then* $*V \otimes U \leq W$*.*

A classic consequence of the above is that queries without select can always be denested to a single iteration [20]. However, select breaks denesting since iteration and select do not commute. For instance the following query requires two iterations. The result is that for each of the first continuation triggered, zero or more instances of the second continuation are triggered. This query can be expressed using sub-queries in the current SPARQL Query working draft [10].

$$*\bigvee a.\bigvee n.\Big(((a\ \mathit{name}\ n)\,;P) \otimes *\bigvee e.((a\ \mathit{email}\ e)\,;Q)\Big)$$

Iteration can be expressed as a colimit of repeated queries. This is a strictly more general property than Proposition 6 [18]. Since all constructs are colimits which distribute over tensor, the ideals generated by queries form a (commutative) quantale, as exploited by Montanari, Hoare and others [5, 12]. Quantales are related to spectral theory, which is related to information retrieval techniques used by search engines. Clarification of this connection is future work.

Proposition 7. *Iteration is a colimit of powers of queries. So, $U^n \otimes V \leq W$ for all n, if and only if $*U \otimes V \leq W$.*

Kozen demonstrates that Boolean algebras can be embedded in Kleene algebras [20]. The 'tests' of Kozen correspond to 'constraints' in SPARQL. Bisimulation verifies that the Boolean algebra of constraints embeds in the Kleene algebra, in the same manner, with similar consequences.

Proposition 8. *The Boolean algebra of constraints embeds in queries. Using standard classical implication, $\phi \Rightarrow \psi$ if and only if $\phi \leq \psi$. Or is choice, and is tensor, exists is select and an iterated constraint is always true.*

As with classical implication, the preorder over triples can be embedded in the partial order over processes. However, since alias assumptions are only a preorder, if $C \sim D$ then it holds that $C \sqsubseteq D$ and $D \sqsubseteq C$, which is weaker than equality. Maintaining distinction of names is important for applications where β is not fixed over time.

Proposition 9. *$C \sqsubseteq D$ if and only if $C \leq D$.*

The multiplicatives then, par and times and the units are related in the following manner. Combined with the previous rules the properties of then are established. The second rule shows that 'then' can be replaced by the unit delay (as in [1]).

Proposition 10. *An empty continuation can be removed, a continuation can be decomposed into the guard and a unit delayed process, and two continuations can be combined in a single par continuation, as follows.*

$$\mathrm{I}\,;\bot \sim \mathrm{I} \qquad U \otimes (\mathrm{I}\,;P) \sim U\,;P \qquad (U\,;P)\,;Q \sim U\,;(P \parr Q)$$

The algebra can be applied to optimise queries for distribution. In the example below the first query is rewritten as the tensor product of two queries.

$$*\bigvee a.(((a \textit{ knows } b_2)\,;P) \oplus ((a \textit{ knows } b_3)\,;Q)) \sim *\bigvee a.((a \textit{ knows } b_2)\,;P) \otimes *\bigvee a.((a \textit{ knows } b_3)\,;Q)$$

The second query above is better for distribution. The tensor product allows two smaller queries to be immediately evaluated in parallel. The tighter scope of the select quantifiers reduces the branching when potential values to select are considered. The distribution of queries across clusters of servers is a major problem for processing Linked Data [11].

5 Conclusion

The calculus introduced provides the first operational semantics for SPARQL Query -- a W3C recommendation for querying Linked Data. The calculus has a concise logical semantics defined by a reduction system. The power of the calculus lies in the synchronisation primitives for queries. The synchronisation primitives are required to match the expressiveness of the core of SPARQL Query. Queries are internalised in a high-level process calculus, where query results determine continuation processes.

An alternative labelled transition system is shown to match the expressive power of the reduction system. Furthermore, the notion of bisimulation in the labelled transition system is sound with respect to equivalence in the reduction system. Bisimulation is used to verify an algebra over queries, which extends existing notions of an algebra for SPARQL Query. An algebra of queries is useful when tackling problems associated with Linked Data, such as distributed query planning.

The operational semantics combines several formalisms, as expected for a real language. The queries form a semiring, which provides a natural partial order. This partial order is used to characterise choice, selects and iteration as colimits. Also, iteration is the least fixed point of a monotonic map over queries, hence queries form a Kleene algebra. A preorder over URIs allows small permissible mismatches between content and queries to be resolved, capturing key features of the RDFS standard. Also, a Boolean algebra of constraints is naturally embedded in queries, to provide further control. The calculus demonstrates that key features of SPARQL and related standards for Linked Data can be tightly integrated in one framework.

References

[1] Samson Abramsky, Simon Gay & Rajagopal Nagarajan (1995): *Interaction Categories and the Foundations of Typed Concurrent Programming*. In Manfred Broy, editor: *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, Springer, pp. 35--114.

[2] Harith Alani et al. (2002): *Managing Reference: Ensuring Referential Integrity of Ontologies for the Semantic Web*. In Gómez-Pérez et al., editors: *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, 2473, Springer, pp. 235--246, doi:10.1007/3-540-45810-7_29.

[3] G. Bellin & P. J. Scott (1994): *On the π-Calculus and Linear Logic*. *Theoretical Computer Science* 135, pp. 11--65, doi:10.1016/0304-3975(94)00104-9.

[4] Paul V. Biron & Ashok Malhotra (2004): *XML Schema part 2: Datatypes Second Edition*. W3C, MIT, Cambridge, MA. REC-xmlschema-2-20041028.

[5] Stefano Bistarelli, Ugo Montanari & Francesca Rossi (1997): *Semiring-Based Constraint Satisfaction and Optimization*. *Journal of the ACM* 44(2), pp. 201--236, doi:10.1145/256303.256306.

[6] Christian Bizer (2009): *The Emerging Web of Linked Data*. *IEEE Intelligent Systems* 24, pp. 87--92, doi:10.1109/MIS.2009.102.

[7] Dan Brickley & R.V. Guha (2004): *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C, MIT, Cambridge, MA. REC-rdf-schema-20040210.

[8] Richard Cyganiak (2005): *A relational algebra for SPARQL*. Hewlit Packard Labs, Bristol. HPL-2005-170.

[9] Jean-Yves Girard (1987): *Linear Logic*. *Theoretical Computer Science* 50(1), pp. 1--112, doi:10.1016/0304-3975(87)90045-4.

[10] Steve Harris, Andy Seaborne & Eric Prud'hommeaux (2010): *SPARQL 1.1 Query Language*. W3C, MIT, Cambridge, MA. WD-sparql11-query-20101014.

[11] Olaf Hartig et al. (2009): *Executing SPARQL Queries over the Web of Linked Data*. In A. Bernstain et al., editors: *The Semantic Web -- ISWC 2009, Chantilly, VA*, 5823, Springer, pp. 293--309, doi:10.1007/978-3-642-04930-9_19.

[12] C. A. R. Tony Hoare, Bernhard Möller, Georg Struth & Ian Wehrman (2009): *Concurrent Kleene Algebra*. In Mario Bravetti & Gianluigi Zavattaro, editors: *CONCUR 2009, Bologna, Italy*, 5710, Springer, pp. 399--414, doi:10.1007/978-3-642-04081-8_27.

[13] Joshua S. Hodas & Dale Miller (1994): *Logic Programming in a Fragment of Intuitionistic Linear Logic*. *Information and Computation* 110(2), pp. 327--365.

[14] Ross Horne (2011): *Programming Languages and Principles for Read--Write Linked Data*. Ph.D. thesis, Electronics and Computer Science, University of Southampton.

[15] Alan Jeffrey & Julian Rathke (2005): *Contextual equivalence for higher-order π-calculus revisited*. *Logical Methods in Computer Science* 1(4), pp. 1--22, doi:10.2168/LMCS-1(1:4)2005.

[16] Graham Klyne & Jeremy Carroll (2004): *Resource Description Framework: Concepts and Abstract Syntax*. W3C, MIT, Cambridge, MA. REC-rdf-concepts-20040210.

[17] Naoki Kobayashi & Akinori Yonezawa (1993): *ACL -- A Concurrent Linear Logic Programming Paradigm*. In: *Proceedings of the 1993 International Logic Programming Symposium*, MIT Press, pp. 279--294.

[18] Dexter Kozen (1990): *On Kleene algebras and Closed Semirings*. In Rovan, editor: *Proceedings on Mathematical Foundations of Computer Science*, 452, Springer-Verlag, pp. 26--47.

[19] Dexter Kozen (1994): *A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events*. *Information and Computation* 110, pp. 366--390, doi:10.1006/inco.1994.1037.

[20] Dexter Kozen (1997): *Kleene algebra with tests*. *ACM Transactions on Programing Languages and Systems* 19, pp. 427--443, doi:10.1145/256167.256195.

[21] Massimo Merro & Matthew Hennessy (2002): *Bisimulation congruences in safe ambients*. In: *Principles of programming languages*, ACM, pp. 71--80, doi:10.1145/503272.503280.

[22] Robin Milner, Joachim Parrow & David Walker (1992): *A calculus of mobile processes, part I and II*. *Information and Computation* 100(1), pp. 1--40, doi:10.1016/0890-5401(92)90008-4.

[23] Jorge Pérez, Marcelo Arenas & Claudio Gutierrez (2009): *Semantics and Complexity of SPARQL*. *ACM Transactions on Database Systems* 34(3), pp. 1--45, doi:10.1145/1567274.1567278.

[24] Eric Prud'hommeaux & Andy Seaborne (2008): *SPARQL Query Language for RDF*. W3C, MIT, Cambridge, MA. REC-rdf-sparql-query-20080115.

[25] V. N. Redko (1964): *On defining relations for the algebra of regular events*. *Ukrainskii Matematicheskii Zhurnal* , pp. 120--126.

A State-Based Characterisation of the Conflict Preorder

Simon Ware Robi Malik
Department of Computer Science, University of Waikato, Hamilton, New Zealand
{siw4,robi}@waikato.ac.nz

This paper proposes a way to effectively compare the potential of processes to cause *conflict*. In discrete event systems theory, two concurrent systems are said to be in conflict if they can get trapped in a situation where they are both waiting or running endlessly, forever unable to complete their common task. The *conflict preorder* is a process-algebraic pre-congruence that compares two processes based on their possible conflicts in combination with other processes. This paper improves on previous theoretical descriptions of the conflict preorder by introducing *less conflicting pairs* as a concrete state-based characterisation. Based on this characterisation, an effective algorithm is presented to determine whether two processes are related according to the conflict preorder.

1 Introduction

A key question in process algebra is how processes can be composed and compared [4, 6]. An understanding of what makes processes equivalent is important for several applications, ranging from comparison and minimisation in model checking to program construction using abstraction and refinement. Several equivalence relations have been studied, most notably *observation equivalence* [12], *failures equivalence* [7], and *trace equivalence* [7]. Each equivalence has its own properties, making it suitable for particular applications and verification tasks [6].

This paper focuses on *conflict equivalence*, which compares processes based on which other processes they can come into conflict [3, 14] with. Two processes are in conflict, if they can reach a state from which termination is no longer possible. This can be because of *deadlock* where neither process is capable of doing anything, or *livelock* where the system continues to run without ever terminating.

It is difficult to reason about conflicts in a modular way. If two processes are free from conflict individually, they may well be involved in a conflict when running together, and vice versa [18]. This makes it difficult to apply most methods of abstraction common in model checking [1] to verify systems to be free from conflict, and standard process-algebraic equivalences [6] are not applicable either.

Conflict equivalence is introduced in [11] as the best possible process equivalence to reason compositionally about conflicts. Conflict equivalence is coarser than observation equivalence [12] and different from failures and trace equivalence [7]. The process-algebraic theory most closely related to conflict equivalence is *fair testing* [2, 13, 15]. The essential difference between conflict equivalence and fair testing lies in the capability to compare processes that exhibit blocking behaviour, as expressed by the *set of certain conflicts* [9, 10, 11].

In [5, 16, 17], various conflict-preserving rewrite rules are used to simplify processes and check whether or not large systems of concurrent finite-state automata are free from conflict. While of good use in practice, the rewrite rules are incomplete, and it remains an open question how processes can be normalised or compared for conflict equivalence.

This paper improves on previous results about conflict equivalence and the associated conflict preorder [11], and fair testing [15], by providing a state-based characterisation of the conflict preorder. It

M.R. Mousavi and A. Ravara: FOCLASA 2011
EPTCS 58, 2011, pp. 34–48, doi:10.4204/EPTCS.58.3

proposes *less conflicting pairs* as a more concrete way to compare processes for their conflicting behaviour than the abstract test-based characterisation using *nonconflicting completions* in [11] and the *refusal trees* of [15]. Less conflicting pairs give a means to directly compare processes based on their reachable state sets, which leads to an alternative algorithm to test the conflict preorder. While still linear exponential, this algorithm is simpler and has better time complexity than the decision procedure for fair testing [15].

In the following, Section 2 briefly reviews the needed terminology of languages, automata, and conflict equivalence. Then Section 3 introduces less conflicting pairs and shows how they can be used to describe certain conflicts and the conflict preorder. Afterwards, Section 4 proposes an algorithm to calculate less conflicting pairs for finite-state automata, and Section 5 adds some concluding remarks.

2 Preliminaries

2.1 Languages and Automata

Event sequences and languages are a simple means to describe process behaviours. Their basic building blocks are *events*, which are taken from a finite *alphabet* Σ. Two special events are used, the *silent event* τ and the *termination event* ω. These are never included in an alphabet Σ unless mentioned explicitly.

Σ^* denotes the set of all finite *traces* of the form $\sigma_1\sigma_2\cdots\sigma_n$ of events from Σ, including the *empty trace* ε. The *length* of trace s is denoted by $|s|$. A subset $L \subseteq \Sigma^*$ is called a *language*. The *concatenation* of two traces $s,t \in \Sigma^*$ is written as st, and a trace s is called a *prefix* of t, written $s \sqsubseteq t$, if $t = su$ for some trace u. A language $L \subseteq \Sigma^*$ is *prefix-closed*, if $s \in L$ and $r \sqsubseteq s$ implies $r \in L$.

In this paper, process behaviour is modelled using nondeterministic *labelled transitions systems* or *automata* $A = \langle \Sigma, Q, \to, Q^\circ \rangle$, where Σ is a finite alphabet of *events*, Q is a set of *states*, $\to \subseteq Q \times (\Sigma \cup \{\tau,\omega\}) \times Q$ is the *state transition relation*, and $Q^\circ \subseteq Q$ is the set of *initial states*. The automaton A is called *finite-state* if its state set Q is finite.

The transition relation is written in infix notation $x \xrightarrow{\sigma} y$, and is extended to traces by letting $x \xrightarrow{\varepsilon} x$ for all $x \in Q$, and $x \xrightarrow{s\sigma} y$ if $x \xrightarrow{s} z \xrightarrow{\sigma} y$ for some $z \in Q$. The transition relation must satisfy the additional requirement that, whenever $x \xrightarrow{\omega} y$, there does not exist any outgoing transition from y. The automaton A is called *deterministic* if $|Q^\circ| \leq 1$ and the transition relation contains no transitions labelled τ, and if $x \xrightarrow{\sigma} y_1$ and $x \xrightarrow{\sigma} y_2$ always implies $y_1 = y_2$.

To support silent transitions, $x \stackrel{s}{\Rightarrow} y$, with $s \in (\Sigma \cup \{\omega\})^*$, denotes the existence of a trace $t \in (\Sigma \cup \{\omega,\tau\})^*$ such that $x \xrightarrow{t} y$, and s is obtained from t by deleting all τ events. For a state set $X \subseteq Q$ and a state $y \in Q$, the expression $X \stackrel{s}{\Rightarrow} y$ denotes the existence of $x \in X$ such that $x \stackrel{s}{\Rightarrow} y$, and $A \stackrel{s}{\Rightarrow} y$ means that $Q^\circ \stackrel{s}{\Rightarrow} y$. Furthermore, $x \Rightarrow y$ denotes the existence of a trace s such that $x \stackrel{s}{\Rightarrow} y$, and $x \stackrel{s}{\Rightarrow}$ denotes the existence of a state $y \in Q$ such that $x \stackrel{s}{\Rightarrow} y$. For a state, state set, or automaton $\mathbf{X}$, the *language* and the *marked language* are

$$\mathbf{L}(\mathbf{X}) = \{\, s \in (\Sigma \cup \{\omega\})^* \mid \mathbf{X} \stackrel{s}{\Rightarrow} \,\} \qquad \text{and} \qquad \mathbf{L}^\omega(\mathbf{X}) = \mathbf{L}(\mathbf{X}) \cap \Sigma^*\omega \,. \tag{1}$$

Every prefix-closed language L is recognised by an automaton A such that $\mathbf{L}(A) = L$, but only *regular* languages are recognised by a finite-state automaton [8].

When two automata are running in parallel, lock-step synchronisation in the style of [7] is used. The *synchronous composition* of $A = \langle \Sigma_A, Q_A, \to_A, Q_A^\circ \rangle$ and $B = \langle \Sigma_B, Q_B, \to_B, Q_B^\circ \rangle$ is

$$A \parallel B = \langle \Sigma_A \cup \Sigma_B, Q_A \times Q_B, \to, Q_A^\circ \times Q_B^\circ \rangle \tag{2}$$

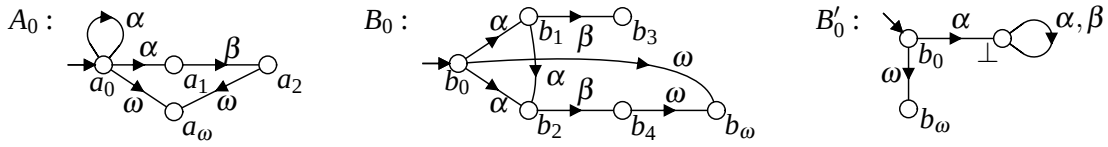


Figure 1: Examples of blocking and nonblocking automata.

where

$$
\begin{array}{ll}
(x_A, x_B) \xrightarrow{\sigma} (y_A, y_B) & \text{if } \sigma \in (\Sigma_A \cap \Sigma_B) \cup \{\omega\},\ x_A \xrightarrow{\sigma}_A y_A, \text{ and } x_B \xrightarrow{\sigma}_B y_B\ ; \\
(x_A, x_B) \xrightarrow{\sigma} (y_A, x_B) & \text{if } \sigma \in (\Sigma_A \setminus \Sigma_B) \cup \{\tau\} \text{ and } x_A \xrightarrow{\sigma}_A y_A\ ; \\
(x_A, x_B) \xrightarrow{\sigma} (x_A, y_B) & \text{if } \sigma \in (\Sigma_B \setminus \Sigma_A) \cup \{\tau\} \text{ and } x_B \xrightarrow{\sigma}_B y_B\ .
\end{array}
$$

In synchronous composition, shared events (including ω) must be executed by all automata together, while events used by only one of the composed automata and silent (τ) events are executed independently.

2.2 Conflict Equivalence

The key liveness property in supervisory control theory [14] is the *nonblocking* property. Given an automaton A, it is desirable that every trace in $\mathbf{L}(A)$ can be completed to a trace in $\mathbf{L}^\omega(A)$, otherwise A may become unable to terminate. A process that may become unable to terminate is called *blocking*. This concept becomes more interesting when multiple processes are running in parallel—in this case the term *conflicting* is used instead.

Definition 1. An automaton $A = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$ is *nonblocking* if for every state $x \in Q$, $Q^\circ \Rightarrow x$ implies that $\mathbf{L}^\omega(x) \neq \emptyset$. Otherwise A is *blocking*. Two automata A and B are *nonconflicting* if $A \parallel B$ is nonblocking, otherwise they are *conflicting*.

Example 1. Automaton A_0 in Figure 1 is nonblocking, as it is always possible to reach state a_2 and terminate. Automaton B_0 on the other hand is blocking, because it can enter state b_3 after execution of $\alpha\beta$, from where it is no longer possible to reach a state where the termination event ω is enabled.

For an automaton to be nonblocking, it is enough that a terminal state *can* be reached from *every* reachable state. There is no requirement for termination to be guaranteed. For example, automaton A_0 in Figure 1 is nonblocking despite the presence of a possibly infinite loop of α-transitions in state a_0. Nonblocking is also different from "may"-testing [15], which only requires the possibility of termination from the initial state. The testing semantics most similar to nonblocking is "should"-testing, which is also known as *fair testing* [15].

To reason about nonblocking in a compositional way, the notion of *conflict equivalence* is developed in [11]. According to process-algebraic testing theory, two automata are considered as equivalent if they both respond in the same way to all tests of a certain type [4]. For conflict equivalence, a *test* is an arbitrary automaton, and the *response* is the observation whether or not the test is conflicting with the automaton in question.

Definition 2. Let A and B be two automata. A is *less conflicting* than B, written $A \lesssim_{\text{conf}} B$, if, for every automaton T, if $B \parallel T$ is nonblocking then $A \parallel T$ also is nonblocking. A and B are *conflict equivalent*, $A \simeq_{\text{conf}} B$, if $A \lesssim_{\text{conf}} B$ and $B \lesssim_{\text{conf}} A$.

Example 2. Consider automata A_1 and B_1 in Figure 2. A_1 is *not* less conflicting than B_1, since $A_1 \parallel T_1$ is blocking while $B_1 \parallel T_1$ is nonblocking. This is because $A_1 \parallel T_1$ can enter the blocking state (a_2, q_1) after executing of α, whereas after executing α in B_1, it eventually becomes possible to continue using either the β- or γ-transition of T_1. It can also be shown that $B_1 \lesssim_{\text{conf}} A_1$ does not hold.

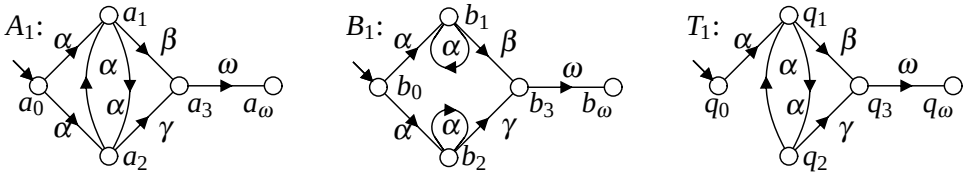


Figure 2: Two automata that are not conflict equivalent.

The properties of the conflict preorder $\lesssim_{\text{conf}}$ and of conflict equivalence and their relationship to other process-algebraic relations are studied in [11]. It is enough to consider deterministic tests in Definition 2, and conflict equivalence is is the coarsest possible congruence with respect to synchronous composition that respects blocking, making it an ideal equivalence for use in compositional verification [5, 17].

2.3 The Set of Certain Conflicts

Every automaton can be associated with a language of *certain conflicts*, which plays an important role in conflict semantics [9].

Definition 3. For an automaton $A = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$, write

$$\textsc{Conf}(A) = \{\, s \in \Sigma^* \mid \text{For every automaton } T \text{ such that } T \stackrel{s}{\Rightarrow}, A \parallel T \text{ is blocking} \,\}\,; \tag{3}$$

$$\textsc{NConf}(A) = \{\, s \in \Sigma^* \mid \text{There exists an automaton } T \text{ such that } T \stackrel{s}{\Rightarrow} \text{ and } A \parallel T \text{ is nonblocking} \,\}\,. \tag{4}$$

$\textsc{Conf}(A)$ is the set of *certain conflicts* of A. It contains all traces that, when possible in the environment, necessarily cause blocking. Its complement $\textsc{NConf}(A)$ is the most general behaviour of processes that are to be nonconflicting with A. If A is nonblocking, then $\textsc{Conf}(A) = \emptyset$ and $\textsc{NConf}(A) = \Sigma^*$, because in this case $A \parallel U$ is nonblocking, where U is a deterministic automaton such that $\mathbf{L}^\omega(U) = \Sigma^*\omega$. The set of certain conflicts becomes more interesting for blocking automata.

Example 3. Consider again automaton B_0 in Figure 1. Clearly $\alpha\beta \in \textsc{Conf}(B_0)$ as B_0 can enter the deadlock state b_3 by executing $\alpha\beta$, and therefore every test T that can execute $\alpha\beta$ is conflicting with B_0. But also $\alpha \in \textsc{Conf}(B_0)$, because B_0 can enter state b_2 by executing α, from where the only possibility to terminate is by executing $\beta\omega$. So any test that can execute α also needs to be able to execute $\alpha\beta$ if it is to be nonconflicting with B_0; but such a test is conflicting with B_0 as explained above. It can be shown that $\textsc{Conf}(B_0) = \alpha\Sigma^*$.

The set of certain conflicts is introduced in [9], and its properties and its relationship to conflict equivalence are studied in [11]. Even if an automaton is nondeterministic, its set of certain conflicts is a *language*, but as shown in Example 3, it is not necessarily a subset of the language $\mathbf{L}(A)$ of its automaton. If a trace s is a trace of certain conflicts, then so is any extension st. An algorithm to compute the set of certain conflicts for a given finite-state automaton is presented in [10].

Certain conflicts constitute the main difference between conflict equivalence and *fair testing* [15]. In fair testing, processes are not allowed to synchronise on the termination event ω, so termination is determined solely by the test. This can be expressed as conflict equivalence by requiring that ω be enabled in all states of the automata compared [11].

Conversely, it is possible to factor out certain conflicts from any given automaton, by redirecting all traces of certain conflicts to a single state [9, 10]. For example, automaton B_0 in Figure 1 can be replaced by the conflict equivalent automaton B_0', which uses the single deadlock state $\perp$. Two automata A and B are conflict equivalent if and only if their normalised forms A' and B' are fair testing equivalent. The decision procedure for fair testing [15] can be used to test the conflict preorder, and vice versa.

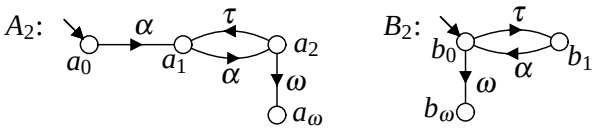


Figure 3: Two automata that are conflict equivalent.

3 Characterising the Conflict Preorder

This section is concerned about characterising two automata A and B as conflict equivalent, or characterising A as less conflicting than B, in a state-based way. First, 3.1 explains the crucial properties of conflict equivalence using examples. *Less conflicting pairs* are introduced in 3.2, and they are used to characterise certain conflicts in 3.3 and the conflict preorder in 3.4.

3.1 Understanding Conflict Equivalence

Every reachable state of an automaton A carries a *nonblocking requirement* (also known as a *nonconflicting completion* [11]) that needs to be satisfied by tests that are to be nonconflicting with A. For example, if $A \stackrel{s}{\Rightarrow} x_A$, then every test T that can execute s needs to be able to continue with at least one trace $t \in \mathbf{L}^\omega(x_A)$, or T is conflicting with A. An automaton A is less conflicting than another automaton B, if every nonblocking requirement associated with A also is a nonblocking requirement associated with B.

Example 4. Consider again automata A_1 and B_1 in Figure 2. They have the same marked languages. Thus, if the initial state a_0 of A_1 is blocking in combination with some test T, then so is the initial state b_0 of B_1. But this is not the case when $A_1 \parallel T$ enters a state (a_1, x_T) after execution of α. State a_1 requires x_T to be capable of performing at least one trace from the language $\mathbf{L}^\omega(a_1) = (\alpha\alpha)^*\beta\omega + (\alpha\alpha)^*\alpha\gamma\omega$, whereas the states b_1 and b_2, which can both be entered after executing α, require a trace from the language $\alpha^*\beta\omega$ and $\alpha^*\gamma\omega$, respectively. Both of these languages contain traces outside of the language $\mathbf{L}^\omega(a_1)$. Automaton T_1 in Figure 2 is in conflict with A_1 but not with B_1.

In general, it is not enough to compare only the marked languages of states reached by equal traces. Not every nonblocking requirements is a marked language of some state of its automaton. The following example shows one of the problems.

Example 5. Consider automata A_2 and B_2 in Figure 3. The marked language of the initial state of A_2 is $\mathbf{L}^\omega(a_0) = \alpha\alpha^+\omega$, while the marked languages of the two states in B_2 that can be entered initially are $\mathbf{L}^\omega(b_0) = \alpha^*\omega$ and $\mathbf{L}^\omega(b_1) = \alpha^+\omega$. Although the marked languages are different, for any automaton T, if $B_2 \parallel T$ is nonblocking, then $A_2 \parallel T$ must also be nonblocking. If T is to be nonconflicting in combination with B_2, since B_2 may initially enter state b_1, there must be the possibility to continue with event α. However, after executing α, automaton B_2 may again silently enter state b_1, which means that α must be possible again. This is enough to ensure that $A_2 \parallel T$ is nonblocking. Using this argument, it can be shown that A_2 and B_2 are conflict equivalent.

3.2 Less Conflicting Pairs

In order to compare two nondeterministic automata according to conflicts, it is necessary to identify sets of states the two automata may reach under the same input. This is done using the well-known *subset construction* [8]. To capture termination, the usual powerset state space is extended by a special state ω entered only after termination.

Definition 4. The *deterministic state space* of automaton $A = \langle \Sigma, Q, \to, Q^\circ \rangle$ is

$$Q_A^{\det} = 2^Q \cup \{\omega\} , \tag{5}$$

and the *deterministic transition function* $\delta_A^{\det} \colon Q^{\det} \times (\Sigma \cup \{\omega\}) \to Q^{\det}$ for A is defined as

$$\delta_A^{\det}(X, \sigma) = \begin{cases} \omega, & \text{if } \sigma = \omega \text{ and } X \stackrel{\omega}{\Rightarrow}; \\ \{y \in Q \mid X \stackrel{\sigma}{\Rightarrow} y\}, & \text{otherwise.} \end{cases} \tag{6}$$

The deterministic transition function $\delta_A^{\det}$ is extended to traces $s \in \Sigma^* \cup \Sigma^*\omega$ in the standard way. Note that $\delta_A^{\det}(X, s)$ is defined for every trace $s \in \Sigma^* \cup \Sigma^*\omega$; if none of the states in X accepts the trace s, this is indicated by $\delta_A^{\det}(X, s) = \emptyset$. This is also true for termination: if ω is enabled in some state in X, then $\delta_A^{\det}(X, \omega) = \omega$, otherwise $\delta_A^{\det}(X, \omega) = \emptyset$.

In order to compare two automata A and B with respect to possible conflicts, *pairs* of state sets of the subset construction of A and B need to be considered. Therefore, the deterministic transition function is also applied to pairs $\mathbf{X} = (X_A, X_B)$ of state sets $X_A \subseteq Q_A$ and $X_B \subseteq Q_B$,

$$\delta_{A,B}^{\det}(\mathbf{X}, s) = \delta_{A,B}^{\det}(X_A, X_B, s) = (\delta_A^{\det}(X_A, s), \delta_B^{\det}(X_B, s)) . \tag{7}$$

To determine whether $A \lesssim_{\text{conf}} B$, it is necessary to check all states $x_A \in Q_A$ against matching state sets $X_B \subseteq Q_B$ and determine whether all possible conflicts of x_A are also present in X_B. For example, when automaton A_2 in Figure 3 is in state a_1, then B_2 may be in b_0 or b_1. In state a_1, at least one of the traces in $\alpha^+\omega$ needs to be enabled to avert blocking, and the same requirement to avert blocking is seen in state b_1. When state a_1 is entered with some test T, blocking occurs if none of the traces in $\alpha^+\omega$ is enabled, and such a test T is also blocking when combined with a system that may be in b_0 or b_1. Therefore, a_1 is considered in the following as *less conflicting* (**LC**) than $\{b_0, b_1\}$.

It cannot always be determined directly whether a state $x_A \in Q_A$ is less conflicting than a state set $X_B \subseteq Q_B$. In some cases, it is necessary also to consider the deterministic successors of x_A and X_B. Therefore, the following definition considers pairs (X_A, X_B) of state sets.

Definition 5. Let $A = \langle \Sigma, Q_A, \to_A, Q_A^\circ \rangle$ and $B = \langle \Sigma, Q_B, \to_B, Q_B^\circ \rangle$ be automata. The set $\mathbf{LC}(A, B) \subseteq Q_A^{\det} \times Q_B^{\det}$ of *less conflicting pairs* for A and B is inductively defined by

$$\mathbf{LC}^0(A, B) = \{\omega\} \times Q_B^{\det} \cup \{ (X_A, X_B) \mid X_B \subseteq Q_B \text{ and there exists } x_B \in X_B \text{ with } \mathbf{L}^\omega(x_B) = \emptyset \} ; \tag{8}$$

$$\mathbf{LC}^{n+1}(A, B) = \{ (X_A, X_B) \mid \text{there exists } x_B \in X_B \text{ such that for all } t \in \Sigma^*, \text{ if } x_B \stackrel{t\omega}{\Rightarrow} \text{ then there exists } r \sqsubseteq t\omega \text{ such that } \delta_{A,B}^{\det}(X_A, X_B, r) \in \mathbf{LC}^i(A, B) \text{ for some } i \leq n \} ; \tag{9}$$

$$\mathbf{LC}(A, B) = \bigcup_{n \geq 0} \mathbf{LC}^n(A, B) . \tag{10}$$

Remark 1. If $(X_A, X_B) \notin \mathbf{LC}(A, B)$, then according to (9), for every state $x_B \in X_B$, there exists $t \in \Sigma^*$ such that $x_B \stackrel{t\omega}{\Rightarrow}$, and $\delta^{\det}(X_A, X_B, r) \notin \mathbf{LC}(A, B)$ for all prefixes $r \sqsubseteq t\omega$.

The idea of Definition 5 is to classify a pair (X_A, X_B) as less conflicting, if the marked language of X_A is a *nonconflicting completion* [11] for the process with initial states X_B. That is, every test that is nonconflicting in combination with each of the states in X_B can terminate with at least one trace from the marked language of X_A. Or conversely, every test that cannot terminate using any of the traces in the marked language of X_A also is conflicting with X_B (see Lemma 1 below).

The first state set X_A of a pair (X_A, X_B) is just used to represent a *language* of possible completions. If state sets X_A and Y_A have the same languages, then all pairs (X_A, X_B) and (Y_A, X_B) have exactly the same less conflicting status. For the second state set X_B on the other hand, the complete nondeterministic behaviour is relevant.

A pair (ω, X_B) is considered as "less conflicting" (8), since termination has already been achieved in A. If X_B contains a state x_B such that $\mathbf{L}^\omega(x_B) = \emptyset$, then (X_A, X_B) also is less conflicting (8), because conflict is guaranteed in X_B. For other pairs (X_A, X_B), it must be checked whether X_B contains a requirement to avert blocking matching that given by the language of X_A (9).

Example 6. Consider again automata A_0 and B_0 in Figure 1. It holds that $(\{a_0\}, \{b_0\}) \in \mathbf{LC}^1(A_0, B_0)$. There are three ways to terminate from b_0, by executing ω or $\alpha\beta\omega$ or $\alpha\alpha\beta\omega$. All three traces are possible in a_0, each taking the pair $(\{a_0\}, \{b_0\})$ to the deterministic successor $(\omega, \omega) \in \mathbf{LC}^0(A_0, B_0)$. This is enough to confirm that (9) is satisfied.

On the other hand, $(\{a_0\}, \{b_2\}) \notin \mathbf{LC}^1(A_0, B_0)$. From state a_0, blocking occurs with a test T that can only execute $\beta\omega$, but this test is nonblocking with b_2. It holds that $b_2 \stackrel{\beta\omega}{\rightarrow}$, where trace $\beta\omega$ has the prefixes ε, β, and $\beta\omega$, but $\delta^{\text{det}}_{A_0,B_0}(\{a_0\}, \{b_2\}, \varepsilon) = (\{a_0\}, \{b_2\}) \notin \mathbf{LC}^0(A_0, B_0)$, $\delta^{\text{det}}_{A_0,B_0}(\{a_0\}, \{b_2\}, \beta) = (\emptyset, \{b_4\}) \notin \mathbf{LC}^0(A_0, B_0)$, and $\delta^{\text{det}}_{A_0,B_0}(\{a_0\}, \{b_2\}, \beta\omega) = (\emptyset, \omega) \notin \mathbf{LC}^0(A_0, B_0)$. Therefore, (9) is not satisfied and $(\{a_0\}, \{b_2\}) \notin \mathbf{LC}^1(A_0, B_0)$. It can also be shown that $(\{a_0\}, \{b_2\}) \notin \mathbf{LC}(A_0, B_0)$.

For a *level-1* less conflicting pair $(X_A, X_B) \in \mathbf{LC}^1(A, B)$, if X_B does not contain blocking states, then there must exist a state $x_B \in X_B$ such that $\mathbf{L}^\omega(x_B) \subseteq \mathbf{L}^\omega(X_A)$. This is not the case for every less conflicting pair, as some nonblocking requirements are only implicitly contained in the automaton. To show that (X_A, X_B) is a less conflicting pair, it is enough to find a state in $x_B \in X_B$ that can cover an initial segment of $\mathbf{L}^\omega(X_A)$, as long as a less conflicting pair of a *lower level* is reached afterwards.

Example 7. Consider again automata A_2 and B_2 in Figure 3. By definition, $(\omega, \omega) \in \mathbf{LC}^0(A_2, B_2)$, and following from this, $(\{a_1\}, \{b_0, b_1\}) \in \mathbf{LC}^1(A_2, B_2)$, because the marked language of a_1 is $\alpha^+\omega$, which also is the marked language of b_1.

Now consider the pair $(\{a_0\}, \{b_0, b_1\})$. State a_0 has the marked language $\alpha\alpha^+\omega$, i.e., to avert blocking from a_0, a test must be able to execute at least one of the traces in $\alpha\alpha^+\omega$. Although this language is not directly associated with any state in B_2, the nonblocking requirement is implicitly present in state b_1. If blocking is to be averted from state b_1, event α must be possible. After executing α, state b_0 is entered, from where it is always possible to silently return to state b_1 with marked language $\alpha^+\omega$. Therefore, in order to avert blocking from state b_1, it is necessary to execute α and afterwards be able to terminate using one of the traces in $\alpha^+\omega$. This amounts to the implicit nonblocking requirement to execute a trace from $\alpha\alpha^+\omega$ in state b_1.

Therefore $(\{a_0\}, \{b_0, b_1\}) \notin \mathbf{LC}^1(A_2, B_2)$, but $(\{a_0\}, \{b_0, b_1\}) \in \mathbf{LC}^2(A_2, B_2)$ according to (9): every trace that leads to a terminal state from state b_1 has the prefix α, and $\delta^{\text{det}}_{A_2,B_2}(\{a_0\}, \{b_0, b_1\}, \alpha) = (\{a_1\}, \{b_0, b_1\}) \in \mathbf{LC}^1(A_2, B_2)$.

As shown in the example, some nonblocking requirements have to be constructed using a saturation operation that combines two previously found nonblocking requirements. The level n of a less conflicting pair $(X_A, X_B) \in \mathbf{LC}^n(A, B)$ represents the nesting depth of applications of this saturation operation.

The following two lemmas relate the state-based definition of less conflicting pairs to possible tests and thus to the conflict preorder. A pair (X_A, X_B) is a less conflicting pair, if every test T such that $\mathbf{L}^\omega(X_A) \cap \mathbf{L}^\omega(T) = \emptyset$ also is conflicting with X_B.

Lemma 1. Let $A = \langle \Sigma, Q_A, \rightarrow_A, Q_A^\circ \rangle$, $B = \langle \Sigma, Q_B, \rightarrow_B, Q_B^\circ \rangle$, and $T = \langle \Sigma, Q_T, \rightarrow_T, Q_T^\circ \rangle$ be automata, and let $x_T \in Q_T$ be a (possibly unreachable) state. For every less conflicting pair $(X_A, X_B) \in \mathbf{LC}(A, B)$, at least one of the following conditions holds.

(i) $X_A = \omega$, or $X_A \subseteq Q_A$ and there exists $x_A \in X_A$ such that $\mathbf{L}^\omega(x_A, x_T) \neq \emptyset$.

(ii) There exist states $x_B \in X_B$, $y_B \in Q_B$, and $y_T \in Q_T$ such that $(x_B, x_T) \Rightarrow (y_B, y_T)$ and $\mathbf{L}^\omega(y_B, y_T) = \emptyset$.

(Here and in the following, notation $\mathbf{L}^\omega(x_A, x_T)$ is abused to be a shorthand for $\mathbf{L}^\omega((x_A, x_T))$.)

Proof. As (X_A, X_B) is a less conflicting pair, it holds that $(X_A, X_B) \in \mathbf{LC}^n(A, B)$ for some $n \in \mathbb{N}_0$. The claim is shown by induction on n.

If $(X_A, X_B) \in \mathbf{LC}^0(A, B)$ then by (8) it holds that $X_A = \omega$, or $X_B \subseteq Q_B$ and there exists $x_B \in X_B$ such that $\mathbf{L}^\omega(x_B) = \emptyset$. In the first case (i) holds, and in the second case (ii) holds as $(x_B, x_T) \xrightarrow{\varepsilon} (x_B, x_T)$ and $\mathbf{L}^\omega(x_B, x_T) = \mathbf{L}^\omega(x_B) \cap \mathbf{L}^\omega(x_T) = \emptyset$.

Now assume the claim holds for all $i \leq n$, i.e., for all $(X_A, X_B) \in \mathbf{LC}^i(A, B)$, one of the conditions (i) or (ii) holds, and consider $(X_A, X_B) \in \mathbf{LC}^{n+1}(A, B)$. By (9), there exists $x_B \in X_B$ such that for all $t \in \Sigma^*$, if $x_B \overset{t\omega}{\Rightarrow}$ then there exists a prefix $r \sqsubseteq t\omega$ such that $\delta^{\det}_{A,B}(X_A, X_B, r) \in \mathbf{LC}^i(A, B)$ for some $i \leq n$. If $\mathbf{L}^\omega(x_B, x_T) = \emptyset$, (ii) follows immediately as $(x_B, x_T) \xrightarrow{\varepsilon} (x_B, x_T)$. Therefore assume that $\mathbf{L}^\omega(x_B, x_T) \neq \emptyset$, i.e., there exists $t \in \Sigma^*$ such that $(x_B, x_T) \overset{t\omega}{\Rightarrow}$. Then $x_B \overset{t\omega}{\Rightarrow}$, so there exists $r \sqsubseteq t\omega$ such that $\delta^{\det}_{A,B}(X_A, X_B, r) \in \mathbf{LC}^i(A, B)$ for some $i \leq n$. As $r \sqsubseteq t\omega$ and $x_T \overset{t\omega}{\Rightarrow}$, it also holds that $x_T \overset{r}{\Rightarrow} y_T$ for some $y_T \in Q_T$. Let $\delta^{\det}_{A,B}(X_A, X_B, r) = (Y_A, Y_B)$. By inductive assumption, (i) or (ii) holds for $(Y_A, Y_B) \in \mathbf{LC}^i(A, B)$ and y_T.

(i) In this case, either $Y_A = \omega$, or $Y_A \subseteq Q_A$ and there exists $y_A \in Y_A$ and $u \in \Sigma^*$ such that $(y_A, y_T) \overset{u\omega}{\Rightarrow}$. If $Y_A = \omega$, then $\delta^{\det}_A(X_A, r) = Y_A = \omega$ and according to Definition 4 there exists $r_A \in \Sigma^*$ such that $r = r_A\omega$, and there exist states $x_A \in X_A$ and $y_A \in Q_A$ such that $x_A \overset{r_A}{\Rightarrow} y_A \overset{\omega}{\Rightarrow}$, i.e., $(x_A, x_T) \overset{r_A\omega}{\Longrightarrow}$. If there exists $y_A \in Y_A$ and $u \in \Sigma^*$ such that $(y_A, y_T) \overset{u\omega}{\Rightarrow}$, then since $\delta^{\det}_A(X_A, r) = Y_A$, there exists $x_A \in X_A$ such that $x_A \overset{r}{\Rightarrow} y_A$, i.e., $(x_A, x_T) \overset{r}{\Rightarrow} (y_A, y_T) \overset{u\omega}{\Rightarrow}$. In both cases, (i) holds for (X_A, X_B) and x_T.

(ii) If there exists a state $y_B \in Y_B$ such that $(y_B, y_T) \Rightarrow (z_B, z_T)$ where $\mathbf{L}^\omega(z_B, z_T) = \emptyset$, then since $\delta^{\det}_B(X_B, r) = Y_B$, there exists $x_B \in X_B$ such that $x_B \overset{r}{\Rightarrow} y_B$, which implies $(x_B, x_T) \overset{r}{\Rightarrow} (y_B, y_T) \Rightarrow (z_B, z_T)$ with $\mathbf{L}^\omega(z_B, z_T) = \emptyset$. Thus, (ii) holds for (X_A, X_B) and x_T. □

Conversely, if a pair of state sets is *not* a less conflicting pair for A and B, then this pair gives rise to a test automaton to show that A is not less conflicting than B. This test exhibits blocking behaviour in combination with A but not with B.

Lemma 2. Let $A = \langle \Sigma, Q_A, \rightarrow_A, Q_A^\circ \rangle$ and $B = \langle \Sigma, Q_B, \rightarrow_B, Q_B^\circ \rangle$ be automata. For every pair $\mathbf{X} = (X_A, X_B) \notin \mathbf{LC}(A, B)$, there exists a deterministic automaton $T_{\mathbf{X}} = \langle \Sigma, Q_T, \rightarrow_T, \{x_T^\circ\} \rangle$ such that both the following conditions hold.

(i) For all states $x_A \in X_A$, it holds that $\mathbf{L}^\omega(x_A, x_T^\circ) = \emptyset$.

(ii) For all states $x_B \in X_B$, $y_B \in Q_B$, $y_T \in Q_T$ such that $(x_B, x_T^\circ) \Rightarrow (y_B, y_T)$, it holds that $\mathbf{L}^\omega(y_B, y_T) \neq \emptyset$.

Proof. Construct the deterministic automaton $T_{\mathbf{X}} = \langle \Sigma, Q_T, \rightarrow_T, \{x_T^\circ\} \rangle$ such that

$$\mathbf{L}(T_{\mathbf{X}}) = \{ s \in \Sigma^* \cup \Sigma^*\omega \mid \delta^{\det}_{A,B}(\mathbf{X}, r) \notin \mathbf{LC}(A, B) \text{ for all } r \sqsubseteq s \} . \tag{11}$$

This language is prefix-closed by construction and nonempty because $\mathbf{X} \notin \mathbf{LC}(A, B)$. Therefore, $T_{\mathbf{X}}$ is a well-defined automaton.

(i) Let $x_A \in X_A$. If $x_A \overset{t\omega}{\Rightarrow}$ for some $t \in \Sigma^*$, then $\delta^{\det}_{A,B}(\mathbf{X}, t\omega) = (\omega, Y_B) \in \mathbf{LC}^0(A, B) \subseteq \mathbf{LC}(A, B)$ for some $Y_B \in Q_B^{\det}$ by Definition 4 and 5. It follows from (11) that $t\omega \notin \mathbf{L}(T_{\mathbf{X}})$, and thus $(x_A, x_T^\circ) \overset{t\omega}{\Rightarrow}$ does not hold. Since $t \in \Sigma^*$ was chosen arbitrarily, it follows that $\mathbf{L}^\omega(x_A, x_T^\circ) = \emptyset$.

(ii) Let $x_B \in X_B$, $y_B \in Q_B$, $y_T \in Q_T$, and $s \in \Sigma^*$ such that $(x_B, x_T^\circ) \stackrel{s}{\Rightarrow} (y_B, y_T)$. Clearly $s \in \mathbf{L}(T_\mathbf{X})$, and by (11) it follows that $\delta^{\det}_{A,B}(\mathbf{X}, r) \notin \mathbf{LC}(A,B)$ for all prefixes $r \sqsubseteq s$. Let $\delta^{\det}_{A,B}(\mathbf{X}, s) = \mathbf{Y}$. Then $\mathbf{Y} \notin \mathbf{LC}(A,B)$, so there exists a trace $t \in \Sigma^*$ such that $y_B \stackrel{t\omega}{\Rightarrow}$ and for all $r \sqsubseteq t$ it holds that $\delta^{\det}_{A,B}(\mathbf{Y}, r) \notin \mathbf{LC}(A,B)$ (see Remark 1). Thus $x_B \stackrel{s}{\Rightarrow} y_B \stackrel{t\omega}{\Rightarrow}$ and for all prefixes $u \sqsubseteq st\omega$, it holds that $\delta^{\det}_{A,B}(\mathbf{X}, u) \notin \mathbf{LC}(A,B)$. Then $st\omega \in \mathbf{L}(T_\mathbf{X})$ according to (11), and since $T_\mathbf{X}$ is deterministic, it follows that $y_T \stackrel{t\omega}{\Rightarrow}$. Therefore, $(y_B, y_T) \stackrel{t\omega}{\Rightarrow}$, i.e., $\mathbf{L}^\omega(y_B, y_T) \neq \emptyset$. □

3.3 Less Conflicting Pairs and Certain Conflicts

Less conflicting pairs can be used to characterise the set of *certain conflicts* of an automaton as defined in 2.3. This shows the close link between the conflict preorder and the set of certain conflicts. If a pair $(\emptyset, X_B)$ is a less conflicting pair then, since termination is impossible from $\emptyset$, conflict must be also present in X_B. In this case, every trace leading to X_B must be a trace of certain conflicts. This observation leads to the following alternative characterisation of the set of certain conflicts.

Theorem 3. The set of certain conflicts of $B = \langle \Sigma, Q, \to, Q^\circ \rangle$ can also be written as

$$\textsc{Conf}(B) = \{\, s \in \Sigma^* \mid (\emptyset, \delta^{\det}_B(Q^\circ, r)) \in \mathbf{LC}(O,B) \text{ for some prefix } r \sqsubseteq s \,\}\,, \tag{12}$$

where $O = \langle \Sigma, \emptyset, \emptyset, \emptyset \rangle$ stands for the empty automaton.

Proof. First let $s \in \Sigma^*$ such that $(\emptyset, \delta^{\det}_B(Q^\circ, r)) \in \mathbf{LC}(O,B)$ for some $r \sqsubseteq s$, and let $T = \langle \Sigma, Q_T, \to_T, Q_T^\circ \rangle$ be an automaton such that $T \stackrel{s}{\Rightarrow}$. It is to be shown that $B \parallel T$ is blocking. Since $T \stackrel{s}{\Rightarrow}$ and $r \sqsubseteq s$, it holds that $T \stackrel{r}{\Rightarrow} x_T$ for some state $x_T \in Q_T$. Since $(\emptyset, \delta^{\det}_B(Q^\circ, r)) \in \mathbf{LC}(O,B)$, either (i) or (ii) in Lemma 1 holds. However, (i) is impossible as the first state set of the pair is empty, so (ii) must be true. Thus, there exists a state $x \in \delta^{\det}_B(Q^\circ, r)$ such that $(x, x_T) \Rightarrow (y, y_T)$ where $\mathbf{L}^\omega(y, y_T) = \emptyset$. Then $B \parallel T$ is blocking as $B \parallel T \stackrel{r}{\Rightarrow} (x, x_T) \Rightarrow (y, y_T)$.

Conversely, let $s \in \Sigma^*$ such that $(\emptyset, \delta^{\det}_B(Q^\circ, r)) \notin \mathbf{LC}(O,B)$ for every prefix $r \sqsubseteq s$. It is to be shown that $s \in \mathrm{N}\textsc{Conf}(B)$. Consider the deterministic automaton T such that

$$\mathbf{L}(T) = \{\, t \in \Sigma^* \mid (\emptyset, \delta^{\det}_B(Q^\circ, r)) \notin \mathbf{LC}(O,B) \text{ for all } r \sqsubseteq t \,\}\,. \tag{13}$$

T is a well-defined automaton as $\mathbf{L}(T)$ is prefix-closed by construction. It remains to be shown that $B \parallel T$ is nonblocking. Let $B \parallel T \stackrel{t}{\Rightarrow} (x, x_T)$. Then $t \in \mathbf{L}(T)$, and by definition of T (13), it holds that $(\emptyset, \delta^{\det}_B(Q^\circ, t)) \notin \mathbf{LC}(O,B)$, and the same holds for all prefixes of t. Also $x \in \delta^{\det}_B(Q^\circ, t)$, so there exists a trace $u \in \Sigma^*$ such that $x \stackrel{u\omega}{\Rightarrow}$, and for every prefix $r \sqsubseteq u\omega$, it holds that $\delta^{\det}_{O,B}(\emptyset, \delta^{\det}_B(Q^\circ, t), r) \notin \mathbf{LC}(O,B)$ (see Remark 1). By definition (13), it follows that $tu\omega \in \mathbf{L}(T)$, and since T is deterministic also $x_T \stackrel{u\omega}{\Rightarrow}$. Therefore, $B \parallel T \stackrel{t}{\Rightarrow} (x, x_T) \stackrel{u\omega}{\Rightarrow}$, i.e., $B \parallel T$ is nonblocking. □

The result of Theorem 3 shows how less conflicting pairs generalise certain conflicts for the case when two automata are compared, and in combination with the algorithm in Section 4, less conflicting pairs lead to an alternative presentation of the algorithm [10] to compute the set of certain conflicts.

3.4 Testing the Conflict Preorder

Given the less conflicting pairs for two automata A and B, it is possible to determine whether $A \lesssim_{\text{conf}} B$. Automaton A is less conflicting than B if every test T that is nonconflicting in combination with B also is

nonconflicting with A. To check this condition, it is enough to consider traces $B \parallel T \stackrel{s}{\Rightarrow} (x_B, x_T)$, and check whether termination is also possible for every state x_A of A such that $A \parallel T \stackrel{s}{\Rightarrow} (x_A, x_T)$. This amounts to checking whether $(\{x_A\}, X_B) \in \mathbf{LC}(A,B)$ when $A \stackrel{s}{\Rightarrow} x_A$ and $\delta_B^{\det}(Q_B^\circ, s) = X_B$.

However, this condition does not apply to traces of certain conflicts. If $s \in \mathrm{CONF}(B)$, then every test T that can execute s is in conflict with B. In this case, A can still be less conflicting than B, no matter whether A can or cannot execute the trace s and terminate afterwards. This observation leads to the following result.

Theorem 4. Let $A = \langle \Sigma, Q_A, \rightarrow_A, Q_A^\circ \rangle$ and $B = \langle \Sigma, Q_B, \rightarrow_B, Q_B^\circ \rangle$ be two automata. A is less conflicting than B if and only if for all $s \in \mathrm{NCONF}(B)$ and all $x_A \in Q_A$ such that $A \stackrel{s}{\Rightarrow} x_A$ it holds that $(\{x_A\}, X_B) \in \mathbf{LC}(A,B)$, where $\delta_B^{\det}(Q_B^\circ, s) = X_B$.

Proof. First assume that for all $s \in \mathrm{NCONF}(B)$ and all $x_A \in Q_A$ such that $A \stackrel{s}{\Rightarrow} x_A$ it holds that $(\{x_A\}, X_B) \in \mathbf{LC}(A,B)$, where $\delta_B^{\det}(Q_B^\circ, s) = X_B$. Let $T = \langle \Sigma, Q_T, \rightarrow_T, Q_T^\circ \rangle$ such that $B \parallel T$ is nonblocking, and assume that $A \parallel T \stackrel{s}{\Rightarrow} (x_A, x_T)$. Since $B \parallel T$ is nonblocking and $T \stackrel{s}{\Rightarrow}$, it follows that $s \in \mathrm{NCONF}(B)$. Therefore by assumption $(\{x_A\}, X_B) \in \mathbf{LC}(A,B)$, so (i) or (ii) in Lemma 1 must be true. However, (ii) cannot hold, because for all $x_B \in X_B = \delta_B^{\det}(Q^\circ, s)$ it holds that $B \parallel T \stackrel{s}{\Rightarrow} (x_B, x_T)$, and since $B \parallel T$ is nonblocking, there cannot exist any state (y_B, y_T) such that $(x_B, x_T) \Rightarrow (y_B, y_T)$ and $\mathbf{L}^\omega(y_B, y_T) = \emptyset$. Thus, (i) must be true, and this means that $\mathbf{L}^\omega(x_A, x_T) \neq \emptyset$. Since T and s such that $A \parallel T \stackrel{s}{\Rightarrow} (x_A, x_T)$ were chosen arbitrarily, it follows that $A \lesssim_{\mathrm{conf}} B$.

Second assume that there exists $s \in \mathrm{NCONF}(B)$ and $x_A \in Q_A$ such that $A \stackrel{s}{\Rightarrow} x_A$ and $\mathbf{X} = (\{x_A\}, X_B) \notin \mathbf{LC}(A,B)$, where $X_B = \delta_B^{\det}(Q_B^\circ, s)$. Let $N_B = \langle \Sigma, Q_N, \rightarrow_N, \{x_N^\circ\} \rangle$ be a deterministic recogniser of the language $\mathrm{NCONF}(B)$, and let $T_{\mathbf{X}} = \langle \Sigma, Q_T, \rightarrow_T, \{x_T^\circ\} \rangle$ be the deterministic automaton that exists according to Lemma 2. Since $s \in \mathrm{NCONF}(B)$, there exists a unique state $x_s \in Q_N$ such that $N_B \stackrel{s}{\rightarrow} x_s$. Then construct the automaton

$$T = \langle \Sigma, Q_N \dot\cup Q_T, \rightarrow_N \cup \rightarrow_T \cup \{(x_s, \tau, x_T^\circ)\}, \{x_N^\circ\} \rangle \,. \tag{14}$$

Clearly, $A \parallel T \stackrel{s}{\Rightarrow} (x_A, x_s) \stackrel{\tau}{\rightarrow} (x_A, x_T^\circ)$, and $\mathbf{L}^\omega(x_A, x_T^\circ) = \emptyset$ by Lemma 2 (i). Thus, $A \parallel T$ is blocking.

On the other hand, $B \parallel T$ is nonblocking. To see this, consider $B \parallel T \stackrel{t}{\Rightarrow} (y_B, y_T)$. If $y_T \in Q_N$, then it follows from the fact that $B \parallel N_B$ is nonblocking [11] that there exists $u \in \Sigma^*$ such that $(y_B, y_T) \stackrel{u\omega}{\Rightarrow}$. Otherwise $y_T \in Q_T$, which means that $t = su$ and $T \stackrel{s}{\rightarrow} x_s \stackrel{\tau}{\rightarrow} x_T^\circ \stackrel{u}{\rightarrow} y_T$. Also since $B \stackrel{t}{\Rightarrow} y_B$, it follows that $y_B \in \delta_B^{\det}(Q_B^\circ, t) = \delta^{\det}(Q_B^\circ, su) = \delta_B^{\det}(\delta_B^{\det}(Q_B^\circ, s), u) = \delta_B^{\det}(X_B, u)$, i.e., there exists $x_B \in X_B$ such that $x_B \stackrel{u}{\Rightarrow} y_B$. Thus $(x_B, x_T^\circ) \stackrel{u}{\Rightarrow} (y_B, y_T)$, and by Lemma 2 (ii), it holds that $\mathbf{L}^\omega(y_B, y_T) \neq \emptyset$.

Thus, $A \parallel T$ is blocking and $B \parallel T$ is nonblocking, so $A \lesssim_{\mathrm{conf}} B$ cannot hold. □

Example 8. Consider again automata A_0 and B_0 in Figure 1. Recall that $\mathrm{CONF}(B_0) = \alpha\Sigma^*$ from Example 3, so the only state in A_0 that can be reached by a trace $s \notin \mathrm{CONF}(B_0)$ is a_0. Therefore, it is enough to check the pair $(\{a_0\}, \{b_0\})$ according to Theorem 4, and it has been shown in Example 6 that $(\{a_0\}, \{b_0\}) \in \mathbf{LC}^1(A_0, B_0)$. It follows that $A_0 \lesssim_{\mathrm{conf}} B_0$. This conclusion is made despite the fact that $(\{a_0\}, \{b_2\}) \notin \mathbf{LC}(A_0, B_0)$, because $(\{a_0\}, \{b_2\})$ is only reachable by traces $\alpha^n \in \mathrm{CONF}(B_0)$, $n \geq 2$.

When using Theorem 4 to determine whether an automaton A is less conflicting than some blocking automaton B, the set of certain conflicts of B must be known first. This can be achieved using Theorem 3, which makes it possible to classify state sets in the subset construction of B as certain conflicts. If a state set $X_B \subseteq Q_B$ is found to represent certain conflicts, i.e., $(\emptyset, X_B) \in \mathbf{LC}(O,B)$ according to Theorem 3, then $(X_A, X_B) \in \mathbf{LC}(A,B)$ for every state set $X_A \subseteq Q_A$. Successors reached only from such pairs are also certain conflicts of B and should not be considered when testing whether $A \lesssim_{\mathrm{conf}} B$ according to Theorem 4.

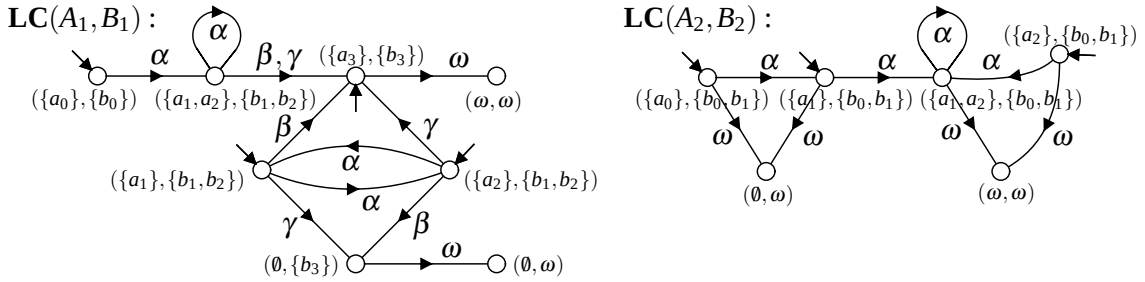


Figure 4: Less conflicting pairs for the automata pairs in Figure 2 and 3.

Example 9. Consider again automata A_1 and B_1 in Figure 2. Composing A_1 with a deterministic version of B_1 results in the following four pairs of states in A_1 and sets of states in B_1 that should be tested according to Theorem 4 to determine whether $A_1 \lesssim_{\text{conf}} B_1$:

$$(\{a_0\},\{b_0\}) \quad (\{a_1\},\{b_1,b_2\}) \quad (\{a_2\},\{b_1,b_2\}) \quad (\{a_3\},\{b_3\})\,. \tag{15}$$

All four pairs need to be considered as B_1 is nonblocking and thus $\text{CONF}(B_1) = \emptyset$.

The graph to the left in Figure 4 shows these four pairs and their deterministic successors. The four pairs (15) are marked as initial states, and the arrows in the graph represent the deterministic transition function. Although the deterministic transition function is defined for all state set pairs and events, arrows to $(\emptyset,\emptyset)$ are suppressed for clarity of presentation.

The following less conflicting pairs to compare A_1 to B_1 are determined from the graph:

$$(\omega,\omega) \in \mathbf{LC}^0(A_1,B_1)\,; \tag{16}$$

$$(\{a_0\},\{b_0\}),\ (\{a_1,a_2\},\{b_1,b_2\}),\ (\{a_3\},\{b_3\}) \in \mathbf{LC}^1(A_1,B_1)\,. \tag{17}$$

For example, $(\{a_1,a_2\},\{b_1,b_2\}) \in \mathbf{LC}^1(A_1,B_1)$, because all the ways to reach termination from state b_1, i.e., all traces in $\mathbf{L}^\omega(b_1) = \alpha^*\beta\omega$ take the pair $(\{a_1,a_2\},\{b_1,b_2\})$ to $(\omega,\omega) \in LC^0(A_1,B_1)$. No further pairs are found in $\mathbf{LC}^2(A_1,B_1)$, so $\mathbf{LC}(A_1,B_1)$ consists only of the pairs listed above. For example, $(\{a_1\},\{b_1,b_2\}) \notin \mathbf{LC}^2(A_1,B_1)$, because the traces $\alpha\beta\omega \in \mathbf{L}^\omega(b_1)$ and $\gamma\omega \in \mathbf{L}^\omega(b_2)$ do not have any prefixes that reach a pair in $\mathbf{LC}^1(A_1,B_1)$.

As $(\{a_1\},\{b_1,b_2\}) \notin \mathbf{LC}(A_1,B_1)$, it follows from Theorem 4 that A_1 is *not* less conflicting than B_1.

Example 10. Consider again automata A_2 and B_2 in Figure 3. Again note that $\text{CONF}(B_2) = \emptyset$. By composing A_2 with a deterministic version of B_2, it becomes clear that the only pairs that need to be tested to determine whether $A_2 \lesssim_{\text{conf}} B_2$ according to Theorem 4 are $(\{a_0\},\{b_0,b_1\})$ reached after ε, $(\{a_1\},\{b_0,b_1\})$ reached after α^+, and $(\{a_2\},\{b_0,b_1\})$ reached after $\alpha\alpha^+$.

The graph with these pairs and their deterministic successors is shown to the right in Figure 4, with the three crucial pairs marked as initial. The following less conflicting pairs are discovered (see Example 7):

$$(\omega,\omega) \in \mathbf{LC}^0(A_2,B_2)\,; \tag{18}$$

$$(\{a_1\},\ \{b_0,b_1\}),\ (\{a_1,a_2\},\{b_0,b_1\}),\ (\{a_2\},\{b_0,b_1\}) \in \mathbf{LC}^1(A_2,B_2)\,; \tag{19}$$

$$(\{a_0\},\{b_0,b_1\}) \in \mathbf{LC}^2(A_2,B_2)\,. \tag{20}$$

As the three crucial pairs are all in $\mathbf{LC}(A_2,B_2)$, it follows from Theorem 4 that $A_2 \lesssim_{\text{conf}} B_2$.

The result of Theorem 4 is related to the decision procedure for fair testing [15]. The fair testing decision procedure starts by composing the automaton A with a determinised form of B, which gives rise to the same state set combinations that need to be considered as in Theorem 4. From this point on, the two methods differ. The fair testing decision procedure annotates each state of the synchronous product of A and the determinised form of B with automata representing the associated refusal trees, and searches for matching automata (or more precisely, for matching *productive subautomata*) within these annotations. The method based on less conflicting pairs avoids some of the resulting complexity by performing the complete decision on the flat state space of the synchronous product of the determinised forms of A and B.

4 Algorithm to Compute Less Conflicting Pairs

This section proposes a method to effectively compute the less conflicting pairs for two given finite-state automata A and B. This is done in a nested iteration. Assuming that the set $\mathbf{LC}^n(A,B)$ is already known, the set $\mathbf{LC}^{n+1}(A,B)$ is computed in a secondary iteration based on *more conflicting triples*.

Definition 6. Let $A = \langle \Sigma, Q_A, \to_A, Q_A^\circ \rangle$ and $B = \langle \Sigma, Q_B, \to_B, Q_B^\circ \rangle$ be automata. The set $\mathbf{MC}^n(A,B) \subseteq Q_A^{\det} \times Q_B^{\det} \times Q_B$ of n^{th} level *more conflicting triples* for A and B is defined inductively as follows.

$$\mathbf{MC}_0^n(A,B) = \{\, (\emptyset, \omega, x_B) \mid x_B \in Q_B \,\}\,; \tag{21}$$

$$\mathbf{MC}_{m+1}^n(A,B) = \{\, (X_A, X_B, x_B) \mid (X_A,X_B) \notin \mathbf{LC}^n(A,B) \text{ and } x_B \in X_B \text{ and there exists } (Y_A,Y_B,y_B) \in \mathbf{MC}_m^n(A,B) \text{ and } \sigma \in \Sigma \text{ such that } \delta_{A,B}^{\det}(X_A,X_B,\sigma) = (Y_A,Y_B) \text{ and } x_B \stackrel{\sigma}{\Rightarrow} y_B \,\}\,; \tag{22}$$

$$\mathbf{MC}^n(A,B) = \bigcup_{m \geq 0} \mathbf{MC}_m^n(A,B)\,. \tag{23}$$

For a pair (X_A,X_B) to be a less conflicting pair, according to Definition 5 there must be a state $x_B \in X_B$ such that every trace that takes x_B to termination in B has a prefix that leads to another less conflicting pair. A triple (X_A,X_B,x_B) is considered "more conflicting" if (X_A,X_B) is not yet known to be a less conflicting pair, and the state $x_B \in X_B$ cannot be used to confirm the above property. Therefore, Lemma 5 shows that a triple (X_A,X_B,x_B) is n^{th}-level "more conflicting" if and only if the state $x_B \in X_B$ can reach termination without passing through a pair in $\mathbf{LC}^n$.

If (X_A,X_B,x_B) is "more conflicting" for all $x_B \in X_B$, then the pair (X_A,X_B) cannot be a less conflicting pair. Otherwise, if there exists at least one state $x_B \in X_B$ such that (X_A,X_B,x_B) is not "more conflicting", then (X_A,X_B) is added to set of less conflicting pairs in the next iteration. Theorem 6 below confirms the correctness of this approach.

Lemma 5. Let $A = \langle \Sigma, Q_A, \to_A, Q_A^\circ \rangle$ and $B = \langle \Sigma, Q_B, \to_B, Q_B^\circ \rangle$ be automata, let $n \in \mathbb{N}_0$ and $(X_A,X_B,x_B) \in Q_A^{\det} \times Q_B^{\det} \times Q_B$. The following statements are equivalent.

(i) $(X_A,X_B,x_B) \in \mathbf{MC}^n(A,B)$;

(ii) There exists a trace $s \in \Sigma^*\omega \cup \{\varepsilon\}$ such that $\delta_{A,B}^{\det}(X_A,X_B,s) = (\emptyset,\omega)$ and $x_B \stackrel{s}{\Rightarrow}$, and $\delta_{A,B}^{\det}(X_A,X_B,r) \notin \mathbf{LC}^n(A,B)$ for all prefixes $r \sqsubseteq s$.

Proof. First let $(X_A,X_B,x_B) \in \mathbf{MC}^n(A,B)$, i.e., $(X_A,X_B,x_B) \in \mathbf{MC}_m^n(A,B)$ for some $m \in \mathbb{N}_0$. It is shown by induction on m that (ii) holds.

In the base case, $m = 0$, and by definition $(X_A,X_B,x_B) \in \mathbf{MC}_0^n(A,B)$ means that $(X_A,X_B) = (\emptyset,\omega)$. Then consider $s = \varepsilon$, and note $\delta_{A,B}^{\det}(X_A,X_B,\varepsilon) = (X_A,X_B) = (\emptyset,\omega)$ and $x_B \stackrel{\varepsilon}{\Rightarrow}$. Clearly $r \sqsubseteq \varepsilon$ implies $r = \varepsilon$, and $\delta_{A,B}^{\det}(X_A,X_B,\varepsilon) = (\emptyset,\omega) \notin \mathbf{LC}(A,B) \supseteq \mathbf{LC}^n(A,B)$ by Lemma 1.

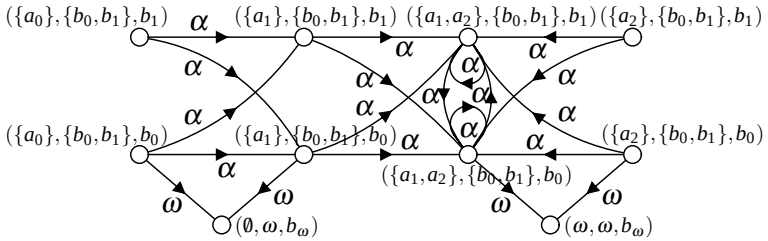


Figure 5: Calculating more conflicting triples for automata A_2 and B_2 in Figure 3.

Now consider $(X_A,X_B,x_B) \in \mathbf{MC}^n_{m+1}(A,B)$. It follows from Definition 6 that $(X_A,X_B) \notin LC^n(A,B)$ and $x_B \in X_B$, and there exists $(Y_A,Y_B,y_B) \in \mathbf{MC}^n_m(A,B)$ and $\sigma \in \Sigma$ such that $\delta^{\det}_{A,B}(X_A,X_B,\sigma) = (Y_A,Y_B)$ and $x_B \stackrel{\sigma}{\Rightarrow} y_B$. By inductive assumption, there exists a trace $s \in \Sigma^*\omega \cup \{\varepsilon\}$ such that $\delta^{\det}_{A,B}(Y_A,Y_B,s) = (\emptyset,\omega)$ and $y_B \stackrel{s}{\Rightarrow}$, and for all $r \sqsubseteq s$ it holds that $\delta^{\det}_{A,B}(Y_A,Y_B,r) \notin \mathbf{LC}^n(A,B)$. Then $\delta^{\det}_{A,B}(X_A,X_B,\sigma s) = \delta^{\det}_{A,B}(Y_A,Y_B,s) = (\emptyset,\omega)$ and $x_B \stackrel{\sigma}{\Rightarrow} y_B \stackrel{s}{\Rightarrow}$, and for all $r \sqsubseteq \sigma s$ it holds that $\delta^{\det}_{A,B}(X_A,X_B,r) \notin \mathbf{LC}^n(A,B)$.

Conversely, let $s \in \Sigma^*\omega \cup \{\varepsilon\}$ such that (ii) holds. This means that $\delta^{\det}_{A,B}(X_A,X_B,s) = (\emptyset,\omega)$ and $x_B \stackrel{s}{\Rightarrow}$, and $\delta^{\det}_{A,B}(X_A,X_B,r) \notin \mathbf{LC}^n(A,B)$ for all $r \sqsubseteq s$. It is shown by induction on $m = |s|$ that $(X_A,X_B,x_B) \in \mathbf{MC}^n_m(A,B)$.

In the base case, $m = 0$ and $s = \varepsilon$, it holds by definition that $(X_A,X_B) = \delta^{\det}_{A,B}(X_A,X_B,\varepsilon) = (\emptyset,\omega) \in \mathbf{MC}^n_0(A,B)$.

Now let $s = \sigma t$ such that $|t| = m$, and $\delta^{\det}_{A,B}(X_A,X_B,s) = (\emptyset,\omega)$ and $x_B \stackrel{s}{\Rightarrow}$, and $\delta^{\det}_{A,B}(X_A,X_B,r) \notin \mathbf{LC}^n(A,B)$ for all prefixes $r \sqsubseteq s$. Write $\delta^{\det}_{A,B}(X_A,X_B,\sigma) = (Y_A,Y_B)$ and $x_B \stackrel{\sigma}{\Rightarrow} y_B \stackrel{t}{\Rightarrow}$. Then $y_B \stackrel{t}{\Rightarrow}$ and $\delta^{\det}_{A,B}(Y_A,Y_B,t) = \delta^{\det}_{A,B}(X_A,X_B,\sigma t) = \delta^{\det}_{A,B}(X_A,X_B,s) = (\emptyset,\omega)$ and $\delta^{\det}_{A,B}(Y_A,Y_B,r) \notin \mathbf{LC}^n(A,B)$ for all $r \sqsubseteq t$. Then $(Y_A,Y_B,y_B) \in \mathbf{MC}^n_m(A,B)$ by inductive assumption, and by Definition 6 it follows that $(X_A,X_B,x_B) \in \mathbf{MC}^n_{m+1}(A,B)$. □

Theorem 6. Let $A = \langle \Sigma, Q_A, \rightarrow_A, Q^\circ_A \rangle$ and $B = \langle \Sigma, Q_B, \rightarrow_B, Q^\circ_B \rangle$ be automata, and let $n \in \mathbb{N}_0$. Then

$$\mathbf{LC}^{n+1}(A,B) = \{\, (X_A,X_B) \in Q^{\det}_A \times Q^{\det}_B \mid (X_A,X_B,x_B) \notin \mathbf{MC}^n(A,B) \text{ for some } x_B \in X_B \,\}\,. \tag{24}$$

Proof. Let $(X_A,X_B) \in LC^{n+1}(A,B)$. Then by Definition 5, there exists $x_B \in X_B$ such that for all $t \in \Sigma^*$ such that $x_B \stackrel{t\omega}{\Rightarrow}$, there exists $r \sqsubseteq t\omega$ such that $\delta^{\det}_{A,B}(X_A,X_B,r) \in \mathbf{LC}^i(A,B)$ for some $i \leq n$. Equivalently, this means that there does not exist a trace $t \in \Sigma^*$ such that $x_B \stackrel{t\omega}{\Rightarrow}$ and for all prefixes $r \sqsubseteq t\omega$ it holds that $\delta^{\det}_{A,B}(X_A,X_B,r) \notin \mathbf{LC}^n(A,B)$. Then $(X_A,X_B,x_B) \notin \mathbf{MC}^n(A,B)$ because otherwise such a trace would exist by Lemma 5.

Conversely, let $x_B \in X_B$ such that $(X_A,X_B,x_B) \notin \mathbf{MC}^n(A,B)$. To check the condition in Definition 5 (9), consider $t \in \Sigma^*$ such that $x_B \stackrel{t\omega}{\Rightarrow}$. Then clearly $\delta^{\det}_B(X_B,t\omega) = \omega$. By Definition 4, it holds that either $\delta^{\det}_A(X_A,t\omega) = \omega$ or $\delta^{\det}_A(X_A,t\omega) = \emptyset$. If $\delta^{\det}_A(X_A,t\omega) = \omega$, then $\delta^{\det}_{A,B}(X_A,X_B,t\omega) = (\omega,\omega) \in \mathbf{LC}^0(A,B)$. Otherwise $\delta^{\det}_A(X_A,t\omega) = \emptyset$ and thus $\delta^{\det}_{A,B}(X_A,X_B,t\omega) = (\emptyset,\omega)$, and by Lemma 5 there must exist $r \sqsubseteq t\omega$ such that $\delta^{\det}_{A,B}(X_A,X_B,r) \in \mathbf{LC}^n(A,B)$ as otherwise $(X_A,X_B,x_B) \in \mathbf{MC}^n(A,B)$. In both cases, $\delta^{\det}_{A,B}(X_A,X_B,r) \in \mathbf{LC}^i(A,B)$ for some $r \sqsubseteq t\omega$ and $i \leq n$. Since $t \in \Sigma^*$ with $x_B \stackrel{t\omega}{\Rightarrow}$ was chosen arbitrarily, it follows from Definition 5 (9) that $(X_A,X_B) \in \mathbf{LC}^{n+1}(A,B)$. □

Example 11. Figure 5 shows a graph representing the more conflicting triples to check whether $A_2 \lesssim_{\text{conf}} B_2$ in Figure 3. The arrows in the graph represent the deterministic transition function in combination

with the transition relation of B_2. An arrow $(X_A, X_B, x_B) \xrightarrow{\sigma} (Y_A, Y_B, y_B)$ indicates that $\delta^{\det}_{A_2,B_2}(X_A, X_B, \sigma) = (Y_A, Y_B)$ and $x_B \stackrel{\sigma}{\Rightarrow} y_B$.

In the first iteration to compute $\mathbf{MC}^0(A_2,B_2)$, first the triple $(\emptyset, \omega, b_\omega)$ is added to $\mathbf{MC}^0_0(A_2,B_2)$. Next, the triples $(\{a_0\},\{b_0,b_1\},b_0)$ and $(\{a_1\},\{b_0,b_1\},b_0)$ are added to $\mathbf{MC}^0_1(A_2,B_2)$ as they can immediately reach $(\emptyset,\omega,b_\omega)$. Finally, $(\{a_0\},\{b_0,b_1\},b_1)$ is also added to $\mathbf{MC}^0_2(A_2,B_2)$ as it reaches $(\{a_1\},\{b_0,b_1\},b_0) \in \mathbf{MC}^0_1(A_2,B_2)$. No further triples are found to be in $\mathbf{MC}^0_3(A_2,B_2)$. Therefore, $(\{a_1\},\{b_0,b_1\},b_1) \notin \mathbf{MC}^0(A_2,B_2)$, so it follows from Theorem 6 that $(\{a_1\},\{b_0,b_1\}) \in \mathbf{LC}^1(A_2,B_2)$, and likewise $(\{a_1,a_2\},\{b_0,b_1\}), (\{a_2\},\{b_0,b_1\}) \in \mathbf{LC}^1(A_2,B_2)$.

In the next iteration to compute $\mathbf{MC}^1(A_2,B_2)$, note that $(\{a_1\},\{b_0,b_1\},b_0) \notin \mathbf{MC}^1_1(A_2,B_2)$ because $(\{a_1\},\{b_0,b_1\}) \in \mathbf{LC}^1(A_2,B_2)$. Still, $(\{a_0\},\{b_0,b_1\},b_0) \in \mathbf{MC}^1_1(A_2,B_2)$ because of the transition to $(\emptyset,\omega,b_\omega) \in \mathbf{MC}^1_0(A_2,B_2)$, but $(\{a_0\},\{b_0,b_1\},b_1) \notin \mathbf{MC}^1_2(A_2,B_2)$ because now $(\{a_1\},\{b_0,b_1\},b_0) \notin \mathbf{MC}^1_1(A_2,B_2)$. Accordingly, the pair $(\{a_0\},\{b_0,b_1\})$ is added to $\mathbf{LC}^2(A_2,B_2)$.

In a final iteration to compute $\mathbf{MC}^2(A_2,B_2)$, only one more conflicting triple is found, $(\emptyset,\omega,b_\omega) \in \mathbf{MC}^2_0(A_2,B_2)$. No further pairs are added in $\mathbf{LC}^3(A_2,B_2)$. At this point, the iteration terminates, having found exactly the four less conflicting pairs given in Example 10, (19) and (20).

To determine whether an automaton A is less conflicting than automaton B, it is first needed to determine the set of certain conflicts of B, and then to find all the state-set pairs for A and B that are reachable from a pair like $(\{x_A\},X_B)$ associated with some trace that is not a certain conflict of B. The more conflicting triples can be constructed as they are discovered during the backwards search from the terminal states.

The complexity of each iteration of the more conflicting triples computation is determined by the number of arrows in the graph, which is bounded by $|\Sigma| \cdot |Q_B|^2 \cdot 2^{|Q_A|} \cdot 2^{|Q_B|}$, because the powerset transitions are deterministic, which is not the case for the transitions of B. Each iteration except the last adds at least one less conflicting pair, so the number of iterations is bounded by $2^{|Q_A|} \cdot 2^{|Q_B|}$. The complexity of this loop dominates all other tasks of the computation. Therefore, the worst-case time complexity to determine whether $A \lesssim_{\text{conf}} B$ using less conflicting pairs is

$$O(|\Sigma| \cdot |Q_B|^2 \cdot 4^{|Q_A|} \cdot 4^{|Q_B|}) \;=\; O(|\Sigma| \cdot |Q_B|^2 \cdot 2^{2|Q_A|+2|Q_B|}) \,. \tag{25}$$

This shows that the conflict preorder can be tested in linear exponential time, as it is the case for the fair testing preorder. Yet, the complexity (25) is better than the time complexity of the decision procedure for fair testing, which is $O(|Q_A| \cdot |Q_B| \cdot 2^{3|Q_A|+5|Q_B|})$ [15].

5 Conclusions

Less conflicting pairs provide a concrete state-based means to characterise the extent by which one process is or is not less conflicting than another. The characterisation generalises and includes previous results about certain conflicts, and it gives rise to a direct way to test the conflict preorder and the related fair testing preorder by inspecting sets of reachable states. Based on the characterisation, an effective algorithm is presented to test whether a finite-state automaton is less conflicting than another. The algorithm, while still linear exponential, has better time complexity than the previously known decision procedure for fair testing.

In the future, the authors would like to apply the theoretic results of this paper to compute abstractions and improve the performance of compositional model checking algorithms. The more thorough understanding of the conflict preorder will make it possible to better simplify processes with respect to conflict equivalence and other related liveness properties.

References

[1] Christel Baier & Joost-Pieter Katoen (2008): *Principles of Model Checking*. MIT Press.

[2] Ed Brinksma, Arend Rensink & Walter Vogler (1995): *Fair Testing*. In Insup Lee & Scott A. Smolka, editors: *Proc. 6th Int. Conf. Concurrency Theory, CONCUR '95, LNCS* 962, Springer, Philadelphia, PA, USA, pp. 313–327.

[3] C. G. Cassandras & S. Lafortune (1999): *Introduction to Discrete Event Systems*. Kluwer.

[4] R. De Nicola & M. C. B. Hennessy (1984): *Testing Equivalences for Processes*. *Theoretical Comput. Sci.* 34(1–2), pp. 83–133, doi:10.1016/0304-3975(84)90113-0.

[5] Hugo Flordal & Robi Malik (2009): *Compositional Verification in Supervisory Control*. *SIAM J. Control and Optimization* 48(3), pp. 1914–1938, doi:10.1137/070695526.

[6] R. J. van Glabbeek (2001): *The Linear Time — Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes*. In J. A. Bergstra, A. Ponse & S. A. Smolka, editors: *Handbook of Process Algebra*, Elsevier, pp. 3–99.

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

[8] John E. Hopcroft, Rajeev Motwani & Jeffrey D. Ullman (2001): *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

[9] Robi Malik (2004): *On the Set of Certain Conflicts of a Given Language*. In: *Proc. 7th Int. Workshop on Discrete Event Systems, WODES '04*, Reims, France, pp. 277–282.

[10] Robi Malik (2010): *The Language of Certain Conflicts of a Nondeterministic Process*. Working Paper 05/2010, Dept. of Computer Science, University of Waikato, Hamilton, New Zealand.

[11] Robi Malik, David Streader & Steve Reeves (2006): *Conflicts and Fair Testing*. *Int. J. Found. Comput. Sci.* 17(4), pp. 797–813.

[12] Robin Milner (1989): *Communication and concurrency*. Series in Computer Science, Prentice-Hall.

[13] V. Natarajan & Rance Cleaveland (1995): *Divergence and Fair Testing*. In: *Proc. 22nd Int. Colloquium on Automata, Languages, and Programming, ICALP '95*, pp. 648–659.

[14] Peter J. G. Ramadge & W. Murray Wonham (1989): *The Control of Discrete Event Systems*. *Proc. IEEE* 77(1), pp. 81–98.

[15] Arend Rensink & Walter Vogler (2007): *Fair testing*. *Information and Computation* 205(2), pp. 125–198, doi:10.1016/j.ic.2006.06.002.

[16] Rong Su, Jan H. van Schuppen, Jacobus E. Rooda & Albert T. Hofkamp (2010): *Nonconflict check by using sequential automaton abstractions based on weak observation equivalence*. *Automatica* 46(6), pp. 968–978, doi:10.1016/j.automatica.2010.02.025.

[17] Simon Ware & Robi Malik (2010): *Compositional Nonblocking Verification Using Annotated Automata*. In: *Proc. 10th Int. Workshop on Discrete Event Systems, WODES '10*, Berlin, Germany, pp. 374–379.

[18] K. C. Wong, J. G. Thistle, R. P. Malhame & H.-H. Hoang (2000): *Supervisory Control of Distributed Systems: Conflict Resolution*. *Discrete Event Dyn. Syst.* 10, pp. 131–186.

Predicting global usages of resources endowed with local policies *

Chiara Bodei, Viet Dung Dinh and Gian Luigi Ferrari
Dipartimento di Informatica, Università di Pisa, Italy
{chiara,dinh,giangi}@di.unipi.it

The effective usages of computational resources are a primary concern of up-to-date distributed applications. In this paper, we present a methodology to reason about resource usages (acquisition, release, revision, ...), and therefore the proposed approach enables to predict *bad* usages of resources. Keeping in mind the interplay between local and global information occurring in the application-resource interactions, we model resources as entities with local policies and global properties governing the overall interactions. Formally, our model takes the shape of an extension of π-calculus with primitives to manage resources. We develop a Control Flow Analysis computing a static approximation of process behaviour and therefore of the resource usages.

1 Introduction

Evolutionary programming paradigms for distributed systems changed the way computational resources are integrated into applications. Resources are usually geographically distributed and have their own states, costs and access mechanisms. Moreover, resources are not created nor destroyed by applications, but directly acquired on-the-fly when needed from suitable resource rental services. Clearly, resource acquisition is subject to availability and requires the agreement between client requirements and service guarantees (Service Level Agreement – SLA). The dynamic acquisition of resources increases the complexity of software since the capability of adapting behaviour strictly depends on resource availability. *Ubiquitous computing* [1] and *Cloud computing* [8, 16, 2] provide illustrative examples of a new generation of applications where resource awareness has been a major concern.

The design of suitable mechanisms to control the distributed acquisition and ownership of computational resources is therefore a great challenge. Understanding the foundations of the distributed management of resources could support state-of-the-art advances of programming language constructs, algorithms and reasoning techniques for resource-aware programming. In the last few years, the problem of providing the mathematical basis for the mechanisms that support resource acquisition and usage has been tackled by several authors (see e.g. [3, 7, 13, 11, 15], to cite only a few).

Here we consider a programming model where processes and resources are distinguished entities. Resources are computational entities having their own life-cycle. Resources can range from computational infrastructures, storage and data services to special-purpose devices. Processes dynamically acquire the required resources when available, but they cannot create any resource. This simple programming model abstracts the features of several interesting distributed applications. As an example, let us consider a cloud system offering computing resources. The available resources are the CPU units of a given power and processes can only acquire the CPU time, when available, to run some specialised code. Similar considerations apply to storage services, where client processes can only acquire slots of the

*Research supported by the Italian PRIN Project “SOFT”, FET Project “ASCENS” and Autonomous Region of Sardinia Project “TESLA”.

M.R. Mousavi and A. Ravara: FOCLASA 2011
EPTCS 58, 2011, pp. 49–64, doi:10.4204/EPTCS.58.4

available storage. In our programming model, the deployed resources can be dynamically *reconfigured* to deal with resource upgrade, resource un-availability, security intrusion and failures. A distinguished feature of our approach is that the reconfiguration steps updating the structure of the available resources are not under the control of client processes.

In this paper, we introduce the formal basis of our programming model. Specifically, we introduce a process calculus with explicit primitives for the distributed ownerships of resources. In our calculus, resources are not statically granted to processes, but they are dynamically acquired on-the-fly when they are needed.

We start from the π-calculus [14] and we extend it with primitives to represent resources and the operations to acquire and release resources on demand. Central to our approach is the identification of an abstract notion of resource. In our model, resources are *stateful* entities available in the network environment where processes live. Specifically, a resource is described through the declaration of its interaction endpoint (the resource name), its *local* state and its *global* properties. Global properties establish and enforce the SLA to be satisfied by any interaction the resource engages with its client process. The global interaction properties can be expressed by means of a suitable resource-aware logic in the style of [3], or contract-based logic as in [10, 4]. The interplay between local and global information occurring in the process-resource interactions motivates the adjective *G-Local* given to our extension of the π-calculus.

Since we build over the π-calculus, name-passing is the basic communication mechanism among processes. Beyond exchanging channel names, processes can pass resource names as well. Resource acquisition is instead based on a different abstraction. In order to acquire the ownership of a certain resource, a process issues a suitable request. Such request is routed in the network environment to the resource. The resource is granted only if it is available. In other words the process-resource interaction paradigm adheres to the *publish-subscribe* model: resources act as publishers while processes act as subscribers. Notice that processes issue their requests without being aware of the availability of the resources. When they have completed their task on the acquired resource they release it and make it available for new requests. The two-stage nature of the publish-subscribe paradigm relaxes the inter-dependencies among computational components thus achieving a high degree of loose coupling among processes and resources. In this sense our model also resembles tuple-based systems [12]. Consequently, our model seems to be particularly suitable to manage distributed systems where the set of published resources is subject to frequent changes and dynamic reconfigurations.

To summarise, our approach combines the basic features of the π-calculus (i.e. dynamic communication topology of processes via name passing) with the publish-subscribe paradigm for the distributed acquisition of resources. This is our first contribution. The interplay between local and global views is also one of the novel features of our proposal. A second contribution consists in the development of a *Control Flow Analysis* (CFA) for our calculus. The analysis computes a safe approximation of resource usages. Hence, it can be used to statically check whether or not the global properties of resources usages are respected by process interactions. In particular, it helps detecting *bad usages* of resources, due to policy violations. This suggests where are sensible points in the code that need dynamic check in order to avoid policy violations.

Related Work. The primitives for resource management make our approach easy to specify a wide range of the resource behaviour of distributed systems such as Cloud Computing and Ubiquitous Computing. We believe that our approach also leverages analysis technique such as CFA and behavioural types. A simplified version of the G-Local π-calculus has been presented in [6]. The work presented here differs in several ways from the previous one. The version of the calculus we considered in this paper is more expressive of the one presented in [6] since here processes can pass resource names around. This feature was not allowed in [6]. Also, the management of resource acquisition and release is much more powerful.

P,P'	::=		*processes*	π,π'	::=		*action prefixes*
		$\mathbf{0}$	empty process			τ	internal action
	$\mid$	$\pi.P$	prefix action		$\mid$	$x(w)$	free input
	$\mid$	$(\nu z)\,P$	restriction		$\mid$	$\bar{x}w$	free output
	$\mid$	$P+P'$	choice		$\mid$	$\alpha(r)$	access action
	$\mid$	$P \parallel P'$	parallel composition		$\mid$	$rel(r)$	release action
	$\mid$	$(r,\varphi,\eta)\{P\}$	resource joint point				
	$\mid$	$req(s)\{P\}$	resource request point				
	$\mid$	$!P$	replication				

Figure 1: The syntax of G-Local π-calculus.

In [3] an extension of the λ-calculus is proposed to statically verify resource usages. Our notion of global usages is inspired by this work. The π-calculus dialect of [13] provides a general framework for checking resource usages in distributed systems. In this approach private names are extended to resources, i.e. names with a set of traces to define control over resources. Also resource request and resource release are simulated through communicating private names and structural rules respectively. This gives shared semantics of resources, i.e. several processes can have a concurrent access to resources (by communicating private names). In our approach, when a process obtains a resource, it has an exclusive access to it. Furthermore, resource entities can be dynamically reconfigured, while this is not the case in [13].

In [11], resources form a monoid and the evolution of processes and resources happens in a SCCS style. In our approach, resources are independent stateful entities equipped with their own global interaction usage policy. A dialect of the π-calculus, where resources are abstractly represented via names and can be allocated or de-allocated has been introduced in [15]. In this approach reconfigurations steps are internalized inside processes via the operations for allocating and de-allocating channels. A type system capturing safe reconfigurations over channels has been introduced. In our approach resources are more structured than channels and their reconfiguration steps are not under the control of processes. Finally, the work presented in [**?**] mainly focuses on specifying SLA by describing resources as suitable constraints. Our approach can exploit constraints to express global resource usages as well.

2 The G-Local π-Calculus

Syntax We consider the monadic version of π-calculus [14] extended with suitable primitives to declare, access and dispose resources. The syntax is displayed in Fig. 1. Here, $\mathcal{N}$ is a set of channel names (ranged over by x,y,z), $\mathcal{R}$ is a set of resource names (ranged over by r,s,t) and $\mathcal{A}$ is a set of actions (ranged over by α,β) for running over resources. We assume that these sets are pairwise disjoint. From now on, for the sake of simplicity, we often omit the trailing $\mathbf{0}$.

The input prefix $x(w).P$ binds the name w (either a channel or a resource) within the process P, while the output prefix $\bar{x}w.P$ sends the name w along channel x and then continues as P. Note that resource names can be communicated, however they cannot be used as private names and used as channels. As usual, input prefixes and restrictions act as bindings. The meaning of the remaining operators is standard. The notions of names $\mathsf{n}()$, free names $\mathsf{fn}()$, bound names $\mathsf{bn}()$ and substitution $\{-/-\}$ are defined as expected.

Our extension introduces resource-aware constructs in the π-calculus. The access prefix $\alpha(r)$ models

the invocation of the operation $\alpha \in \mathcal{A}$ over the resource bound to the variable r. Traces, denoted by $\eta, \eta' \in \mathcal{A}^*$, are finite sequences of events. A usage policy is a set of traces. The release prefix $rel(r)$ describes the operation of releasing the ownership of the resource s. In our programming model, resources are viewed as stateful entities, equipped with policies constraining their usages. More precisely, a resource is a triple (r, φ, η), where $r \in \mathcal{R}$ is a resource name, $\varphi \in \Phi$ is the associated policy and $\eta \in \mathcal{A}^*$ is a state (ε denotes the empty state). Policies specify the required properties on resource usages. Policies are usually defined by means of a resource-aware logic (see [3, 4, 9, 10]), while states keep track of the sequence of actions performed on resources, by means of (an abstraction of) execution traces.

For instance, in [3], the policies are expressed in terms of automata over an infinite alphabet, where automata steps correspond to actions on resources and final states indicate policy violations.

To cope with resource-awareness, we introduce two primitives managing resource boundaries: resource joint point $(r, \varphi, \eta)\{P\}$ and resource request point $req(r)\{P\}$. Intuitively, process P when plugged inside the resource boundary $(r, \varphi, \eta)\{P\}$ can fire actions acting over the resource r. The state η is updated at each action $\alpha(r)$ according to the required policy φ. A resource request point $req(r)\{P\}$ represents a process asking for the resource r. Only if the request is fulfilled, i.e. the required resource is available, the process can enter the required resource boundary and can use the resource r, provided that the policy is satisfied. Processes of the form $(r, \varphi, \eta)\{\mathbf{0}\}$ represent available resources. These processes are idle: they cannot perform any operation. In other words, resources can only react to requests.

Example 2.1 *To illustrate the main features of the calculus, we consider a small example, which describes a workshop with two hammers and one mallet. Tools are modelled as resource entities: hammer and mallet, with the policy φ_h (φ_m, resp.) that one can only make hard hit (soft hit, resp.) when using hammer (mallet, resp.). We model workers as a replicated process, whose instantiations take a hammer or a mallet to do jobs, whose chain is described by Jobs. Job arrivals are modelled as sending/receiving hammer and mallet on the channels x, y. Furthermore, we assume that there are two types of jobs, hard jobs on the channel x and soft jobs on the channel y, which get done by hard_hit and soft_hit actions respectively.*

The initial configuration of the workshop is given below. Resources (hammer and mallet) have empty traces. Note that we have two resources of the same name hammer, which corresponds to the number of available hammers in the workshop. Intuitively, it means that only two jobs, which use hammers, can be concurrently done. We have a sequence of four jobs described by the process Jobs.

$$
\begin{aligned}
Tools &::= (hammer, \varphi_h, \varepsilon)\{\mathbf{0}\} \mid (hammer, \varphi_h, \varepsilon)\{\mathbf{0}\} \mid (mallet, \varphi_m, \varepsilon)\{\mathbf{0}\} \\
Workers &::= !x(s).req(s)\{hard\_hit(s)\} \mid !y(t).req(t)\{soft\_hit(s)\} \\
Jobs &::= \overline{x}\langle hammer\rangle.\overline{y}\langle mallet\rangle.\overline{x}\langle mallet\rangle.\overline{x}\langle hammer\rangle.\mathbf{0} \\
Workshop &::= Tools \mid Workers \mid Jobs
\end{aligned}
$$

Operational semantics The operational semantics of our calculus is defined by the transition relation given in Tab. 1. Labels μ, μ' for transitions are τ for silent actions, $x(w)$ for free input, $\overline{x}v$ for free output, $\overline{x}(v)$ for bound output, $\alpha(r)$, $\alpha?\mathbf{r}$ and $\overline{\alpha(r)}$ ($rel(r)$, $rel?r$ and $\overline{rel(r)}$, resp.) for closed, open and faulty access or release actions over resource r. The effect of bound output is to extrude the sent name from the initial scope to the external environment.

We assume a notion of structural congruence and we denote it by $\equiv$. This includes the standard laws of the π-calculus, such as the monoidal laws for the parallel composition and the choice operator. To simplify the definition of our Control Flow Analysis, we impose a discipline in the choice of fresh names, and therefore to alpha-conversion. Indeed, the result of analysing a process P, must still hold

$$(\nu x)(r,\varphi,\eta)\{P\} \equiv (r,\varphi,\eta)\{(\nu x)P\}$$
$$(\nu x)req(r)\{P\} \equiv req(r)\{(\nu x)P\}$$
$$(r_2,\varphi_2,\eta_2)\{\mathbf{0}\} \parallel (r_1,\varphi_1,\eta_1)\{P\} \equiv (r_1,\varphi_1,\eta_1)\{(r_2,\varphi_2,\eta_2)\{\mathbf{0}\} \parallel P\}$$

Figure 2: Structural congruence.

for all its derivative processes Q, including all the processes obtained from Q by alpha-conversion. In particular, the CFA uses the names and the variables occurring in P. If they were changed by the dynamic evolution, the analysis values would become a sort of dangling references, no more connected with the actual values. To statically maintain the identity of values and variables, we partition all the names used by a process into finitely many equivalence classes. We denote with $\lfloor n \rfloor$ the equivalence class of the name n, that is called *canonical name* of n. Not to further overload our notation, we simply write n for $\lfloor n \rfloor$, when unambiguous. We further demand that two names can be alpha-renamed only when they have the same canonical name.

In addition, we introduce specific laws for managing the resource-aware constructs, reported in Fig. 2. If two processes P_1 and P_2 are equivalent, then also P_1 and P_2 when plugged inside the same resource boundaries are. Resource request and resource joint points can be swapped with the restriction boundary since restriction is not applied to resource names but only to channel names. The last law is crucial for managing the discharge of resources. This law allows rearrangements of available resources, e.g. an available resource is allowed to enter or escape within a resource boundary.

The rules *Act*, *Par*, *Res*, *Comm*, *Cong*, *Choice*, *Open* and *Close* are the standard π-calculus ones. The rule *Act* describes actions of processes, e.g. the silent action, free input and free output. Concretely, $\overline{x}w.P$ sends the name w along the channel x and then behaves like P, while $x(w).P$ receives a name via the channel x, to which w is bound, and then behaves like P. We only observe that our semantics is a late one, e.g. w is actually bound to a value when a communication occurs. Finally, $\tau.P$ performs the silent action τ and then behaves like P.

The rule *Par* expresses the parallel computation of processes, while the rule *Choice* represents a choice among alternatives. The rule *Comm* is used to communicate free names. The rules *Res* and *Open* are rules for restriction. The first ensures that an action of P is also an action of $(\nu z)P$, provided that the restricted name z is not in the action. In the case of z in the action, the rule *Open* transforms a free output action $\overline{x}z$ into a bound output action $\overline{x}(z)$, which basically expresses opening scope of a bound name. The rule *Close* describes communication of bound names, which also closes the scope of a bound name in communication.

We are now ready to comment on the semantic rules corresponding to the treatment of resources. The rule Act_R models a process that tries to perform an action α (*rel*, resp.) on the resource r. This attempt is seen as an *open action*, denoted by the label $\alpha?r$ (*rel*$?r$, resp.).

Intuitively, if the process is inside the scope of r (see the rule $Local_1$), and the action satisfies the policy for r, then the attempt will be successful and the corresponding action will be denoted by the label $\alpha(r)$ (see the rule $Policy_1$). If this is not the case, the process is stuck. Similarly, if the process tries to release a resource with the action *rel*.

We introduce the rule $Comm_R$ to model the communication of resource names between processes.

When a resource r is available, then it can be acquired by a process P that enters the corresponding resource boundary (r,φ,η), as stated by the rule *Acquire*.

Symmetrically, according to the rule *Release*, the process P can release an acquired resource r and update the state of its resources by appending *rel* to η. In the resulting process, the process P escapes the

$$\text{(Act)}\quad \pi.P \xrightarrow{\pi} P \quad \pi \neq \alpha(r), rel(r) \qquad \text{(Cong)}\quad \frac{P_1 \equiv P_1' \;\; P_1' \xrightarrow{\mu} P_2' \;\; P_2' \equiv P_2}{P_1 \xrightarrow{\mu} P_2}$$

$$\text{(Par)}\quad \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \parallel P_2 \xrightarrow{\mu} P_1' \parallel P_2}\ \mathsf{bn}(\mu) \cap \mathsf{fn}(P_2) = \varnothing \qquad \text{(Choice)}\quad \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 + P_2 \xrightarrow{\mu} P_1'}$$

$$\text{(Res)}\quad \frac{P \xrightarrow{\mu} P'}{(\nu z)P \xrightarrow{\mu} (\nu z)P'}\ z \notin \mathsf{n}(\mu) \qquad \text{(Open)}\quad \frac{P \xrightarrow{\bar{x}y} P'}{(\nu y)P \xrightarrow{\bar{x}(y)} P'}\ y \neq x$$

$$\text{(Comm)}\quad \frac{P_1 \xrightarrow{\bar{x}y} P_1' \;\; P_2 \xrightarrow{x(z)} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} P_1' \parallel P_2'\{y/z\}}) \qquad \text{(Close)}\quad \frac{P_1 \xrightarrow{x(z)} P_1' \;\; P_2 \xrightarrow{\bar{x}(y)} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} (\nu y)(P_1' \parallel P_2'\{y/z\})}$$

$$\text{(Act}_\text{R}\text{)}\quad \begin{array}{l} \alpha(r).P \xrightarrow{\alpha?r} P \\ rel(r).P \xrightarrow{rel?r} P \end{array} \qquad \text{(Comm}_\text{R}\text{)}\quad \frac{P_1 \xrightarrow{\bar{x}r} P_1' \;\; P_2 \xrightarrow{x(s)} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} P_1' \parallel P_2'\{r/s\}}$$

$$\text{(Acquire)}\quad req(r)\{P\} \parallel (r, \varphi, \eta)\{\mathbf{0}\} \xrightarrow{\tau} (r, \varphi, \eta)\{P\}$$

$$\text{(Release)}\quad \frac{P \xrightarrow{rel?r} P'}{(r, \varphi, \eta)\{P\} \xrightarrow{rel(r)} (r, \varphi, \eta.rel)\{\mathbf{0}\} \parallel P'}$$

$$\text{(Policy}_1\text{)}\quad \frac{P \xrightarrow{\alpha?r} P' \quad \eta.\alpha \vDash \varphi}{(r, \varphi, \eta)\{P\} \xrightarrow{\alpha(r)} (r, \varphi, \eta.\alpha)\{P'\}} \qquad \text{(Policy}_2\text{)}\quad \frac{P \xrightarrow{\alpha?r} P' \quad \eta.\alpha \nvDash \varphi}{(r, \varphi, \eta)\{P\} \xrightarrow{\overline{\alpha(r)}} (r, \varphi, \eta)\{\mathbf{0}\} \parallel P'}$$

$$\text{(Local}_1\text{)}\quad \frac{P \xrightarrow{\mu} P'}{(r, \varphi, \eta)\{P\} \xrightarrow{\mu} (r, \varphi, \eta)\{P'\}}\ r \notin \mathsf{n}(\mu) \qquad \text{(Local}_2\text{)}\quad \frac{P \xrightarrow{\mu} P'}{req(r)\{P\} \xrightarrow{\mu} req(r)\{P'\}}\ r \notin \mathsf{n}(\mu)$$

$$\text{(Appear)}\quad P \xrightarrow{\tau} P \parallel (r, \varphi, \eta)\{\mathbf{0}\} \qquad \text{(Disappear)}\quad (r, \varphi, \eta)\{P\} \xrightarrow{\tau} \mathbf{0}$$

Table 1: Operational Semantics.

resource boundary. Furthermore, the resource becomes available, i.e. it encloses the empty process $\mathbf{0}$. If the process is not inside the scope of r (see the rule $Local_1$), then, as in the case of accesses, the process is stuck.

The rules $Policy_1, Policy_2$ check whether the execution of the action α on the resource r obeys the policy φ, i.e. whether the updated state $\eta.\alpha$, obtained by appending α to the current state η, is consistent w.r.t. φ. If the policy is obeyed, then the updated state $\eta.\alpha$ is stored in the resource state according to the rule $Policy_1$ and the action becomes *closed* and if not, then the resource is forcibly released according to the rule $Policy_2$ and the action becomes *faulty*. Notice that $Policy_2$ is the rule managing the recovery from bad access to resources.

The rules $Local_1$ and $Local_2$ express that actions can bypass resource boundaries for r only if they do not involve the resource r.

Finally, the rules *Appear* and *Disappear* describe the abstract behaviour of the resource manager performing asynchronous resource reconfigurations. In other words, resource configuration is not under

the control of processes. Resources are created and destroyed by external entities and processes can only observe their presence/absence. This is formally represented by the rules *Appear* and *Disappear*.

Example 2.2 *To explain the operational semantics, we come back to our running example. The following trace illustrates how the workshop works. At the beginning, Workers instantiates a new worker (a resource request point) when receiving a hard job:*

$$\begin{array}{l} Workshop \\ \equiv Workers|Tools|x(s).req(s)\{hard\_hit(s)\}|\overline{x}\langle hammer\rangle.Jobs' \\ \xrightarrow{\tau} Workers|Tools|Jobs'|req(hammer)\{hard\_hit(hammer)\}, \end{array}$$

where $Jobs' ::= \overline{y}\langle mallet\rangle.\overline{x}\langle mallet\rangle.\overline{x}\langle hammer\rangle$. *At this point the new worker can take a hammer and other jobs are also available (on the channel* x,y*). In the following, for the sake of simplicity, we only show sub-processes that involve computation. Assume that the new worker takes a hammer, then we have the following transition:*

$$\begin{array}{l} req(hammer)\{hard\_hit(hammer)\}|(hammer,\varphi_h,\varepsilon)\{\mathbf{0}\} \\ \xrightarrow{\tau} (hammer,\varphi_h,\varepsilon)\{hard\_hit(hammer)\} \end{array}$$

Now, three workers are similarly instantiated for doing all remaining jobs.

$$\begin{array}{l} Workers|Jobs' \\ \xrightarrow{\tau} Workers|req(mallet)\{soft\_hit(mallet)\}|\overline{x}\langle mallet\rangle.\overline{x}\langle hammer\rangle \\ \xrightarrow{\tau} Workers|req(mallet)\{soft\_hit(mallet)\}|req(mallet)\{hard\_hit(mallet).\}|\overline{x}\langle hammer\rangle \\ \xrightarrow{\tau} Workers|req(mallet)\{soft\_hit(mallet)\}|req(mallet)\{hard\_hit(mallet)\}|req(hammer)\{hard\_hit(mallet)\} \end{array}$$

In the current setting, the new three workers make one request on the remaining hammer and two requests on the mallet. Since we have only one mallet, one of two mallet requests could be done at a time. Suppose the first job get done first, we have the following transition:

$$\begin{array}{l} (hammer,\varphi_h,\varepsilon)\{hard\_hit(hammer)\} \\ \xrightarrow{hard\_hit(hammer)} (hammer,\varphi_h,hard\_hit)\{\mathbf{0}\} \end{array}$$

Note that the hammer is available again. Similarly, the second job is done as follows:

$$\begin{array}{l} req(mallet)\{soft\_hit(mallet)\}|(mallet,\varphi_m,\varepsilon)\{\mathbf{0}\} \\ \xrightarrow{\tau} (mallet,\varphi_m,\varepsilon)\{soft\_hit(mallet)\} \\ \xrightarrow{soft\_hit(mallet)} (mallet,\varphi_m,soft\_hit)\{\mathbf{0}\} \end{array}$$

If the third job would be processed, then a forced release could occur. This happens because the worker attempts to do a hard hit by using a mallet in doing the job, which violates the mallet policy.

$$\begin{array}{l} req(mallet)\{hard\_hit(mallet).\mathbf{0}\}|(mallet,\varphi_m,\varepsilon)\{\mathbf{0}\} \\ \xrightarrow{\tau} (mallet,\varphi_m,\varepsilon)\{hard\_hit(mallet)\} \\ \xrightarrow{\overline{hard\_hit(hammer)}} (hammer,\varphi_h,\varepsilon)\{\mathbf{0}\}|\mathbf{0} \end{array}$$

Finally, the similar trace is for the fourth job.

3 Control Flow Analysis

In this section, we present a CFA for our calculus, extending the one for π-calculus [5]. The CFA computes a safe over-approximation of all the possible communications of resource and channel names on channels. Furthermore, it provides an over-approximation of all the possible usage traces on the given resources and records the names of the resources that can be possibly not released, thus providing information on possible bad usages. The analysis is performed under the perspective of processes. This amounts to saying that the analysis tries to answer the following question: "Are the resources initially granted sufficient to guarantee a correct usage?". In other words, we assume that a certain fixed amounts of resources is given and we do not consider any dynamic reconfiguration, possible in our calculus, due to the rules *Appear* and *Disappear*. The reconfiguration is up to the resource manager and is not addressed by the CFA.

For the sake of simplicity, we provide the analysis for a subset of our calculus, in which processes enclosed in the scopes of resources are *sequential processes* (ranged over by Q, Q'), as described by the following syntax. Intuitively, a sequential process represents a single thread of execution in which one or more resources can be used.

$$
\begin{array}{lcl@{\qquad}lcl}
P,P' & ::= & \text{as before in Fig.1} & Q,Q' & ::= & \textit{sequential processes} \\
 & | & (r,\varphi,\eta)\{Q\} & & & \mathbf{0} \\
 & | & req(s)\{Q\} & & | & (\nu z)\ Q \\
 & & & & | & \pi.Q \\
 & & & & | & Q+Q' \\
 & & & & | & (r,\varphi,\eta)\{Q\} \\
 & & & & | & (r,\varphi,\eta)\{\mathbf{0}\}\|Q \\
 & & & & | & req(s)\{Q\}
\end{array}
$$

This implies that one single point for releasing each resource occurs in each non deterministic branch of a process. The extension to general parallel processes is immediate. Nevertheless, it requires some more complex technical machinery in order to check whether all the parallel branches synchronise among them, before releasing the shared resource.

In order to facilitate our analysis, we further associate labels $\chi \in \mathcal{L}$ with resource boundaries as follows: $(r,\varphi,\eta)\{Q\}^{\chi}$ and $req(r)\{Q\}^{\chi}$, in order to give a name to the sub-processes in the resource scopes. Note that this annotation can be performed in a pre-processing step and does not affect the semantics of the calculus. During the computation, resources are released and acquired by other processes. Statically, sequences of labels $S \in \mathcal{L}^*$ are used to record the sequences of sub-processes possibly entering the scope of a resource. Furthermore, to make our analysis more informative, we enrich the execution traces η with special actions that record the fact that a resource has been possibly:

- acquired by the process labelled χ: $in(\chi)$, with a successful request;
- released by the process labelled χ: $out(\chi)$ with a successful release;
- taken away from the process labelled χ: $err\_out(\chi)$ because of an access action on r that does not satisfy the policy.

The new set of traces is $\hat{\mathcal{A}}^*$, where $\hat{\mathcal{A}} = \mathcal{A} \cup \{in(\chi), out(\chi), err\_out(\chi) \mid \chi \in \mathcal{L}\}$. The corresponding dynamic traces can be obtained by simply removing all the special actions.

The result of analysing a process P is a tuple $(\rho,\kappa,\Gamma,\Psi)$ called *estimate* of P, that provides an approximation of resource behavior. More precisely, ρ and κ offer an over-approximation of all the possible values that the variables in the system may be bound to, and of the values that may flow on

channels. The component Γ provides a set of traces of actions on each resource. Finally, Ψ records a set of the resources that can be possibly not released. Using this information, we can statically check resource usages against the required policies.

To validate the correctness of a given estimate $(\rho,\kappa,\Gamma,\Psi)$, we state a set of clauses that operate upon judgments in the form $(\rho,\kappa,\Gamma,\Psi)\models^{\delta} P$, where δ is a sequence of pairs $[(r,\varphi,\eta),S]$, recording the resource scope nesting. This sequence is initially empty, denoted by $[\varepsilon,\varepsilon]$.

The analysis correctly captures the behavior of P, i.e. the estimate $(\rho,\kappa,\Gamma,\Psi)$ is valid for all the derivatives P' of P. In particular, the analysis keeps track of the following information:

- An approximation $\rho:\mathcal{N}\cup\mathcal{R}\to\wp(\mathcal{N}\cup\mathcal{R})$ of names bindings. If $a\in\rho(x)$ then the channel variable x can assume the channel value a. Similarly, if $r\in\rho(s)$ then the resource variable s can assume the resource value r.
- An approximation $\kappa:\mathcal{N}\to\wp(\mathcal{N}\cup\mathcal{R})$ of the values that can be sent on each channel. If $b\in\kappa(a)$, then the channel value b can be output on the channel a, while $r\in\kappa(a)$, then the resource value r can be output on the channel a.
- An approximation $\Gamma:\mathcal{R}\to\wp(\{[(\varphi,\eta),S]|\ \varphi\in\Phi,S\in\mathcal{L}^*,\eta\in\hat{\mathcal{A}}^*\})$ of resource behavior. If $[(\varphi,\eta),S]\in\Gamma(r)$ then η is one of the possible traces over r that is performed by a sequence of sub-processes, whose labels χ are juxtaposed in S.
- An approximation $\Psi\in\wp(\{\delta\mid\delta$ is a sequence of pairs $[(r,\varphi,\eta),S]\}$ of the resources which are possible locked by processes in deadlock for trying to access or to release a resource not in their scope. More precisely, if δ is in Ψ and $[(r,\phi,\eta),S]$ occurs in δ, then the resource r can be possibly acquired by a process that can be stuck and that therefore could not be able to release it.

The judgments of the CFA are given in Tab. 2, which are based on structural induction of processes. We use the following shorthands to simplify the treatment of the sequences δ. The predicate $[(r,\varphi,\eta),\chi]\Subset\delta$ is used to check whether the pair $[(r,\varphi,\eta),\chi]$ occurs in δ, i.e. whether $\delta=\delta'[r,(\varphi,\eta),\chi]\delta''$. With $\delta\{[(r,\varphi,\eta.\alpha),S]/[(r,\varphi,\eta),S]\}$ we indicate that the pair $[(r,\varphi,\eta),S]$ is replaced by $[(r,\varphi,\eta.\alpha),S]$ in the sequence δ. With $\delta\setminus[(r,\varphi,\eta),S]$ we indicate the sequence where the occurrence $[(r,\varphi,\eta),S]$ has been removed, i.e. the sequence $\delta'\delta''$, if $\delta=\delta'[(r,\phi,\eta),S]\delta''$.

All the clauses dealing with a compound process check that the analysis also holds for its immediate sub-processes. In particular, the analysis of $!P$ and that of $(\nu x)P$ are equal to the one of P. This is an obvious source of imprecision (in the sense of over-approximation). We comment on the main rules. Besides the validation of the continuation process P, the rule for output, requires that the set of names that can be communicated along each element of $\rho(x)$ includes the names to which y can evaluate. Symmetrically, the rules for input demands that the set of names that can pass along x is included in the set of names to which y can evaluate. Intuitively, the estimate components take into account the possible dynamics of the process under consideration. The clauses' checks mimic the semantic evolution, by modelling the semantic preconditions and the consequences of the possible synchronisations. In the rule for input, e.g., CFA checks whether the precondition of a synchronisation is satisfied, i.e. whether there is a corresponding output possibly sending a value that can be received by the analysed input. The conclusion imposes the additional requirements on the estimate components, necessary to give a valid prediction of the analysed synchronisation action, mainly that the variable y can be bound to that value.

To gain greater precision in the prediction of resource usages, in the second rule, the continuation process is analysed, for all possible bindings of the resource variable s. This explains why we have all the other rules for resources, without resource variables.

$(\rho,\kappa,\Gamma,\Psi) \models^{\delta} \mathbf{0}$	iff true
$(\rho,\kappa,\Gamma,\Psi) \models^{\delta} \tau.P$	iff $(\rho,\kappa,\Gamma,\Psi) \models^{\delta} P$
$(\rho,\kappa,\Gamma,\Psi) \models^{\delta} \overline{x}w.P$	iff $\forall a \in \rho(x) : \rho(w) \subseteq \kappa(a) \ \wedge \ (\rho,\kappa,\Gamma,\Psi) \models^{\delta} P$
$(\rho,\kappa,\Gamma,\Psi) \models^{\delta} x(y).P$	iff $\forall a \in \rho(x) : \kappa(a) \cap \mathcal{N} \subseteq \rho(y) \ \wedge \ (\rho,\kappa,\Gamma,\Psi) \models^{\delta} P$
$(\rho,\kappa,\Gamma,\Psi) \models^{\delta} x(s).P$	iff $\forall a \in \rho(x) : \kappa(a) \cap \mathcal{R} \subseteq \rho(s)$ $\wedge \ \forall r \in \rho(s) : (\rho,\kappa,\Gamma,\Psi) \models^{\delta} P\{r/s\}$
$(\rho,\kappa,\Gamma,\Psi) \models^{\delta} P_1 + P_2$	iff $(\rho,\kappa,\Gamma,\Psi) \models^{\delta} P_1 \wedge (\rho,\kappa,\Gamma,\Psi) \models^{\delta} P_2$
$(\rho,\kappa,\Gamma,\Psi) \models^{\delta} P_1 \parallel P_2$	iff $(\rho,\kappa,\Gamma,\Psi) \models^{\delta} P_1 \wedge (\rho,\kappa,\Gamma,\Psi) \models^{\delta} P_2$
$(\rho,\kappa,\Gamma,\Psi) \models^{\delta} (\nu x)P$	iff $(\rho,\kappa,\Gamma,\Psi) \models^{\delta} P \wedge x \in \rho(x)$
$(\rho,\kappa,\Gamma,\Psi) \models^{\delta} !P$	iff $(\rho,\kappa,\Gamma,\Psi) \models^{\delta} P$
$(\rho,\kappa,\Gamma,\Psi) \models^{\delta} (r,\varphi,\eta)\{Q\}^{S}$	iff $(\rho,\kappa,\Gamma,\Psi) \models^{\delta[(r,\varphi,\eta),S]} Q$
$(\rho,\kappa,\Gamma,\Psi) \models^{\delta} (r,\varphi,\eta)\{\mathbf{0}\}^{S}$	iff $[(\varphi,\eta),S] \in \Gamma(r)$
$(\rho,\kappa,\Gamma,\Psi) \models^{\delta} req(r)\{Q\}^{\chi}$	iff $\forall [(\varphi,\eta),S] \in \Gamma(r) \wedge \chi \notin S \wedge$ $\Rightarrow (\rho,\kappa,\Gamma,\Psi) \models^{\delta[(r,\varphi,\eta.in(\chi)),S\chi]} Q$
$(\rho,\kappa,\Gamma,\Psi) \models^{\delta} \alpha(r).Q$	iff $[(r,\varphi,\eta),S\chi] \Subset \delta \wedge \eta.\alpha \models \varphi \Rightarrow (\rho,\kappa,\Gamma,\Psi) \models^{\delta'} Q$ $\wedge \ [(r,\varphi,\eta),S\chi] \Subset \delta \wedge \eta.\alpha \not\models \varphi$ $\Rightarrow [(\varphi,\eta.err\_out(\chi)),S\chi] \in \Gamma(r) \wedge (\rho,\kappa,\Gamma,\Psi) \models^{\delta''} Q$ $\wedge \ [(r,\varphi,\eta),S\chi] \notin \delta \Rightarrow \delta \in \Psi$ with $\delta' = \delta\{[(r,\varphi,\eta.\alpha),S\chi]/[(r,\varphi,\eta),S\chi]\}$ and $\delta'' = \delta \setminus [(r,\varphi,\eta),S\chi]$
$(\rho,\kappa,\Gamma,\Psi) \models^{\delta} \omega(r).Q$	iff $[(r,\varphi,\eta),S\chi] \Subset \delta \Rightarrow (\rho,\kappa,\Gamma,\Psi) \models^{\delta \setminus [(r,\varphi,\eta),S\chi]} Q$ $\wedge \ [(\varphi,\eta.\omega.out(\chi)),S\chi] \in \Gamma(r)$ $\wedge \ [(r,\varphi,\eta),S\chi] \notin \delta \Rightarrow \delta \in \Psi$
$(\rho,\kappa,\Gamma,\Psi) \models^{\delta} (r,\varphi,\eta)\{\mathbf{0}\}^{S} \parallel Q$	iff $(\rho,\kappa,\Gamma,\Psi) \models^{\delta} (r,\varphi,\eta)\{\mathbf{0}\}^{S} \wedge (\rho,\kappa,\Gamma,\Psi) \models^{\delta} Q$

Table 2: CFA Equational Laws

The rule for *resource joint point* updates δ to record that the immediate sub-process is inside the scope of the new resource and there it is analysed. If the process is empty, i.e. in the case the resource is available, the trace of actions is recorded in $\Gamma(r)$.

In the rule for *resource request point*, the analysis for Q is performed for every possible element $[(\varphi,\eta),S]$ from the component $\Gamma(r)$. This amounts to saying that the resource r can be used starting from any possible previous trace η. In order not to append the same trace more than once, we have the condition that S does not contain χ. This prevents the process labelled χ to do it. Furthermore, η is enriched by the special action $in(\chi)$ that records the fact that the resource r can be possibly acquired by the process labelled χ.

According to the rule for *access action*, if the pair $[(r,\varphi,\eta),S\chi]$ occurs in δ (i.e. if we are inside the resource scope of r) and the updated history $\eta.\alpha$ obeys the policy φ, then the analysis result also holds for the immediate subprocess and δ is updated in δ', by replacing $[(r,\varphi,\eta),S\chi]$ in δ with $[(r,\varphi,\eta.\alpha),S\chi]$, therefore recording the resource accesses to r possibly made by the sub-process labelled by χ.

In case the action possibly violates the policy associated with r (see the last conjunct), the process labelled χ may loose the resource r, as recorded by the trace in Γ, $[(\varphi,\eta.err\_out(\chi)),S\chi]$, with the special action $err\_out(\chi)$ appended to η. If instead, the action on r is not viable because the process is not in the scope of r, then all the resources in the context δ could not be released, as recorded by the component Ψ.

According to the rule for *release*, the trace of actions $\eta' = \eta.\omega.out(\chi)$ over r at χ is recorded in $\Gamma(r)$. Other sub-processes can access the resource starting from the trace η'. Furthermore, $[(r,\varphi,\eta),S]$ is removed from δ and this reflects the fact that the process Q can exit its scope, once released the resource r. Similarly, in the last rule, $[(r,\varphi,\eta),S]$ is removed from δ and there the process Q is analysed. Again, if the action on r is not possible because the process is not in the scope of r, then all the resource in the context δ could not be released, as recorded by the component Ψ.

Example 3.1 *We briefly interpret the results of CFA on our running example. A more complex of exemplification of CFA is given in the next example (see below). First we associate labels with the resource boundaries as follows:*

$$\begin{array}{rl} Tools ::= & (hammer,\varphi_h,\varepsilon)\{\mathbf{0}\}^{\chi_1} | (hammer,\varphi_h,\varepsilon)\{\mathbf{0}\}^{\chi_2} | (mallet,\varphi_m,\varepsilon)\{\mathbf{0}\}^{\chi_3} \\ Workers ::= & !x(s).req(s)\{hard\_hit(s)\}^{\chi_h} | !y(t).req(t)\{soft\_hit(s)\}^{\chi_m} \end{array}$$

It is easy to see that there is one policy violation, which is captured by our CFA in the component $\Gamma(hammer)$, from which we can extract the following trace: $(in(\chi_m).err\_out(\chi_m),\chi_m)$. It occurs when doing the third job the worker tries to hit hard using a mallet. We know that the channel x (y, resp.) is supposed to send/receiving hard jobs (soft jobs, resp.), i.e. sending/receiving hammer (mallet, resp.) and names s and t are supposed to be bound to hammer and mallet respectively. By checking the component ρ and κ, we can explain the above violation too. On the one hand, we found that $\rho(t)$ is a singleton set of mallet, while $\rho(s)$ is a set of hammer and mallet, which is a wrong bound of s. On the other hand, similarly we found that $\kappa(x)$ contains only hammer, while $\kappa(y)$ contains hammer and mallet, which is a wrong use of y.

Example 3.2 (Robot Scenario) *We now consider a scenario, where a set of robots collaborate to reach a certain goal, e.g. to move an item from one position to another. Without loss of generality, we assume that robots operate in a space represented by a two-dimensional grid. We also assume that certain positions over the grid are faulty, and therefore they cannot be crossed by robots. To move the item, a robot needs to take it, and this is allowed provided that the item is co-located within the range of robot's*

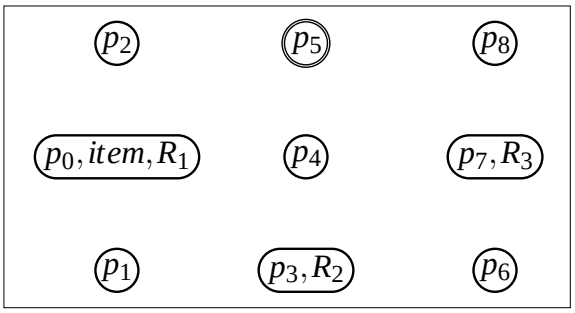


Figure 3: The initial configuration of the robot scenario.

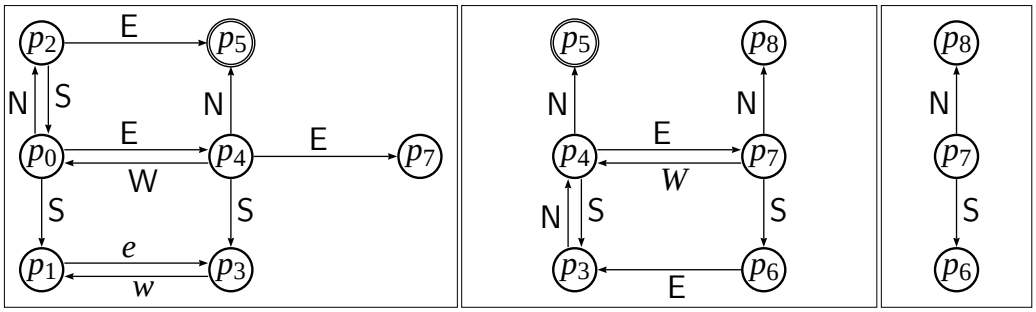


Figure 4: The policy automata of the robots' families: R_1 (left), R_2 (middle) and R_3 (right).

sensor. Moreover, since robots have a small amount of energy power, they can perform just a few of steps with the item. Finally, we consider three families of robots (R_1, R_2 and R_3): each robot in the family has different computational capabilities.

Fig. 3 gives a pictorial description of the initial configuration of the scenario. Positions are represented by circles and double circles. Double circles indicate faulty positions. The item is located at position p_0 and the goal is to move it into the position p_8. There is just one faulty position p_5, crossing through which is considered a failure. Moreover, we consider a scenario where the three families of robots R_1, R_2 and R_3 are initially located at p_0, p_3 and p_7, respectively (e.g. all the robots of the family R_1 are located at p_0).

Sensors are modelled by clearly identified resources. The sensor j^{th} of the i^{th} robot family is specified by the resource $(sns_{i,j}, \varphi_j, \eta_{i,j})$, where $sns_{i,j}$ is the name of the sensor, $\eta_{i,j}$ is the abstract representation of the sequence of moving actions which led the robot from its initial position to the current one and initially equals to ε, and φ_j is the global policy on demand. We assume that each family of robots has its own policy described by the automata in Fig. 4. The policy constraints robots' movement in the grid. We model the movement activities of robots with the following actions: $\mathsf{E}(sns)$, $\mathsf{W}(sns)$, $\mathsf{S}(sns)$, *and* $\mathsf{N}(sns)$ *that describe the movements on east (west, south and north, resp.). Basically, sensors are a sort of private resources of the robots (each robot will never release its sensor) and the actions over sensors update their states.*

The item is modelled by a resource of the form (IT, φ_I, η), where η describes the sequence of actions performed on the item, and φ_I simply states that the item is never located at the position p_5. Initially, η is equal to ε. The same set of actions adopted for robots' movement (namely $\mathsf{E}(IT)$, $\mathsf{W}(IT)$, $\mathsf{S}(IT)$, *and* $\mathsf{N}(IT)$*) are exploited to transport the item in the grid. Finally, each robot in the family $i \in \{1,2,3\}$ is specified by a process $R_{i,j}$ of the form: $(sns_{i,j}, \varphi_j, \eta_{i,j})\{Q_{i,j}\}^{\chi}$, where $Q_{i,j}$ specifies the j^{th} robot's*

behaviour of the i^{th} robot family and χ is a label associated with the resource boundary. For instance, in the process $Q_{2,3}$ (see below), the robot goes to north (without the item), then it tries to grasp the item. If this operation succeeds, the robot goes to east and releases the item there. Note that we use two monadic actions to move the item and the sensor together. This could be done by using polyadic actions, which however we leave for future work.

For the sake of simplicity, we do not model co-location of sensors and items. The specification of the robot scenario is given below.

$$
\begin{array}{l}
R_{1,1} := (sns_{1,1}, \varphi_1, p_0)\{req(IT)\{\mathsf{E}(IT).\mathsf{E}(sns_{1,1}).\mathsf{S}(IT).\mathsf{S}(sns_{1,1}).rel(IT)\}^{\chi_{r11}}\}^{\chi_{s11}}\\
R_{1,2} := (sns_{1,2}, \varphi_1, p_0)\{req(IT)\{\mathsf{E}(IT).\mathsf{E}(sns_{1,2}).\mathsf{E}(IT).\mathsf{E}(sns_{1,2}).rel(IT)\}^{\chi_{r12}}\}^{\chi_{s12}}\\
R_{1,3} := (sns_{1,3}, \varphi_1, p_0)\{req(IT)\{\mathsf{E}(IT).\mathsf{E}(sns_{1,3}).rel(IT)\}^{\chi_{r13}}\}^{\chi_{s13}}\\
R_{2,1} := (sns_{2,1}, \varphi_2, p_3)\{req(IT)\{\mathsf{N}(IT).\mathsf{N}(sns_{2,1}).\mathsf{E}(IT).\mathsf{E}(sns_{2,1}).rel(IT)\}^{\chi_{r21}}\}^{\chi_{s21}}\\
R_{2,2} := (sns_{2,2}, \varphi_2, p_3)\{req(IT)\{\mathsf{N}(IT).\mathsf{N}(sns_{2,2}).\mathsf{N}(IT).\mathsf{N}(sns_{2,2}).rel(IT)\}^{\chi_{r22}}\}^{\chi_{s22}}\\
R_{2,3} := (sns_{2,3}, \varphi_2, p_3)\{NR(sns_{2,3}).req(IT)\{\mathsf{E}(IT).\mathsf{E}(sns_{2,2}).rel(IT)\}^{\chi_{r23}}\}^{\chi_{s23}}\\
R_{3,1} := (sns_{3,1}, \varphi_3, p_7)\{req(IT)\{\mathsf{S}(IT).\mathsf{S}(sns_{3,1}).rel(IT)\}^{\chi_{r31}}\}^{\chi_{s31}}\\
R_{3,2} := (sns_{3,2}, \varphi_3, p_7)\{req(IT)\{\mathsf{N}(IT).\mathsf{N}(sns_{3,2}).rel(IT)\}^{\chi_{r32}}\}^{\chi_{s32}}
\end{array}
$$

$$
System := (IT, \varphi_I, p_0)\{\mathbf{0}\}^{\chi_{IT}} \parallel R_{1,1} \parallel R_{1,2} \parallel R_{1,3} \parallel R_{2,1} \parallel R_{2,2} \parallel R_{2,3} \parallel R_{3,1} \parallel R_{3,2}
$$

The following trace illustrates the behaviour of the specification of the scenario. At the beginning, the item lies in the range of the family of robot R_1. Then a reconfiguration step putting together the robot $R_{1,1}$ and the item is performed.

$$
\begin{array}{c}
System := (IT, \varphi_I, \varepsilon)\{\mathbf{0}\}||(sns_{1,1}, \varphi_1, \varepsilon)\{Q_{1,1}\}||R_{1,2}||R_{1,3}||R_{2,1}||R_{2,2}||R_{2,3}||R_{3,1}||R_{3,2} \equiv\\
(sns_{1,1}, \varphi_1, \varepsilon)\{(IT, \varphi_I, \varepsilon)\{\mathbf{0}\}||Q_{1,1}\}||R_{1,2}||R_{1,3}||R_{2,1}||R_{2,2}||R_{2,3}||R_{3,1}||R_{3,2}
\end{array}
$$

As a result, robot $R_{1,1}$ can grasp (acquire) the item; the pair item-robot moves on east, then on south. Finally, the robot disposes the item at the position p_3.

$$
\begin{array}{rl}
System \xrightarrow{\tau} & (sns_{1,1}, \varphi_1, p_0)\{(IT, \varphi_I, \varepsilon)\{Q_{1,1}\}||R_{1,2}||R_{2,1}||R_{2,2}||R_{3,1}||R_{3,2}\\
& \xrightarrow{E(IT)}\xrightarrow{E(sns_{1,1})}\xrightarrow{S(IT)}\xrightarrow{S(sns_{1,1})}\xrightarrow{rel(IT)}\\
& (IT, \varphi_I, \varepsilon.E.S.rel)\{\mathbf{0}\}||(sns_{1,1}, \varphi_1, \varepsilon.E.S)\{\mathbf{0}\}||R_{1,2}||R_{2,1}||R_{2,2}||R_{3,1}||R_{3,2}
\end{array}
$$

It is easy, given an initial location, to map a sequence of actions performed over the item into a path on the grid, namely each action operated over the item (i.e. $\mathsf{E}(IT)$, $\mathsf{W}(IT)$, $\mathsf{S}(IT)$, and $\mathsf{N}(IT)$) corresponds to a single moving step in the space grid. The release action, instead, is interpreted as a sort of self-loop in the grid, i.e. the execution of the release action does not move the item. For example, the sequence $\varepsilon.E.S.$ in the above setting would model the path $p_0p_4p_3$. From now on, by abuse of notation, we will freely use paths in place of sequences of actions over the item/sensors.

Now, the item is in the range of the family of robots R_2. Again by applying the reconfiguration step, robot $R_{2,1}$ is allowed to operate with the item. Then, it takes the item, makes a move on north, then on east, and disposes the item at the position p_7. For the sake of simplicity, in the following we show only sub-processes of the system that involve computation:

$$
\begin{array}{l}
(IT, \varphi_I, p_0p_4p_3p_3)\{\mathbf{0}\}||R_{2,1}\\
\xrightarrow{\tau}\xrightarrow{N(IT)}\xrightarrow{N(sns_{2,1})}\xrightarrow{E(IT)}\xrightarrow{E(sns_{2,1})}\xrightarrow{rel(IT)}\\
(IT, \varphi_I, p_0p_4p_3p_3p_4p_7p_7)\{\mathbf{0}\}||(sns_{2,1}, \varphi_2, p_3p_4p_7)\{\mathbf{0}\}
\end{array}
$$

Note that a forced release would have occurred at this step if the item proceeded governed by the robot $R_{2,2}$. The reason is that $R_{2,2}$ attempts to move the item into the position p_5 and this results in releasing the item at the position p_4 by the rule $Policy_2$. Now the robot $R_{3,2}$ has the chance to take the item, and, if the north move occurs, the goal is achieved and the task is completed.

$$(IT, \varphi_I, p_0p_4p_3p_3p_4p_7p_7)\{\mathbf{0}\}||R_{3,2}$$
$$\xrightarrow{\tau}\xrightarrow{N(IT)}\xrightarrow{N(sns_{3,2})}\xrightarrow{rel(IT)}$$
$$(IT, \varphi_I, p_0p_4p_3p_3p_4p_7p_7p_8p_8)\{\mathbf{0}\}||(sns_{3,2}, \varphi_3, p_7p_8)\{\mathbf{0}\}$$

Now we explain the features of the CFA. The CFA (in particular the Γ component) computes the set of possible traces of the trajectories in the grid reaching the goal, among which the ones below:

$in(\chi_{r11}).E.S.rel.out(\chi_{r11}).in(\chi_{r21}).N.E.rel.out(\chi_{r21}).in(\chi_{r32}).N.rel.out(\chi_{r32}), \chi_{r11}\cdot\chi_{r21}\cdot\chi_{r32}$
$in(\chi_{r11}).E.E.rel.out(\chi_{r11}).in(\chi_{r32}).N.rel.out(\chi_{r32}), \chi_{r12}\cdot\chi_{r32}$
$in(\chi_{r13}).E.rel.out(\chi_{r13}).in(\chi_{r23}).E.rel.out(\chi_{r23}).in(\chi_{r32}).N.rel.out(\chi_{r32}), \chi_{r13}\cdot\chi_{r23}\cdot\chi_{r32}$
$in(\chi_{r11}).E.S.rel.out(\chi_{r11}).in(\chi_{r22}).N.err\_out(\chi_{r22}).in(\chi_{r23}).E.rel.out(\chi_{r23}).in(\chi_{r32}).N.rel.out(\chi_{r32}), \chi_{r11}\cdot\chi_{r22}\cdot\chi_{r23}\cdot\chi_{r32}$

This set produces the following sequences of positions: $p_0p_4p_3p_3p_4p_7p_7p_8p_8$, $p_0p_4p_7p_7p_8p_8$, and also $p_0p_4p_4p_7p_7p_8p_8$ and $p_0p_4p_3p_3p_4p_4p_7p_7p_8p_8$. Note that the last trace is faulty (e.g. traces contain error actions err_out, see below) since it contains a forced release $err\_out(\chi_{2,2})$ (see below). Consequently, the system does not respect the policy φ_{IT} for the item. In particular, there are three faulty traces found by the analysis, which have the following common prefix:

$$in(\chi_{r11}).E.S.rel.out(\chi_{r11}).in(\chi_{r22}).N.out\_err(\chi_{r22}), \chi_{r11}\cdot\chi_{r22}$$

The reason is that the robot $R_{2,2}$ is forced to release the item when attempting to move it into the bad position p_5. Moreover, there is no faulty trace of actions over sensors, which means the system respects the policies $\varphi_{i,j}$ for sensors and therefore complies with it.

The analysis provides us with an approximation of the overall behaviour of the analysed process. Moreover, it is proved to be correct: the analysis indeed respects the operational semantics of G-Local π-calculus, as shown by the following subject reduction result.

Theorem 3.3 ***(Subject Reduction)*** *$(\rho, \kappa, \Gamma, \psi) \models^{\delta} P$ and $P \xrightarrow{\mu}{}^{*} P'$, then $(\rho, \kappa, \Gamma, \psi) \models^{\delta} P'$.*

We can further prove that there always exists a a least choice of $(\rho, \kappa, \Gamma, \psi)$ that is acceptable for CFA rules, and therefore it always exists a least estimate. This depends from the fact that the set of analysis estimates constitutes a Moore family.

Theorem 3.4 ***(Existence of estimates)*** *For all δ, P, the set $\{(\rho, \kappa, \Gamma, \psi) | (\rho, \kappa, \Gamma, \psi) \models^{\delta} P\}$ is a Moore family.*

Moreover, our analysis offers information on the resource usage, included bad usages. The component Γ is indeed in charge of recording all the possible usage traces on each resource r. Actually, for each r, traces are composed of pairs $[(\phi, \eta), S\chi]$, where S is made of labels of the processes that acquired the resource r and η records every action on r, included the special actions $in(\chi)$, $out(\chi)$ and $err\_out(\chi)$, that indicate that the process labelled χ may acquire and release (or it may be forced to release) the resource. This information offers a basis for studying dynamic properties, by suitably handling the safe over-approximation the CFA introduces. We want to focus now on the traces including special error actions, that we call *faulty*.

Definition 3.5 *A trace $\eta \in \hat{\mathcal{A}}^*$ is* faulty *if it includes $err\_out(\chi)$ for some $\chi \in \mathcal{L}$.*

In particular, on the one hand if the analysis contains faulty traces, then there is the *possibility* of policy violations, while if all the traces are not faulty, then we can prove that policy violations cannot occur at run time, and therefore that the processes correctly use their resources.

We can show it formally, as follows.

Definition 3.6 *The process P, where r is declared with policy ϕ, P* complies with *φ for r, if and only if $P \xrightarrow{\mu}{}^* P'$ implies that there is no P'' such that $P' \xrightarrow{\overline{\alpha(r)}} P''$, where $\xrightarrow{\mu}{}^*$ is the reflexive and transitive closure of $\xrightarrow{\mu}$.*

Definition 3.7 *A process P, where r is declared with policy φ, is said to* respect *φ for r, if and only if*

$$\exists(\rho,\kappa,\Gamma,\Psi).(\rho,\kappa,\Gamma,\psi)^{[\varepsilon,\varepsilon]}P \text{ and } \forall[(\varphi,\eta),S] \in \Gamma(r).\eta \text{ is not faulty}$$

Theorem 3.8 *If P* respects *the policy φ for r then, P* complies with *φ.*

4 Concluding Remarks

Our work combines the name-passing of the π-calculus with the publish-subscribe paradigm to cope with resource-awareness. We have shown that this has lead to a name passing process calculus with primitives for acquiring and releasing stateful resources. Our research program is to provide formal mechanisms underlying the definition of a resource-aware programming model. The work reported in this paper provides a first step in this direction. There is a number of ways in which our calculus could be extended. In terms of calculus design, we assumed a monadic request primitive for managing resource binding. This is a reasonable assumption for several cases. An interesting issue for future research is to extend the calculus with a polyadic request primitives asking for a finite number of resources. In terms of reasoning mechanisms, it would be interesting to exploit CFA techniques to develop methodologies to analyze the code in order to avoid bad accesses to resources. Also it would be interesting to apply the typing techniques (behavioral types) introduced in [3] to capture a notion of resource contract.

References

[1] G. Abowd & E.D. Mynatt (2000): *Charting past, present, and future research in ubiquitous computing*. *ACM Trans. Comput.-Hum. Interact.* 7, pp. 29–58, doi:10.1145/344949.344988.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica & M. Zaharia (2009): *Above the Clouds: A Berkeley View of Cloud Computing*. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley.

[3] M. Bartoletti, P. Degano, G.L. Ferrari & R. Zunino (2009): *Local Policies for Resource Usage Analysis*. *ACM Trans. Program. Lang. Syst.* doi:10.1145/1552309.1552313.

[4] M. Bartoletti & R. Zunino (2010): *A Calculus of Contracting Processes*. In: *Proc of LICS'10*, IEEE Computer Society, pp. 332–341, doi:10.1109/LICS.2010.25.

[5] C. Bodei, P. Degano, F. Nielson & H. Nielson (2001): *Static Analysis for the Pi-Calculus with Applications to Security*. *Inf. Comput.* 168(1), pp. 68–92, doi:10.1006/inco.2000.3020.

[6] C. Bodei, V. D. Dinh & G. L. Ferrari (2011): *A G-Local π-calculus*. In: *Proc. of PLACES 2011*, doi:10.4204/EPTCS.38.6. Available at `http://places11.di.fc.ul.pt/proceedings.pdf/view`.

[7] M. G. Buscemi & U. Montanari (2007): *Cc-pi: A Constraint-based Language for Specifying Service Level Agreements*. In: *Proc of ESOP'07, LNCS 4421*, Springer, pp. 18–32, doi:10.1007/978-3-540-71316-6_3.

[8] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg & I. Brandic (2009): *Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility*. *Future Gener. Comput. Syst.* 25, pp. 599–616, doi:10.1016/j.future.2008.12.001.

[9] Luís Caires (2008): *Spatial-behavioral types for concurrency and resource control in distributed systems*. *Theor. Comput. Sci.* 402(2-3), pp. 120–141, doi:10.1016/j.tcs.2008.04.030.

[10] G. Castagna, N. Gesbert & L. Padovani (2009): *A theory of Contracts for Web services*. *ACM Trans. Program. Lang. Syst.* 31(5), doi:10.1007/978-3-642-00590-9.

[11] M. Collinson & D.J. Pym (2010): *Algebra and Logic for Access Control*. *Formal Asp. Comput.* 22(3-4), pp. 483–484, doi:10.1007/s00165-010-0155-2.

[12] D. Gelernter (1985): *Generative communication in Linda*. *ACM Trans. Program. Lang. Syst.* 7(1), pp. 80–112, doi:10.1145/2363.2433.

[13] N. Kobayashi, K. Suenaga & L. Wischik (2006): *Resource Usage Analysis for the Pi-Calculus*. *Logical Methods in Computer Science* 2(3), pp. 298–312, doi:10.2168/LMCS-2(3:4)2006.

[14] D. Sangiorgi & D. Walker (2001): *Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA.

[15] E. D. Vries, A. Francalanza & M. Hennessy (2009): *Uniqueness Typing for Resource Management in Message-Passing Concurrency*. In: *Proc. of LINEARITY, EPTCS* 22, pp. 26–37, doi:10.4204/EPTCS.22.3.

[16] L. Youseff, M. Butrico & D. Da Silva (2008): *Toward a Unified Ontology of Cloud Computing*. In: *Proc of GCE '08*, pp. 1–10, doi:10.1109/GCE.2008.4738443.

Decoupled execution of synchronous coordination models via behavioural automata

José Proença Dave Clarke
IBBT-DistriNet, KUL,
Leuven, Belgium
{jose.proenca,dave.clarke}@cs.kuleuven.be

Erik de Vink
TUE, Eindhoven,
The Netherlands
evink@win.tue.nl

Farhad Arbab
CWI, Amsterdam,
The Netherlands
farhad.arbab@cwi.nl

Synchronous coordination systems allow the exchange of data by logically indivisible actions involving all coordinated entities. This paper introduces behavioural automata, a logically synchronous coordination model based on the Reo coordination language, which focuses on relevant aspects for the concurrent evolution of these systems. We show how our automata model encodes the Reo and Linda coordination models and how it introduces an explicit predicate that captures the concurrent evolution, distinguishing local from global actions, and lifting the need of most synchronous models to involve all entities at each coordination step, paving the way to more scalable implementations.

1 Introduction

Synchronous constructs in languages such as Reo [1] and Esterel [7] are useful for programming reactive systems, though in general their realisations for coordinating distributed systems become problematic. For example, it is not clear how to efficiently implement the high degrees of synchronisation expressed by Reo in a distributed context. To remedy this situation, the GALS (globally asynchronous, locally synchronous) model [9, 13] has been adopted, whereby local computation is synchronous and communication between different machines is asynchronous.

Our work contributes to the field of coordination, in particular to the Reo coordination language, by incorporating the same ideas behind GALS in our approach to execute synchronisation models. More specifically, we introduce *behavioural automata* to model synchronous coordination, inspired in Reo [6]. Each step taken by an automata corresponds to a round of "synchronous" actions performed by the coordination layer, where data flow atomically through a set of points of the coordinated system. The main motivation behind behavioural automata is to describe the synchronous semantics underlying Dreams [18], a prototype distributed framework briefly discussed in §5.2 that stands out by the decoupled execution of Reo-like coordination models in a concurrent setting. Dreams improves the performance and scalability of previous attempts to implement similar coordination models. Our automata model captures exactly the features implemented by Dreams.

Behavioural automata assume certain properties over their labels, such as the existence of a composition operator, and use a predicate associated to each of its states that is needed to guide the composition of automata. Different choices for the composition operator of labels and the predicates yield different coordination semantics. We instantiate our automata with the semantics for Reo and Linda coordination models, but other semantic models can also be captured by our automata [18]. We do not instantiate behavioural automata with Esterel as the propagation of synchrony in this language differs from our dataflow-driven approach [3].

Summarising, the main contributions of this paper are:

- a *unified* automata model that captures dataflow-oriented synchronous coordination models;

M.R. Mousavi and A. Ravara: FOCLASA 2011
EPTCS 58, 2011, pp. 65–79, doi:10.4204/EPTCS.58.5

- the introduction of *concurrency predicates*, increasing the expressiveness of the model when dealing with composed automata; and
- the *decoupling of execution* of a distributed implementation based on our automata model, by avoiding unnecessary synchronisation of actions whenever possible.

Each behavioural automaton has a concurrenty predicate that indicates, for each state, which labels of other automata require synchronisation. When composing two automata, labels must be either composed in a pairwise fashion, or they can be performed independently when the concurrency predicate does not require synchronisation. We exploit how to use concurrency predicates to distinguish transitions of a composed automaton that originate from all intermediate automata, or from only a subset of them. We also illustrate how to obtain more complex notions of coordination by increasing the complexity of concurrency predicates.

This paper is organised as follows. We introduce behavioural automata in §2. We then encode Reo as behavioural automata in §3 and Linda as behavioural automata in §4. In §5 we motivate the need for concurrency predicates, both from a theoretical and practical perspectives. We conclude in §6.

2 A stepwise coordination model

In this section we present an automata model, dubbed *behavioural automata*. This model represents our view of a dataflow-driven coordination system, following the categorisation of Arbab [3]. Each transition in an automaton represents the *atomic* execution of a number of actions by the coordination system. We describe the behaviour of a system by the *composition* of the behaviour of its sub-systems running concurrently, each with its own automaton. Furthermore, we allow the *data values* exchanged over the coordination layer to influence the choice of how components communicate with each other as well. We borrow ideas from the Tile model [14, 4], distinguishing evolution in time (execution of the coordination system) and evolution in space (composition of coordination systems). Behavioural automata can be built by *composing* more primitive behavioural automata, and each transition of an automaton denotes a round of the coordination process, where data flow *atomically* through zero or more ports of the system.

We use behavioural automata to give semantics to Reo, based on the constraint automata model [6], and to (distributed) Linda [15]. Each label of an automaton describes which ports should have dataflow, and what data should be flowing in each port. We write $\mathbb{P}$ to denote a global set of ports, $\mathsf{L}[P]$ to denote the set of all labels over the ports $P \subseteq \mathbb{P}$, and $\mathbb{D}$ to denote a global set of data values. We associate a predicate over labels to each state q of an automaton, referred to as $\mathcal{C}(q)$. These predicates are used to guide the composition of behavioural automata.

Definition 1 (Behavioural automata) *A* behavioural automaton *of a system over a set of ports $P \subseteq \mathbb{P}$ is a labelled transition system $\langle Q, \mathsf{L}[P], \rightarrow, \mathcal{C} \rangle$, where $\mathsf{L}[P]$ is the set of labels over P, $\rightarrow \subseteq Q \times \mathsf{L}[P] \times Q$ is the transition relation, and $\mathcal{C} : Q \rightarrow \mathbf{2}^{\mathsf{L}[P]}$ is a predicate over states and labels, called* concurrency predicate*, regarded as a function that maps states to sets of labels.*

The key ingredients of behavioural automata are *atomic steps* and *concurrency predicates*. Each label of a behavioural automaton has an associated atomic step, which captures aspects such as the ports that have flow and the data flowing through them, and concurrency predicate describe, for each state, which labels from other automata running concurrently require synchronisation.

Example 1 (Alternating coordinator) *We present the* alternating coordinator *(AC) in Figure 1. It receives data from two data writers W_1 and W_2, and sends data to a reader R. The components W_1, W_2*

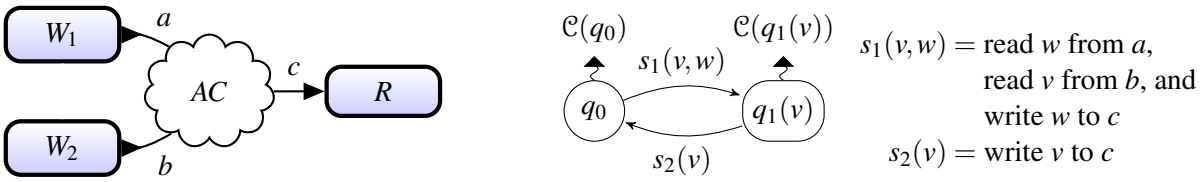


Figure 1: Alternating coordinator (left), and its behavioural automaton (right).

and R are connected, respectively, to the ports a, b and c of the alternating coordinator. The alternating coordinator describes how data can flow between the components, and coordination is specified by the behavioural automaton depicted on the right side of Figure 1. Each transition of this automaton represents a possible step in time of the coordinator AC, describing how the ports a, b, and c can have dataflow. Initially, the coordinator is in state q_0, where the only possible action is reading a value w from W_1 through a and sending it to the reader R through c, while reading and buffering a value v sent by W_2 through b. Note that if only one of the writers can produce data, the step cannot be taken, and the system cannot evolve. In the next state, q_1, the only possible step is to send the value v to the reader R, returning to state q_0. The arrows between states represent the transition relation $\rightarrow$. In both states there is the possibility of allowing the concurrent execution of other automata, provided that this execution does not interfere with the current behaviour. The conditions of when other automata can execute concurrently are captured by the concurrency predicate $\mathcal{C}$, depicted by squiggly arrows ($\rightsquigarrow$) from each state.

2.1 Labels, atomic steps and concurrent predicates

Labels over a set of ports P are elements from a set $\mathsf{L}[P]$ with some properties required for composition, which we will introduce later. Furthermore, a label $\ell \in \mathsf{L}[P]$ can be restricted to a smaller set of ports $P' \subseteq P$, written $\ell^{(P')}$. We require each label $\ell \in \mathsf{L}[\mathbb{P}]$ to have an associated description of where and which data flow in the connector, written as $\alpha(\ell)$, and captured by the notion of *atomic step*.

Definition 2 (Atomic step) *An* atomic step *over the alphabet $P \subseteq \mathbb{P}$ is a tuple $\langle P, F, IP, OP, data \rangle$ where:*
$F \subseteq P \qquad IP \subseteq F \qquad OP \subseteq F \qquad IP \cap OP = \emptyset \qquad$ *and* $\qquad data : (IP \cup OP) \rightarrow \mathbb{D}$.

We write $\mathsf{AS}[P]$ to denote the set of all atomic steps over the ports in P. P is a set of ports in the scope of the atomic step. The flow set F is the set of ports that *synchronise*, i.e., that have data flowing in the same atomic step. The sets IP and OP represent the input and output ports of the atomic step that have dataflow, and whose values are considered to be relevant when performing a step. Ports in F but not in IP or OP are ports with dataflow, but whose data values are not relevant, that is, they are used only for imposing synchronisation constraints. The data values that flow through the relevant ports are given by the data function *data*. We distinguish IP and OP to capture data dependencies.

Concurrency predicates are used to compose behavioural automata. When composing two automata a_1 and a_2, if a_1 has ports P_1, has the concurrency predicate $\mathcal{C}_1$, and is in state q_1, then $\ell_2^{(P_1)} \in \mathcal{C}_1(q_1)$ means that a_2 can perform ℓ_2 only when composed with a transition from a_1, otherwise a_2 can perform ℓ_2 without requiring a_1 to perform a transition.[1] When clear from context, we omit the restriction and write $\ell_2 \in \mathcal{C}_1(q_1)$ instead of $\ell_2^{(P_1)} \in \mathcal{C}_1(q_1)$. We give a possible definition for concurrency predicates based

[1] We present a variation of the original definition of concurrency predicates [18] to make the decision of belonging to a concurrent predicate more local.

solely on the set of known ports.[2] Given a connector with known ports P_0, the concurrency predicate of every state is given by the predicate

$$cp(P_0) \;=\; \{\ell \mid \alpha(\ell) = \langle P,F,IP,OP,data\rangle, P_0 \cap F \neq \emptyset\}. \tag{1}$$

Example 2 *We define the atomic steps and concurrency predicates from Example 1 as follows.*

$$\begin{array}{llll} \alpha(s_1(v,w)) &= \langle P,abc,ab,c,\{a,b,c\mapsto w,v,w\}\rangle & \mathcal{C}(q_1(v)) &= cp(P) \\ \alpha(s_2(v)) &= \langle P,\quad c,\ \emptyset,c,\{c\mapsto v\}\rangle & \mathcal{C}(q_0) &= cp(P) \end{array}$$

For simplicity, we write $a_1 \ldots a_n$ instead of $\{a_1,\ldots,a_n\}$ when the intended notion of set is clear from the context. The alphabet P is $\{a,b,c\}$, and the concurrency predicates allow only steps where none of the known ports has flow.

2.2 Composition of behavioural automata

To compose behavioural automata we require labels to be elements of a partial monoid $\langle \mathsf{L}, \otimes\rangle$, that is, (1) there must be a commutative operator $\otimes : \mathsf{L}^2 \rightharpoonup \mathsf{L}$ for labels, and (2) the composition of two labels can be undefined, meaning that they are incompatible. For technical convenience, we require $\otimes$ to be associative and to have an identity element. The atomic step $\langle P,F,IP,OP,data\rangle$ of a composed label $\ell_1 \otimes \ell_2$ must obey the following conditions, where, for every label ℓ_1 or ℓ_2, $\alpha(\ell_i) = \langle P_i,F_i,IP_i,OP_i,data_i\rangle$.

$$\begin{array}{lll} P \subseteq P_1 \cup P_2 & IP \subseteq (IP_1 \cup IP_2)\backslash(OP_1 \cup OP_2) & data_1 \frown data_2 \\ F \subseteq F_1 \cup F_2 & OP \subseteq OP_1 \cup OP_2 & data = data_1 \cup data_2 \end{array}$$

The atomic step of a label ℓ is represented by $\alpha(\ell)$. The notation $m_1 \frown m_2$ represents that the values of the common domain of mappings m_1 and m_2 match. The requirements on the sets *IP* and *OP* reflect that when composing two atomic steps, the input ports that have an associated output port are no longer treated as input ports (since the dependencies have been met), and the output ports are combined. The intuition behind the removal of input ports that match an output port is the preservation of the semantics of Reo: multiple connections to an output port replicate data, but multiple connections to input data require the merging of data from a single source.

We now describe the composition of behavioural automata based on the operator $\otimes$ and on concurrency predicates. This composition mimics the composition of existing Reo models [6, 11, 8].

Definition 3 (Product of behavioural automata) *The product of two behavioural automata $b_1 = \langle Q_1, \mathsf{L}[P_1], \rightarrow_1, \mathcal{C}_1\rangle$ and $b_2 = \langle Q_2, \mathsf{L}[P_2], \rightarrow_2, \mathcal{C}_2\rangle$, denoted by $b_1 \bowtie b_2$, is the behavioural automaton $\langle Q_1 \times Q_2, \mathsf{L}[P_1 \cup P_2], \rightarrow, \mathcal{C}\rangle$, where $\rightarrow$ and $\mathcal{C}$ are defined as follows:*

$$\rightarrow \;=\; \{\langle (p,q),\ell,(p',q')\rangle \mid p \xrightarrow{\ell_1}_1 p', q \xrightarrow{\ell_2}_2 q', \ell = \ell_1 \otimes \ell_2, \ell \neq \bot\} \cup \tag{2}$$

$$\{\langle (p,q),\ell,(p',q)\rangle \mid p \xrightarrow{\ell}_1 p', \ell^{(P_2)} \notin \mathcal{C}_2(q)\} \cup \{\langle (p,q),\ell,(p,q')\rangle \mid q \xrightarrow{\ell}_2 q', \ell^{(P_1)} \notin \mathcal{C}_1(p)\} \tag{3}$$

$$\mathcal{C}(p,q) \;=\; \mathcal{C}_1(p) \cup \mathcal{C}_2(q) \quad \textit{for} \quad p \in Q_1, q \in Q_2. \tag{4}$$

Case (3) covers the situation where one of the behavioural automata performs a step admitted by the concurrency predicate of the other, and case (4) defines the composition of two concurrency predicates.

In practice, our framework based on behavioural automata, briefly described in §5.2, uses a symbolic representation for data values assuming that variables can be instantiated after selecting the transition. This suggests the use of a late-semantics for data-dependencies. Our approach to compose labels resembles Milner's synchronous product in SCCS [17], with the main difference that the product of behavioural automata do not require the all labels to be synchronised. The product of labels from two behavioural automata can be undefined, and labels can avoid synchronisation when the concurrency predicate holds.

[2] Other semantic models may require more complex concurrency predicates. For example, the concurrency predicates for the Reo automata model [8] depend on the current state (Section 3.6.2 of [18]).

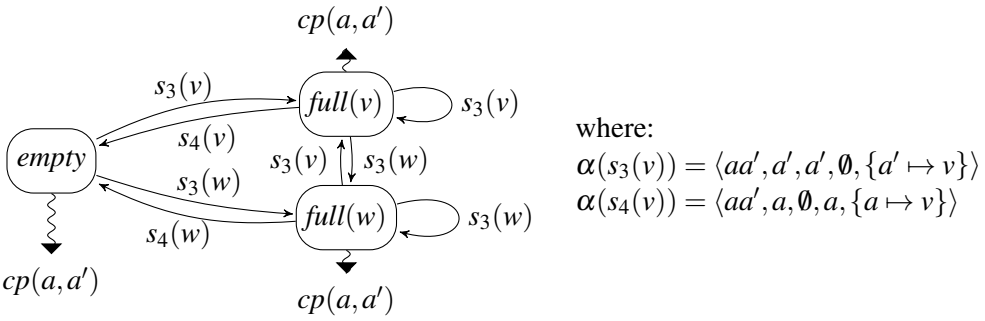


Figure 2: Behavioural automaton of the lossy-FIFO connector.

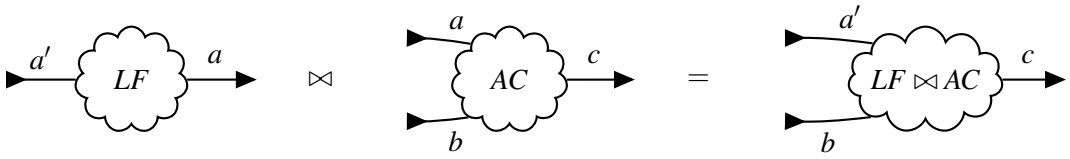


Figure 3: The sink and source ports of LF, AC, and their composition.

2.3 Example: lossy alternator

Recall the behavioural automaton AC of the alternating coordinator, illustrated in Figure 1. Data is received always via ports a and b simultaneously, and sent via port c, alternating the values received from a and b. We now imagine the following scenario: the data on a becomes available always at a much faster rate than data on b. To adapt our alternating coordinator to this new scenario, we introduce a lossy-FIFO connector LF [1] and compose it with the alternating coordinator, yielding $LF \bowtie AC$.

Recall the definition of $cp : \mathbb{P} \to \mathsf{L}[\mathbb{P}]$ given by Equation (1). The behavioural automaton for the lossy-FIFO connector is depicted in Figure 2, and its atomic steps range over the ports $\{a, a'\}$, where a' is an input port and a is an output port. We depict the interface of both of these connectors on left hand side of Figure 3. After combining the behavioural automata of the two connectors, they become connected via their shared port a. The new variation of the alternating coordinator can then be connected to data producers and consumers by using the ports a', b and c, as depicted at the right hand side of Figure 3.

Intuitively, the lossy-FIFO connector receives data a' and buffers its value before sending it through a. When the buffer is full data received from a' replaces the content of the buffer. The connector resulting from the composition $LF \bowtie AC$ is formalised in Table 1 and in Figure 4. The flow sets of the labels $s_1(v,w)$, $s_2(v)$, $s_3(v)$ and $s_4(v)$ are, respectively, abc, c, a', and $a'a$, and the set of known ports is $P = \{a', a, b, c\}$. Let $\mathcal{C}_{LF}$ and $\mathcal{C}_{AC}$ be the concurrency predicates of LF and AC. The concurrency predicate $\mathcal{C}_{LF \bowtie AC}$ for $LF \bowtie AC$ results from the union of the predicates of the states of each behavioural automaton, and corresponds precisely to the concurrency predicate that maps each state to $cp(a', a, b, c)$. The name of each state in $LF \bowtie AC$ is obtained by pairing names of a state from LF and a state from AC. Some states and transitions are coloured in grey with their labels omitted to avoid cluttering the diagram.

From the diagram it is clear that some transitions originate only from the LF or the AC connector, while others result from the composition via the operator $\otimes$. The transitions $s_2(v)$ and $s_3(w)$ can be per-

$\otimes$	$s_1(u,v)$	$s_2(w)$
$s_3(y)$	$\bot$	$\langle P, a'c, a', c, \{a', c \mapsto y, w\}\rangle$
$s_4(z)$	$\bot$ (for $z \neq v$)	$\bot$
$s_4(v)$	$\langle P, abc, ab, c, \{a, b, c \mapsto v, u, v\}\rangle$	$\bot$

LF	$\mathcal{C}_{LF}(empty)$	$\mathcal{C}_{LF}(full(v'))$
$s_1(v,w)$	*true*	*true*
$s_2(v)$	*false*	*false*

AC	$\mathcal{C}_{AC}(q_0)$	$\mathcal{C}_{AC}(q_1(v'))$
$s_3(v)$	*false*	*false*
$s_4(v)$	*true*	*true*

Table 1: Atomic steps of the composition of labels from LF and AC (left), and verification of the concurrency predicate for each label (right).

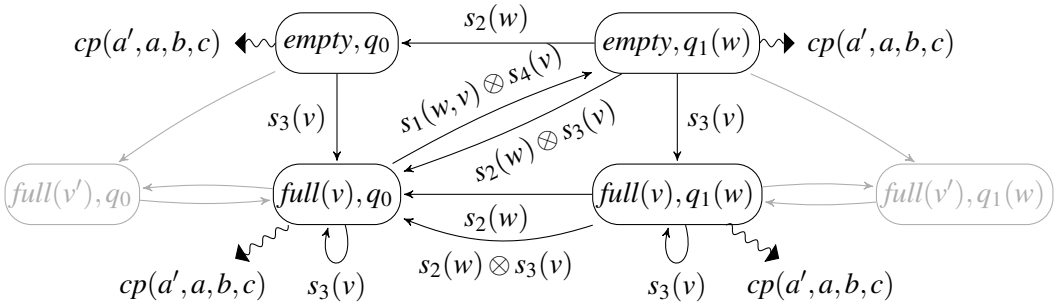


Figure 4: Behavioural automaton for the composition of LF and AC.

formed simultaneously or interleaved; simultaneously because $s_2(v) \otimes s_3(w)$ is defined, and interleaved because $\mathcal{C}_{LF}$ never contains $s_2(v)$ and $\mathcal{C}_{AC}$ never contains $s_3(w)$. The possible execution scenarios of these atomic steps follow our intuition that steps 'approved' by concurrency predicates can be performed independently. The steps $s_1(u,v)$ and $s_4(w)$ can be taken only when composed.

2.4 Locality

We introduce the notion of locality as a property of behavioural automata that guarantees the absence of certain labels in the concurrency predicates of *independent* behavioural automata, that is, in automata without shared ports.

Definition 4 (Locality of behavioural automata) *A behavioural automaton $b = \langle Q, \mathsf{L}[P], \rightarrow, \mathcal{C}\rangle$ obeys the* locality property *if, for any port set P' such that $P \cap P' = \emptyset$, $\forall \ell \in \mathsf{L}[P'] \cdot \forall q \in Q \cdot \ell^{(P)} \notin \mathcal{C}(q)$.*

Any two behavioural automata with disjoint port sets that obey the locality property can therefore evolve concurrently in an interleaved fashion. Let $b = b_1 \bowtie b_2$ be a behavioural automaton and ℓ a label from b_1. We say ℓ is a *local step* of b if $(q_1,q_2) \xrightarrow{\ell} (q_1',q_2')$ is a transition of b and either $q_1 \xrightarrow{\ell}_1 q_1'$, $q_2 = q_2'$, and $\ell \in \mathcal{C}_2(q_2)$; or $q_2 \xrightarrow{\ell}_2 q_2'$, $q_1 = q_1'$, and $\ell \in \mathcal{C}_1(q_1)$. In the behavioural automaton exemplified in Figure 4, the local steps are exactly the transitions labelled by the steps $s_2(w)$ and $s_3(v)$.

Proposition 1 *Let $b = b_1 \bowtie b_2 \bowtie b_3$ be a behavioural automaton where $b_i = \langle Q_i, \mathsf{L}[P_i], \rightarrow_i, \mathcal{C}_i \rangle$, for $i \in 1..3$, and assume the locality property from Definition 4 holds for b_1, b_2 and b_3. Suppose $P_1 \cap P_3 = \emptyset$. Then, for any step $\ell_1^{(P_1)} \in \mathsf{L}[P_1]$ performed by b_1 and $q_2 \in Q_2$, if $\ell_1^{(P_2)} \notin \mathcal{C}_2(q_2)$ then ℓ_1 is a local step of b.*

Proof. Observe that $\bowtie$ is associative, up to the state names, because the composition of labels $\otimes$ is associative. From $P_1 \cap P_3 = \emptyset$, $\ell_1 \in \mathsf{L}[P_1]$, and from the locality property in Definition 4 we conclude that $\forall q \in Q_3 \cdot \ell_1^{(P_3)} \notin \mathcal{C}_3(q)$. Therefore, for any state $q_3 \in Q_3$ and for a state $q_2 \in Q_2$ such that $\ell_1^{(P_2)} \notin \mathcal{C}_2(q_2)$, we have that $\ell_1^{(P_2)} \notin \mathcal{C}_2(q_2) \cup \mathcal{C}_3(q_3)$. We conclude that $\ell_1^{(P_2 \cup P_3)} \notin \mathcal{C}'$, where $\mathcal{C}'$ is the concurrency predicate of $b_2 \bowtie b_3$, hence a local step of b. □

If the locality property holds for each behavioural automata b_i in a composed system $b = b_1 \bowtie \cdots \bowtie b_n$, then, using Proposition 1, we can infer wether atomic steps from b_i are local steps of b based only on the concurrency predicates of its *neighbour automata*, i.e., the automata that share ports with b_i.

2.5 Concrete behavioural automata

A behavioural automaton is an abstraction of concrete coordination models that focuses on aspects relevant to the execution of the coordination model. As we will argue, Reo and Linda can be cast in our framework of behavioural automata. Therefore, both Reo and Linda coordination models can be seen as specific instances of the stepwise model described above. For a concrete coordination model to fit into the stepwise model, we need to define: (1) labels in the concrete model; (2) the encoding α of labels into atomic steps; (3) composition of labels; and (4) concurrency predicates.

We start by encoding the constraint automata semantics of Reo as behavioural automata. Later, because of its relevance in the coordination community as one of the first coordination languages, we also encode Linda as a behavioural automaton. Other coordination models have also been encoded as behavioural automata in Proença's Ph.D. thesis [18].

3 Encoding Reo

Reo [1, 2] is presented as a channel-based coordination language wherein component connectors are compositionally built out of an open set of *primitive connectors*, also called primitives. Channels are primitives with two ends. Existing tools for Reo include an editor, an animation generator, model checkers, editors of Reo-specific automata, QoS modelling and analysis tools, and a code generator [5, 16].

The behaviour of each primitive depends upon its current state.[3] The semantics of a connector is described as a collection of possible steps for each state, and we call the change of state of the connector triggered by one of these steps a *round*. At each round some of the ends of a connector are synchronised, i.e., only certain combinations of synchronous dataflow through its ends are possible. Dataflow on a primitive's end occurs when a single datum is passed through that end. Within any round dataflow may occur on some number of ends. Communication with a primitive connector occurs through its ports, called *ends*. Primitives consume data through their *source ends*, and produce data through their *sink ends*. Connectors are formed by plugging the ends of primitives together in a one-to-one fashion to form *nodes*. A node is a logical place consisting of a sink end, a source end, or both a sink and a source end.[4]

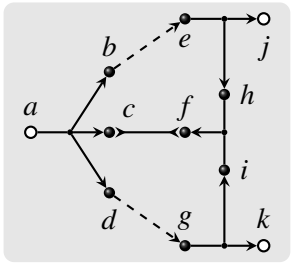
We now give an informal description of some of the most commonly used Reo primitives. Note that, for all of these primitives, no dataflow is one of the behavioural possibilities. The Sync channel

[3] Note that most Reo primitives presented here have a single state.

[4] Generalised nodes with multiple sink and source ends can be defined by combining binary mergers and replicators [6, 11].

(⟶) sends data synchronously from its source to its sink end. The LossySync channel (⇢) differs from the Sync channel only because it can non-deterministically lose data received from its source port. The SyncDrain (⟷) has two source ends, and requires both ends to have dataflow synchronously, or no dataflow is possible. The FIFO_1 channel (⟶□⟶) has two possible states: empty or full. When empty, it can receive a data item from its source end, changing its state to full. When full, it can only send the data item received previously, changing its state back to empty. Finally, a replicator (⟶<) replicates data synchronously to all of its sink ends, while a merger (>⟶) copies data atomically from exactly one of its sink ends to its source end.

Example 3 *The connector on the right is an exclusive router built by composing two LossySync channels (b-e and d-g), one SyncDrain (c-f), one Merger (h-i-f), and three Replicators (a-b-c-d, e-j-h and g-i-k). The constraints of these primitives can be combined to give the following two behavioural possibilities (plus the no-flow-everywhere possibility):*



- *ends* $\{a,b,c,d,e,f,h,j\}$ *synchronise and a data item flows from a to* j,
- *ends* $\{a,b,c,d,f,g,i,k\}$ *synchronise and a data item flows from a to* k.

The merger makes a non-deterministic choice whenever both behaviours are possible. Data can never flow from a to both j and k, as this is excluded by the behavioural constraints of the Merger h-i-f.

3.1 Constraint automata

We briefly describe constraint automata [6]. Constraint automata use a finite set of port names $\mathcal{N} = \{x_1,\ldots,x_n\}$, where x_i is the i-th port of a connector. When clear from the context, we write xyz instead of $\{x,y,z\}$ to enhance readability. We write $\widehat{x_i}$ to represent the variable that holds the data value flowing through the port x_i, and use $\widehat{\mathcal{N}}$ to denote the set of data variables $\{\widehat{x_1},\ldots,\widehat{x_n}\}$, for each $x_i \in \mathcal{N}$. We define DC_X for each $X \subseteq \mathcal{N}$ to be a set of data constraints over the variables in $\widehat{X}$, where the underlying data domain is a finite set $\mathbb{D}$. Data constraints in $DC_{\mathcal{N}}$ can be viewed as a symbolic representation of sets of data-assignments, and are generated by the following grammar:

$$g \;::=\; \texttt{tt} \;\mid\; \widehat{x} = d \;\mid\; g_1 \vee g_2 \;\mid\; \neg g$$

where $x \in \mathcal{N}$ and $d \in \mathbb{D}$. The other logical connectives can be encoded as usual. We use the notation $\widehat{a} = \widehat{b}$ as a shorthand for the constraint $(\widehat{a} = d_1 \wedge \widehat{b} = d_1) \vee \ldots \vee (\widehat{a} = d_n \wedge \widehat{b} = d_n)$, with $\mathbb{D} = \{d_1,\ldots,d_n\}$.

Definition 5 (Constraint Automaton [6]) *A* constraint automaton *(over the finite data domain $\mathbb{D}$) is a tuple $\mathcal{A} = \langle Q, \mathcal{N}, \rightarrow, Q_0\rangle$, where Q is a set of states, $\mathcal{N}$ is a finite set of port names, $\rightarrow$ is a subset of $Q \times \mathbf{2}^{\mathcal{N}} \times DC_{\mathcal{N}} \times Q$, called the transition relation of $\mathcal{A}$, and $Q_0 \subseteq Q$ is the set of initial states.*

We write $q \xrightarrow{X|g} p$ instead of $(q,X,g,p) \in \rightarrow$. For every transition $q \xrightarrow{X|g} p$, we require that g, the guard, is a DC_X-constraint. For every state $q \in Q$, there is a transition $q \xrightarrow{\emptyset|\texttt{tt}} q$.

We define $\mathsf{CAS} \subseteq \mathbf{2}^{\mathcal{N}} \times DC_{\mathcal{N}}$ to be the set of solutions for all possible labels of the transitions of constraint automata. That is, $X|g \in \mathsf{CAS}$ if $X = \{x_1,\ldots,x_n\}$, $g = \bigwedge \widehat{x_i} = v_i$, where $v_i \in \mathbb{D}$, and there is a transition $q \xrightarrow{X|g'} q'$ such that g satisfies g'. We call each $s \in \mathsf{CAS}$ a constraint automaton step. Firing a transition $q \xrightarrow{X|g} p$ is interpreted as having dataflow at all the ports in X, while excluding flow at ports in $\mathcal{N} \setminus X$, when the automaton is in the state q. The data flowing through the ports X must satisfy the constraint g, and the automaton evolves to the state p. Figure 5 exemplifies the constraint automata for three Reo channels. We do not define here the composition of constraint automata, but encode labels of constraint automata as labels of behavioural automata, whose composition has been defined in §2.2.

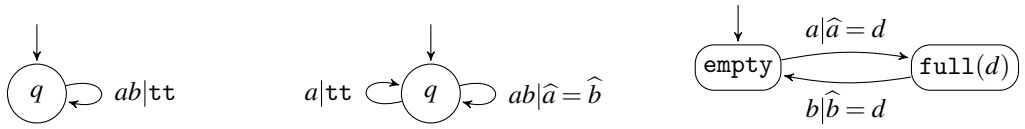


Figure 5: From left to right, constraint automata for the SyncDrain, LossySync and FIFO$_1$ channels.

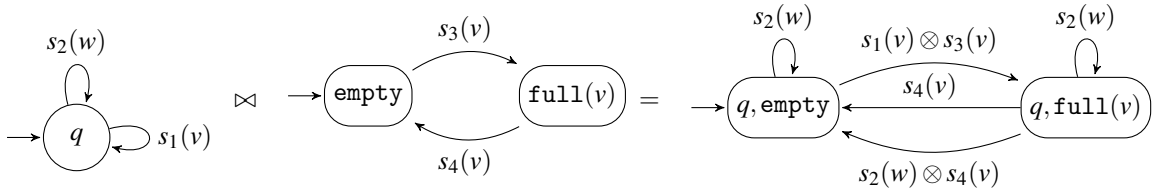


Figure 6: Composition of $[\![\mathcal{A}_L]\!]_{\mathsf{CA}}$ and $[\![\mathcal{A}_F]\!]_{\mathsf{CA}}$, for any $v, w \in \mathbb{D}$.

3.2 Constraint automata as behavioural automata

The CA model assumes a finite data domain $\mathbb{D}$, and that data constraints such as $\mathtt{tt}$, $\widehat{a} \neq d$, or $\widehat{a} = \widehat{b}$ stand for simpler data constraints that use $\widehat{a} = d$ and the operators $\wedge$ and $\vee$.

The encoding of the constraint automaton $\mathcal{A} = \langle Q, \mathcal{N}, \rightarrow_{\mathsf{CA}}, Q_0 \rangle$ is the behavioural automaton

$$[\![\mathcal{A}]\!]_{\mathsf{CA}} = \langle Q, \mathsf{L}[\mathcal{N}], \rightarrow_{\mathsf{BA}}, \mathcal{C} \rangle$$

with $\mathsf{L}[\mathcal{N}]$, $\rightarrow_{\mathsf{BA}}$, $\mathcal{C}$, and the composition of labels defined as follows:

- $\mathsf{L} = \mathsf{CAS}$, and α is defined as: $\alpha(X | \bigwedge_{i=1}^{n} \widehat{x_i} = d_i) \;=\; \langle \mathcal{N}, X, \emptyset, X, \{x_i \mapsto d_i\}_{i=1}^{n} \rangle$.
- We have $q \xrightarrow{X|g}_{\mathsf{BA}} q'$ for $X|g \in \mathsf{L}[\mathcal{N}]$ if $q \xrightarrow{X|g'}_{\mathsf{CA}} q'$ and g satisfies g'.
- Let $cas_i = X_i | g_i$ be a solution for a label in a constraint automaton with ports $\mathcal{N}_i$, for $i \in 1..2$. Then

$$cas_1 \otimes cas_2 = \begin{cases} (X_1 \cup X_2) | (g_1 \wedge g_2) & \text{if } X_1 \cap \mathcal{N}_2 = X_2 \cap \mathcal{N}_1 \;\wedge\; g_1 \frown g_2 \\ \bot & \text{otherwise} \end{cases}$$

  where $g_1 \frown g_2$ if for any port $x \in X_1 \cap X_2$ and for any $d \in \mathbb{D}$, $x = d$ satisfies g_1 iff $x = d$ satisfies g_2.
- $\mathcal{C}(q) = cp(\mathcal{N})$ for every $q \in Q$. Recall that $cp(\mathcal{N}) = \{\ell \mid \alpha(\ell) = \langle P, F, IP, OP, data \rangle, P_0 \cap F \neq \emptyset\}$.

Example 4 *Let $\mathcal{A}_L = \langle Q_L, \mathcal{N}_L, \rightarrow_1, Q_1 \rangle$ and $\mathcal{A}_F = \langle Q_F, \mathcal{N}_F, \rightarrow_2, Q_2 \rangle$ be the constraint automata of the LossySync and the FIFO$_1$ channels, depicted in Figure 5. The encoding of $\mathcal{A}_L$ into behavioural automata is $\langle Q_L, \mathsf{L}[\mathcal{N}_L], \rightarrow_L, \mathcal{C}_L \rangle$, depicted in the left hand side of Figure 6, where:*

$Q_L = \{q\}$, $\mathcal{N}_L = \{a, b\}$, $\mathcal{C}_L(q) = cp(\mathcal{N}_L)$ *for* $q \in Q_L$, $s_1(v) = ab|(\widehat{a} = v \wedge \widehat{b} = v)$, $s_2(v) = a|(\widehat{a} = v)$, *and* $\rightarrow_L = \{\langle q, s_1(v), q \rangle \mid v \in \mathbb{D}\} \cup \{\langle q, s_2(v), q \rangle \mid v \in \mathbb{D}\}$.

Similarly, the encoding of $\mathcal{A}_F$ into behavioural automata is $\langle Q_F, \mathsf{L}[\mathcal{N}_F], \rightarrow_F, \mathcal{C}_F \rangle$, also depicted in Figure 6, where:

$Q_F = \{\mathtt{empty}\} \cup \{\mathtt{full}(v) \mid v \in \mathbb{D}\}$, $\mathcal{C}_F(q) = cp(\mathcal{N}_F)$ *for* $q \in Q_F$, $\mathcal{N}_F = \{b, c\}$, $s_3(v) = b|(\widehat{b} = v)$, $s_4(v) = c|(\widehat{c} = v)$, *and* $\rightarrow_F = \{\langle \mathtt{empty}, s_3(v), \mathtt{full}(v) \rangle \mid v \in \mathbb{D}\} \cup \{\langle \mathtt{full}(v), s_4(v), \mathtt{empty} \rangle \mid v \in \mathbb{D}\}$.

The composed automaton $[\![\mathcal{A}_L]\!]_{\mathsf{CA}} \bowtie [\![\mathcal{A}_F]\!]_{\mathsf{CA}}$ is depicted in the right hand side of Figure 6, where $s_1(v) \otimes s_3(v) \;=\; ab|(\widehat{a} = v \wedge \widehat{b} = v)$ *and* $s_2(w) \otimes s_4(v) \;=\; ac|(\widehat{a} = w \wedge \widehat{c} = v)$.

The composed automata presented in Example 4, which differs from the lossy-FIFO, is equivalent to the product of the two associated constraint automata [6], with respect to the atomic steps of the labels of the automata. We expect this equivalence to hold in general, but we do not give a formal proof here.

4 Encoding Linda

Linda, introduced by Gelernter [15], is seen by many as the first coordination language. We describe it using Linda-calculus [10], and show how it can be modelled using behavioural automata. Linda is based on the *generative communication* paradigm, which describes how different processes in a distributed environment exchange data. In Linda, data objects are referred to as *tuples*, and multiple processes can communicate using a *shared tuple-space*, where they can write or read tuples.

Communication between processes and the tuple-space is done by actions executed by processes over the tuple-space. In general, these actions can occur only atomically, that is, the shared tuple-space can accept and execute an action from only one of the processes at a time. There are four possible actions, $\mathbf{out}(t)$, $\mathbf{in}(s)$, $\mathbf{rd}(s)$, and $\mathbf{eval}(P)$. The actions $\mathbf{out}(t)$ and $\mathbf{in}(s)$ write and take values to and from the shared tuple-space, respectively. The action $\mathbf{rd}(s)$ is similar to $\mathbf{in}(s)$, except that the tuple t is not removed from the tuple-space. Finally, $\mathbf{eval}(P)$ denotes the creation of a new process P that will run in parallel. We do not address $\mathbf{eval}(P)$ here because it is regarded as a reconfiguration of the system.

4.1 Linda-Calculus

We use the Linda-Calculus model, described by Goubault [12], to give a formal description of Linda, studied also by Ciancarini *et al.* [10] and others. The Linda-Calculus abstracts away from the local behaviour of processes, and focuses on the communication primitives between a *store* and a set of *processes*. Processes P are generated by the following grammar.

$$P ::= Act.P \;\mid\; X \;\mid\; \mathbf{rec}X.P \;\mid\; P[]P \;\mid\; \mathbf{end} \tag{5}$$

$$Act ::= \mathbf{out}(t) \;\mid\; \mathbf{in}(s) \;\mid\; \mathbf{rd}(s) \tag{6}$$

We denote the set of all Linda terms as Linda. The first case $Act.P$ represents the execution of a Linda action. The productions X and $\mathbf{rec}X.P$ are used to model recursive processes, where X ranges over a set of variables, and $P[]P$ is used to model local non-deterministic choice. We assume that processes do not have free variables, i.e., every X is bound by a corresponding $\mathbf{rec}X$. Finally **end** represents termination.

We model a Linda store as a multi-set of tuples from a global set *Tuple*. Each tuple consists of a sequence of parameters, which can be either a data value v from a domain $\mathbb{D}$ (an actual parameter), or a variable X (a formal parameter). We use the $\oplus$ operator to denote multi-set construction and multi-set union. For example, we write $M = t \oplus t = \{|t,t|\}$ and $M \oplus M = \{|t,t,t,t|\}$, where t is a tuple and $\{|s,t|\}$ denotes a multi-set with the elements s and t.

A *tuple-space term* M is a multi-set of processes and tuples, generated by the grammar $M ::= P \mid t \mid M \oplus M$. We adopt the approach of Goubault and provide a set of inference rules that give the operational semantics of Linda-Calculus. A relation $match \subseteq Tuple \times Tuple$ represents the matching of two tuples. $(s,t) \in match$ if t has only $\mathbb{D}$ values, and there is a substitution γ whose domain is the set of free variables of s, such that $t = s[\gamma]$. $u[\gamma]$ denotes the tuple or process u after replacing its free variables according to γ. We also write $\gamma = P/x$ to denote the substitution of x by the process P, and say t γ-matches s when t matches s and $t = s[\gamma]$.

Definition 6 (Semantics of Linda) *The semantics of Linda is defined by the inference rules below.*

$$\frac{M \oplus P[\mathbf{rec}X.P/X] \longrightarrow M \oplus P'}{M \oplus \mathbf{rec}X.P \longrightarrow M \oplus P'} \quad \text{(rec)}$$

$$M \oplus P[]P' \longrightarrow M \oplus P \quad \text{(left)}$$

$$M \oplus P[]P' \longrightarrow M \oplus P' \quad \text{(right)}$$

$$M \oplus \mathbf{out}(t).P \longrightarrow M \oplus P \oplus t \quad \text{(out)}$$

$$M \oplus \mathbf{rd}(s).P \oplus t \longrightarrow M \oplus P[\gamma] \oplus t \quad \textit{if } t \textit{ } \gamma\textit{-matches } s \quad \text{(rd)}$$

$$M \oplus \mathbf{in}(s).P \oplus t \longrightarrow M \oplus P[\gamma] \quad \textit{if } t \textit{ } \gamma\textit{-matches } s \quad \text{(in)}$$

$$M \oplus \mathbf{end} \longrightarrow M \quad \text{(end)}$$

Example 5 *The following sequence of transitions illustrates the sending of data between two processes. The labels on the arrows contain the names of the rules applied in each transition of Linda-Calculus. We use the notation $P(x)$ as syntactic sugar to denote a process P where the variable x occurs freely.*

$$\begin{array}{l} \quad \mathbf{rd}(42,x).P(x) \oplus \mathbf{out}(42,43).\mathbf{end} \oplus \mathbf{in}(42,x).P'(x) \\ \xrightarrow{(out)} \mathbf{rd}(42,x).P(x) \oplus \mathbf{end} \oplus \mathbf{in}(42,x).P'(x) \oplus \langle 42,43\rangle \xrightarrow{(end)} \mathbf{rd}(42,x).P(x) \oplus \mathbf{in}(42,x).P'(x) \oplus \langle 42,43\rangle \\ \xrightarrow{(rd)} P(43) \oplus \mathbf{in}(42,x).P'(x) \oplus \langle 42,43\rangle \qquad\qquad \xrightarrow{(in)} P(43) \oplus P'(43) \end{array}$$

4.2 Linda-calculus as behavioural automata

We define an encoding function $[\![\cdot]\!]_{\mathsf{Linda}} : \mathsf{Linda} \to \mathsf{BA}$, from Linda tuple-space terms to behavioural automata. Furthermore, we define the composition of atomic steps that preserve this semantics. We encode each Linda process P as a behavioural automaton, and we create a special behavioural automaton that describes the multi-set of available tuples.

Let $\overline{Act} = \{\overline{a} \mid a \in Act\}$ and $\tau Act = \{\tau_a \mid a \in Act\}$. A port $\overline{a}$ is regarded as a dual port of a, and flow of data on a port τ_a represents the flow on the ports a and $\overline{a}$ simultaneously. The intuition is that the encoding of processes yields behavioural automata whose ports are actions in Act; the encoding of tuples yield behavioural automata whose ports are *dual* actions in $\overline{Act}$; and the composition forces actions and dual actions to synchronise, i.e., to occur simultaneously. We define the global set of ports to be $\mathbb{P} = Act \cup \overline{Act} \cup \tau Act$, and define $\overline{\overline{a}} = a$.

Let $M = P_1 \oplus \cdots \oplus P_n \oplus T$ be a tuple-space term. In turn, let $T = t_1 \oplus \cdots \oplus t_m$ and $m \geq 0$. We define the encoding of M into a behavioural automaton as follows.

$$[\![M]\!]_{\mathsf{Linda}} = [\![P_1]\!]_{\mathsf{Linda}} \bowtie \cdots \bowtie [\![P_n]\!]_{\mathsf{Linda}} \bowtie [\![T]\!]_{\mathsf{Linda}}$$

Hence, encoding M boils down to encoding Linda processes P_i and the Linda tuple-space T into different behavioural automaton. In both encodings of components and Linda tuple-spaces we define labels L as ports, that is, $\mathsf{L} = \mathbb{P} = Act \cup \overline{Act} \cup \tau Act$, and its encoding as atomic steps by the function α defined below.

$$\alpha(a) = \begin{cases} \langle \mathbb{P}, \{a, \tau_{act}\}, \emptyset, \emptyset, \emptyset \rangle & \text{if } a \in Act \cup \overline{Act}, \{act\} = \{a, \overline{a}\} \cap Act \\ \langle \mathbb{P}, \{a\}, \emptyset, \emptyset, \emptyset \rangle & \text{if } a \in \tau Act \end{cases}$$

The composition of two labels $a_1, a_2 \in \mathsf{L}$ is defined as follows.

$$a_1 \otimes a_2 = \begin{cases} \tau_{act} & \text{if } a_1 \notin \tau Act \wedge a_2 \notin \tau Act \wedge a_1 = \overline{a}_2 \\ \bot & \text{otherwise,} \end{cases}$$

where $\{act\} = \{a_1, a_2\} \cap Act$. The tuple-space is used to enforce every action a performed by a process to synchronise with the corresponding action $\overline{a}$ in the tuple-space encoded as a behavioural automaton. The definition of $\otimes$ replaces every pair of ports with dataflow a and $\overline{a}$ by a new port with dataflow in τ_{act}.

We encode a Linda process P as $[\![P]\!]_{\mathsf{Linda}} = \langle Q_P, \mathsf{L}, \rightarrow_P, \mathcal{C} \rangle$, with components as defined below.

- The set of states Q_P is given by $Q_P = reach(P)$, where

$$\begin{array}{rcl} reach(\mathbf{out}(t).P) & = & \{\mathbf{out}(t).P\} \cup reach(P) \\ reach(\mathbf{rd}(s).P) & = & \{\mathbf{rd}(t).P\} \cup (\bigcup \{reach(P[\gamma]) \mid s \ \gamma\text{-matches } t\}) \\ reach(\mathbf{in}(s).P) & = & \{\mathbf{in}(t).P\} \cup (\bigcup \{reach(P[\gamma]) \mid s \ \gamma\text{-matches } t\}) \\ reach(P \boxbar P') & = & \{P \boxbar P'\} \cup reach(P) \cup reach(P') \\ reach(\mathbf{end}) & = & \{\mathbf{end}\} \end{array}$$

- The transition relation $\rightarrow_P$ is given by the following conditions.

$$\begin{array}{rcll} \mathbf{out}(t).P' & \xrightarrow{\mathbf{out}(t)} & P' & \text{if} \quad t \in \mathit{Tuple} \\ \mathbf{rd}(s).P' & \xrightarrow{\mathbf{rd}(t)} & P'[\gamma] & \text{if} \quad s\ \gamma\text{-matches } t \\ \mathbf{in}(s).P' & \xrightarrow{\mathbf{in}(t)} & P'[\gamma] & \text{if} \quad s\ \gamma\text{-matches } t \end{array} \qquad \begin{array}{l} P_1 \,[\!]\, P_2 \xrightarrow{s} P_1' \quad \text{if} \quad P_1 \xrightarrow{s} P_1' \\ P_1 \,[\!]\, P_2 \xrightarrow{s} P_2' \quad \text{if} \quad P_2 \xrightarrow{s} P_2' \end{array}$$

- $\mathcal{C}(q) = \mathit{Act} \cup \overline{\mathit{Act}}$ for every state q.

We now encode a Linda tuple-space T as $[\![T]\!]_{\mathsf{Linda}} = \langle Q_T, \mathsf{L}, \rightarrow_T, \mathcal{C} \rangle$ with components as defined below.

- $Q_T = \mathbf{2}^{\mathcal{M}(\mathit{Tuple})}$, where $\mathcal{M}(X)$ is a multi-set over the set X.
- The transition relation $\rightarrow_T$ is given by the following conditions:
  $M \xrightarrow{\overline{\mathbf{out}(t)}} M \oplus t$ *if* $t \in \mathit{Tuple}$, $\quad t \oplus M \xrightarrow{\overline{\mathbf{rd}(s)}} t \oplus M$ *if* s *matches* t, and $\quad t \oplus M \xrightarrow{\overline{\mathbf{in}(s)}} M$ *if* s *matches* t.
- $\mathcal{C}(q) = \mathit{Act} \cup \overline{\mathit{Act}}$ for every state q, as in the encoding of Linda processes.

Note that the input and output ports of the atomic steps obtained with α, introduced in §2.1, are always the empty set, that is, the data value flowing through the ports is not relevant, since the name of the port uniquely identifies the data. Alternative approaches to implement the encoding into behavioural automata that use the data values are also possible, but less transparent.

Example 6 *Recall the example presented in the end of §4.1 of a sequence of transitions of a tuple-space term in Linda-Calculus. We present below a simplified version of this example.*

$$\mathbf{rd}(42,x).P(x) \oplus \mathbf{out}(42,43).P' \xrightarrow{(out)} \mathbf{rd}(42,x).P(x) \oplus P' \oplus \langle 42,43 \rangle \xrightarrow{(rd)} P(43) \oplus P' \oplus \langle 42,43 \rangle$$

The corresponding transitions in the encoded behavioural automaton are presented below.

$$\begin{array}{ll} [\![\mathbf{rd}(42,x).P(x)]\!]_{\mathsf{Linda}} \bowtie [\![\mathbf{out}(42,43).P']\!]_{\mathsf{Linda}} \bowtie [\![\emptyset]\!]_{\mathsf{Linda}} & \xrightarrow{\tau_{\mathbf{out}(42,43)}} \\ [\![\mathbf{rd}(42,x).P(x)]\!]_{\mathsf{Linda}} \bowtie [\![P']\!] \bowtie [\![\langle 42,43 \rangle]\!] & \xrightarrow{\tau_{\mathbf{rd}(42,43)}} [\![P(43)]\!]_{\mathsf{Linda}} \bowtie [\![P']\!] \bowtie [\![\langle 42,43 \rangle]\!] \end{array}$$

Observe that we assume an initial empty tuple-space, which is encoded as $[\![\emptyset]\!]_{\mathsf{Linda}}$. A more careful analysis shows a one-to-one correspondence between the traces of the Linda-calculus term and the traces of the behavioural automaton, which we do not elaborate in this paper.

5 Exploiting concurrency predicates

We introduced a unified model for synchronous coordination that explicitly mentions concurrency predicates, which indicate which actions require synchronisation. We now exploit more complex definitions of concurrent predicates for Reo and Linda than in our previous examples, and briefly describe a practical application of behavioural automata in a distributed framework.

5.1 Complex concurrency predicates

In our examples concurrency predicates of Reo hold when some shared ports from a composed automaton have dataflow (Equation (1)), and concurrency predicates of Linda allow only a special set of actions τAct to run concurrently. We now present other concurrency predicates that capture notions such as context dependency and priority.

Reo Other semantic models for Reo, such as connector colouring [11] and Reo automata [8], capture the notion of *context dependency*, a feature missing in constraint automata. By modelling context dependency we avoid the undesired behaviour of the composed connector in Figure 6 where data is lost when the FIFO_1 buffer is empty, represented by the label $s_2(w)$.

To avoid data from being lost, we replace the LossySync channel by a context dependent LossySync channel, which is built based on the LossySync channel by replacing the label $s_2(w)$ by a label $s_2^b(w)$. This new label has the same atomic step, i.e., $\alpha(s_2(w)) = \alpha(s_2^b(w))$, but can be executed in parallel only if its neighbours require the port b to have no dataflow. This condition is enforced by adapting the definition of concurrency predicates to check wether a given set of ports Y requires synchronisation.

$$cp_{ctx}(P_0, Y) = \left\{ s^X \mid s^X \in cp(P_0) \ \ \vee \ \ X \cap Y \neq \emptyset \right\} \tag{7}$$

In our example, we avoid the losing of data by defining $\mathcal{C}(q) = cp_{ctx}(ab, \emptyset)$, $\mathcal{C}(\texttt{empty}) = cp_{ctx}(bc, b)$, and $\mathcal{C}(\texttt{full}(v)) = cp_{ctx}(ab, c)$. The label $s_2^b(w)$ is in $\mathcal{C}(\texttt{empty})$ but not in $\mathcal{C}(\texttt{full})$, i.e., $s_2^b(w)$ can be performed independently of the FIFO_1 channel only when the FIFO_1 is full. Other important details, such as the composition of labels of the form s^X, are not presented in this paper. A more precise and complete formulation can be found in Proença's Ph.D. thesis (Sections 3.6.2 and 4.4.2 of [18]).

Linda Consider now that Linda processes have a total order $\preceq$, representing a ranking among processes. When two processes can interact simultaneously with the shared tuple-space, only the higher rank should be chosen. We present only a sketch of this approach due to space limitation.

We start by tagging labels ℓ of the Linda behavioural automata with the process that executes it. For example, a label ℓ of an automaton of a process p is renamed to ℓ^p. Labels of the shared tuple-space are not changed. The composition of labels must be such that $\ell^p \otimes \overline{\ell} = \tau_\ell^p$. It is then enough to change the concurrency predicates of the automata of each process p in state q to $\mathcal{C}(q) = Act \cup \overline{Act} \cup \left\{ \tau_\ell^x \mid \tau_\ell \in \tau Act \ \wedge \ x \preceq p \ \wedge \ q \neq \textbf{end} \right\}$ and leave the concurrency predicate of the automaton of the shared tuple-space unchanged. Hence, a transition cannot be performed in parallel if it is in Act or $\overline{Act}$, or if it is a τ action from a process with lower priority and the current process is not yet stopped.

5.2 Increased scalability via decoupled execution

We use the behavioural automata model in a distributed framework, Dreams, where several independent threads run concurrently [18]. Each thread has its own behavioural automaton, and communicates only with those threads whose behavioural automata share ports with its own automata. The details regarding this tool are out of the scope this paper, but we explain how it benefits from using behavioural automata.

The diagram in Figure 7 depicts the configuration of a system in Dreams, where each cloud represents an independent thread of execution, and edges represent communication links between threads whose automata share ports. The direction of each edge only illustrates the expected direction of dataflow. For efficiency reasons, and to allow a lightweight reconfiguration, Dreams does not create the complete behavioural automaton of a connector. Instead, it collects only the behaviour of the current round.

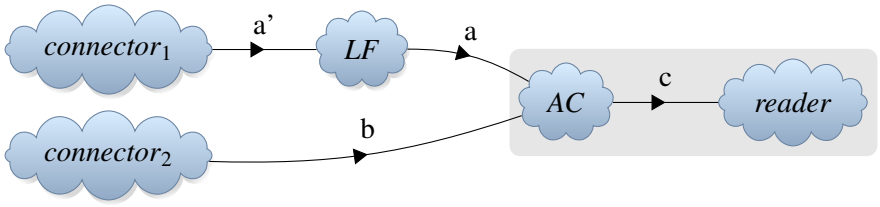


Figure 7: Configuration of a system in Dreams.

Knowing that only the labels of the automata relevant for the current round are composed, and assuming that the locality property introduced in Definition 4 holds, we can perform local steps that, as the name suggests, involve only a subpart of the system. Recall the example of the lossy alternator, presented in §2.3. The diagram in Figure 7 uses the same example, in a context where two arbitrary large connectors $connector_1$ and $connector_2$ are attached to the source of the lossy alternator, and a *reader* component is attached to the sink of the lossy alternator. Consider that the *reader* can always receive any data value, that is, its behavioural automaton has a single state, and a transition labelled by $r(v)$ for every data value v, such that $\alpha(r(v)) = \langle c, c, c, \emptyset, \{c \mapsto v\}\rangle$.

Observe that we do not use explicitly the composed connector $LF \bowtie AC$, but LF and AC as independent entities instead, since the Dreams framework can postpone the composition of their labels to runtime. Consider that the AC automaton is in state $q_1(v)$, hence it can perform a step $s_2(v)$, writing a value v to the port c. In this example AC is connected via the ports a, b, and c. The label $s_2(v)$ does not have dataflow on a nor on b, and the reader can perform a label $r(v)$ because $s_2(v) \otimes r(v) \neq \bot$. Using the concurrency predicate in Equation (1), we conclude that $s_2(v) \otimes r(v)$ is in the concurrency predicates of LF and $connector_2$. Furthermore, from the locality property we conclude that all other connectors not attached to AC also allow $s_2(v) \otimes r(v)$ to be executed concurrently. Hence, Dreams can chose to perform this step by analysing only the behaviour of AC and *reader*, depicted by a grey box.

The instantiations of Linda and Reo yield a similar result. The shared tuple-space can communicate with a single process at a time, without synchronising with every other process. Reo can, for example, send data from a full FIFO_1 independently of the behaviour of the connector attached to its sink port. The benchmarks performed for the Dreams framework [18] show optimistic results regarding the use of local steps in synchronous coordination.

6 Conclusion

We introduce behavioural automata to model coordination systems. The three main concepts that underlie behavioural automata are *atomicity*, *composability*, and *dataflow*. We allow a sequence of actions that cannot be interleaved with interfering instructions (atomicity), we construct more complex systems out of building blocks that can be analysed independently (composability), and we represent the data values that are exchanged between components (dataflow).

Behavioural automata unify existing dataflow-oriented models with synchronous constructs by leaving open the definitions of composition of labels and of concurrency predicates. The focus of behavioural automata is on concurrent systems, and on avoiding synchronisation of actions whenever it is unnecessary. By capturing a multitude of coordination models, we allow any of these models to be included in implementations based on behavioural automata, such as the Dreams framework.

As future work, we expect to formally show the correctness of the encodings of Reo and Linda. We

would also like to discover which properties can be shown for behavioural automata that are directly reflected on encoded models. A more practical track of our work involves the development of tools. Further development of Dreams to make it ready for use by a broader community is in our agenda.

References

[1] Farhad Arbab (2004): *Reo: a channel-based coordination model for component composition*. *Mathematical Structures in Computer Science* 14(3), pp. 329–366, doi:10.1017/S0960129504004153.

[2] Farhad Arbab (2005): *Abstract Behavior Types: a foundation model for components and their composition*. *Science of Computer Programming* 55, pp. 3–52, doi:10.1016/j.scico.2004.05.010.

[3] Farhad Arbab (2006): *Composition of Interacting Computations*, chapter 12, pp. 277–321. Springer-Verlag New York, Inc., Secaucus, NJ, USA, doi:10.1007/3-540-34874-3_12.

[4] Farhad Arbab, Roberto Bruni, Dave Clarke, Ivan Lanese & Ugo Montanari (2009): *Tiles for Reo*. In: *Recent Trends in Algebraic Development Techniques*, *LNCS* 5486, Springer, pp. 37–55, doi:10.1007/978-3-642-03429-9_4.

[5] Farhad Arbab, Christian Koehler, Ziyan Maraikar, Young-Joo Moon & José Proença (2008): *Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools*. In: *Proceedings of FACS*, SCP.

[6] Christel Baier, Marjan Sirjani, Farhad Arbab & Jan J. M. M. Rutten (2006): *Modeling component connectors in Reo by constraint automata*. *Science of Computer Programming* 61(2), pp. 75–113, doi:10.1016/j.scico.2005.10.008.

[7] Gérard Berry (2000): *The foundations of Esterel*. In Gordon D. Plotkin, Colin Stirling & Mads Tofte, editors: *Proof, Language, and Interaction*, The MIT Press, pp. 425–454.

[8] Marcello M. Bonsangue, Dave Clarke & Alexandra Silva (2009): *Automata for Context-Dependent Connectors*. In: *COORDINATION*, *LNCS* 5521, Springer, pp. 184–203.

[9] D. M. Chapiro (1984): *Globally-Asynchronous Locally-Synchronous Systems*. Ph.D. thesis, Standford University.

[10] P. Ciancarini, K.K. Jensen & D. Yankelevich (1995): *On the Operational Semantics of a Coordination Language*. In: *Object-Based Models and Languages for Concurrent Systems*, *LNCS* 924, pp. 77–106.

[11] Dave Clarke, David Costa & Farhad Arbab (2007): *Connector colouring I: Synchronisation and context dependency*. *Science of Computer Programming* 66(3), pp. 205–225, doi:10.1016/j.scico.2007.01.009.

[12] Régis Cridlig & Eric Goubault (1993): *Semantics and Analysis of Linda-Based Languages*. In Patrick Cousot, Moreno Falaschi, Gilberto Filé & Antoine Rauzy, editors: *WSA*, *LNCS* 724, Springer, pp. 72–86.

[13] Frederic Doucet, Massimiliano Menarini, Ingolf H. Krüger, Rajesh K. Gupta & Jean-Pierre Talpin (2006): *A Verification Approach for GALS Integration of Synchronous Components*. *ENTCS* 146(2), pp. 105–131.

[14] Fabio Gadducci & Ugo Montanari (2000): *The tile model*, pp. 133–166. MIT Press, Cambridge, MA, USA.

[15] David Gelernter (1985): *Generative communication in Linda*. *ACM Transactions on Programming Languages and Systems* 7(1), pp. 80–112, doi:10.1145/2363.2433.

[16] Christian Krause (2011): *Reconfigurable component connectors*. Ph.D. thesis, Leiden University.

[17] Robin Milner (1983): *Calculi for Synchrony and Asynchrony*. *Theor. Comput. Sci.* 25, pp. 267–310, doi:10.1016/0304-3975(83)90114-7.

[18] José Proença (2011): *Synchronous Coordination of Distributed Components*. Ph.D. thesis, Leiden University.